① **Phases Of Compiler :-** The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

i) **Lexical Analysis :-** Lexical analyzer phase is the first phase of compilation process. It takes source code as input. It reads the source program one character at a time and convert it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens

ii) **Semantic Analysis :-** It is the ~~second~~ third phase of compilation process. It checks whether the parse tree follows the rules of language. Semantic analyzer keeps track of identifiers, their types and expression. The output of semantic analysis phase is the annotated tree syntax.

iii) **Syntax Analysis :-** It is the second phase of compilation process. It takes token as input and generates a parse tree as output. In syntax analysis phase the parser checks that the expression made by the tokens is syntactically correct or not.

iv) **Intermediate Code generation :-** In this, compiler generat the source code into intermediate code.

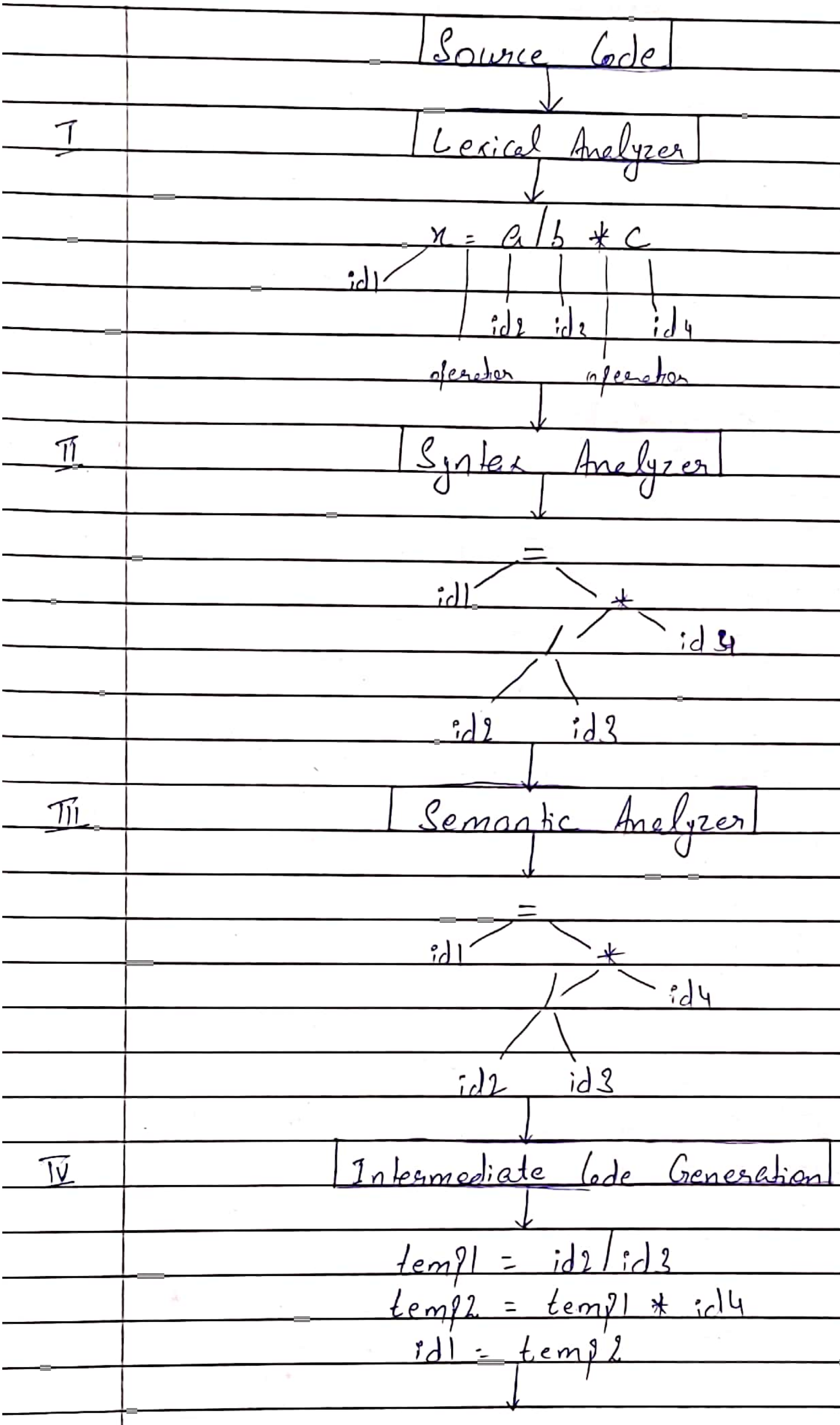
Intermediate code is generated b/w the high level language and the machine language. The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

v) **Code Optimization** : It is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of code and arranges the sequence of statement in order to speed up the program execution.

vi) **Code Generation** : It is the final stage of the Compilation phase. It takes the optimized intermediate code as input and maps it to the target machine language. Code generator translate the intermediate code into the machine code of the specified computer.

Now to understand these phases deeply lets take an example

→ $x = a/b*c$



```
temp1 = id2 / id3
temp2 = temp1 * id4
id1 = temp2
```

V. **Code Optimization**

↓

temp1 = id2 / id3
id1 = temp1 * id4

↓

VI. **Code Generation**

↓

```
Mov   id3, R2
MUL   id2, R2
Mov   id4, R1
Mul   R2, R1
Mov   R1, id1
```

So these are the phases of compiler. With this example you can easily understand all the phases

Q2 Role of Intermediate Code Generation :

i) Because of the machine independent intermediate code, portability will be enhanced. For ex, suppose, if a compiler translates the source language to its target machine code without having the option for generating intermediate code, then for each new machine a full native compiler is required Because obviously there were some modifications in the compiler itself according to the machine specifications.

ii) Retargeting is facilitated.

iii) It is easier to apply source code modificati to improve the performance of source code by Ophimising the intermediate Code

The following are commonly used intermedicte Code representation :-

i) Postfix Notation :- The ordinary way of writing the sum of a and b is with operator in the middle: a+b. The postfix notation for the same expression places the operator at the right end as ab+. In general if e1 and e2 are any postfix expression and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation

ii) Three Address Code :- A statement involving no more than three references is known as three address statement A sequence of three address statement is known as three address Code. Three address statement is of the form x = y op z here x, y, z will have address Sometimes a statement might Contain less than three reference but it is still Called three address statement.

iii) Syntax Tree :- It is nothing more than Condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents And a chain of single production is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands To form syntax tree put paranthesis in the expression this way it's easy to recognize which operand come first.

Backpatching :- The easiest way to implement the syntax directed definition for boolean expression is to use two passes first construct a syntax tree for the input and then walk tree in depth-first order, computing the translations The main problem with generating code for boolean expression and flow of control statement in a single pass is that during one single pass we may not know the label that control must go to at the time the jump statement are generated Hence a series of branching statement with the target of the jumps left unspecified is generated. Each statement will be put on a list of goto statement whose label will be filled in when the proper label can be determined We call this subsequent filling in of labels backpatching.

To manipulate lists of label we use function

i) make list(i)

ii) merge (p1, p2)

iii) backpatch (p, i)

③ **Peephole Optimization** :- It is a type of code optimization performed on a small part of the code It. is performed on the very small set of instructions in a large code.

Objectives :- i) To improve performance
ii) To reduce memory footprint
iii) To reduce code size.

Techniques :-

i) **Redundant load and store elimination** :- In this, technique the redundancy is eliminated.

```
y = x + 5;
i = y ;
z = i ;
w = z * 3;
```

Optimized Code :-

```
y = x + 5;
w = y * 3;
```

ii) **Constant Folding** :-
The code that can be simplified by user itself.

```
x = 2 * 3;
```

Optimized Code :-

```
x = 6 ;
```

iii) **Strength Reduction :** The operators that Consume higher execution time are replaced by the operators consuming less execution time.

y = x * 2 ;

Optimized Code :

y = x + x

iv) **Null Sequences :-** Useless operations are deleted.

④ **Optimization :** It is the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources so that faster running machine code will result.

⇒ **When To Optimize :-** Optimization of code often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance.

⇒ **Why To Optimize :** It may involve reducing the size of the code. It helps to

- Reduce the space consumed and increase the speed of compilation.
- An optimized code often promote re-usability

⇒ **Types Of Optimization :**

i) **Machine Independent :-**
It attempts to improve the intermediate code to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU register or absolute memory location

ii) **Machine Dependent :-**
It is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory reference rather than relative reference. It put efforts to take maximum advantage of the memory hierarchy

⇒ **Where To Apply Optimization :**

- **Source Program :-**
Optimizing the source program involves making changes to the algorithm or changing the loop structures

- Intermediate Code :

Optimizing the intermediate code involves changing the address calculation and transforming the procedure calls involved

- Target Code :
Optimizing the target code is done by the compiler. Usage of registers, select and move instructions is part of optimization involved in the target code

⇒ Phases Of Optimization :-

- Global Optimization :
Transformations are applied to large program segments that includes functions, procedure and loops

- Local Optimization :
Transformations are applied to small blocks of statements. The local optimization is done prior to global optimization.

(5) I Constant Propagation :-
It is the process of substituting the value of known constants in expressions. Constant propagation eliminates cases in which values are copied from one location to another, in order to simply assign value to other variable

For example consider the code

```
x ← 14
y ← 7 - x / 2
z ← y * (28 / x + 2) - x
```

Here x is assigned a constant and thus can be propagated

```
x ← 14
y ← 7 - 14 / 2
z ← y * (28 / 14 + 2) - 14
```

Constant propagation enables the code to assign static values, which is faster than looking up and copying the value of a variable and also saves time by eliminating assigning a value to a variable that is itself subsequent used only to propagate the value throughout the code.

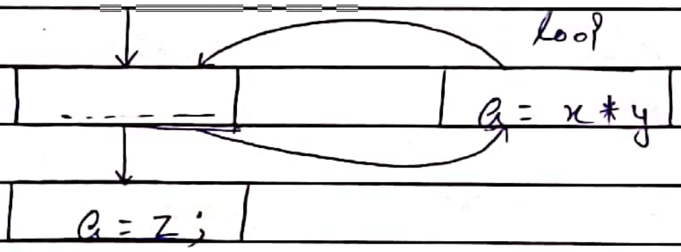
In some cases, copy propagation itself may not provide direct optimization but simply facilitates other transformations such as constant folding, code motion and dead code elimination.

II **Dead Code Elimination** :- Dead code is one or more than one code statement, which are :

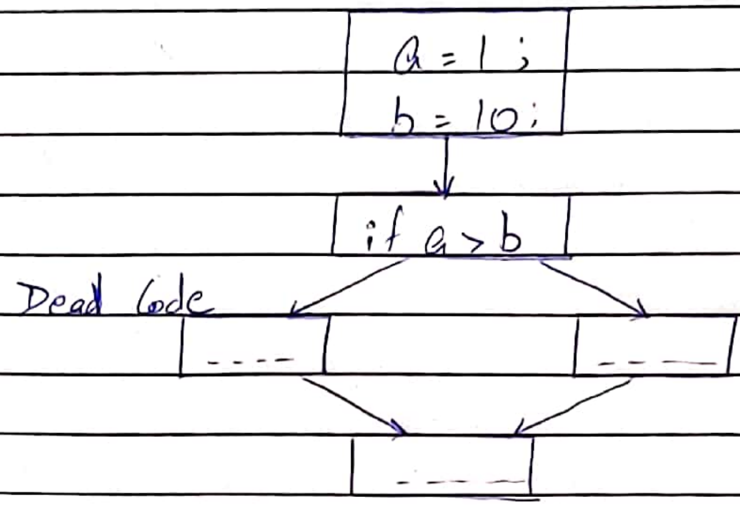
- Either never executed or unreachable.
- if executed, their output is never used.

Thus dead code plays no role in any program operation and therefore it can simply be eliminated.

There are some code statement whose computed values are used only under certain circumstances i.e sometimes the values are used and sometimes they are not. Such codes are known as partially dead-code



The above control flow graph decides a chunck of program where variable 'a' is used to asign the output of expression 'x*y'. Let us assume that the value assigned to 'a' is never used inside the loop. Immediately after the control leaves the loop, 'a' is assigned the value of variable 'z', which would be used later in the program. We conclude here that the assignment code of 'a' is never used anywhere, therefore it is eligible to be eliminated.



Like wise the picture above depicts that the conditional statement is always false implying that the code, written in true case, will never be executed, hence it can be removed.

III Loop Optimization &

It is the process of increasing execution speed and reducing the overheads associated with loops. It play an important role in improving cache performance and making effective use of parallel processing capabilities

Techniques &

i) Code Motion &

In this, amount of code in loop is decreased. A statement or expression which can be moved outside the loop body without affecting the semantic of the program is moved outside the loop.

```
while (i<100)
{
   a = Sin(x)/Cos(x) + i;
   i++;
}
```

Optimized Code :-

```
t = Sin(x)/Cos(x);
while (i<100)
{
   a = t+i;
   i++;
}
```

ii) Loop Unrolling :- In this, we remove or reduce the iterations

```
for (int i=0; i<5; i++)
     printf("Neme\n");
```

Optimized Code :-

```
printf("Neme\n");
printf("Name\n");
printf("Neme\n");
printf("Neme\n");
printf("Name\n");
```

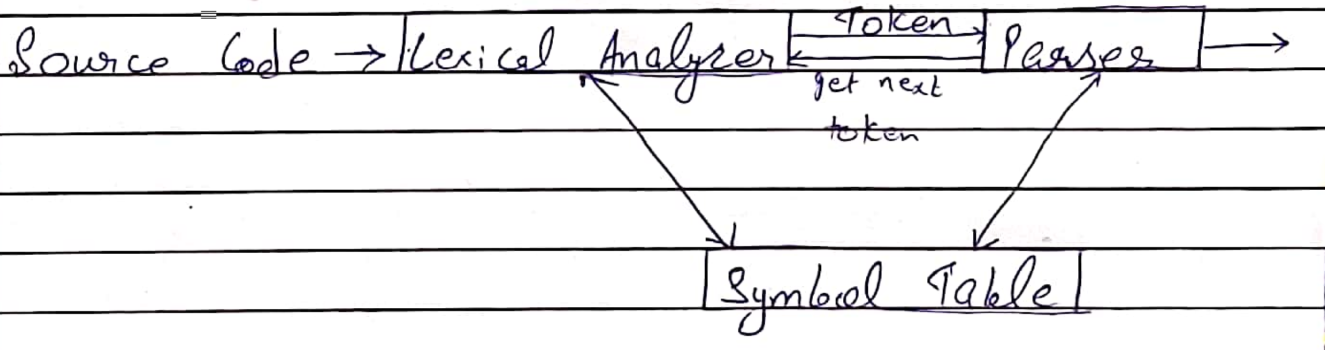
iii) Loop Jemming :- It is the Combining the two or more loops in a single loop It reduces the time taken to compile the many no. of loops.

```
for(int i=0; i<5; i++)
     a = i+5;
for(int i=0; i<5; i++)
     b = i+10;
```

Optimized Code :

```
for(int i=0; i<5; i++)
{
   a = i+5;
   b = i+10;
}
```

(6) Lexemes are recognized in lexical analyzer ie


Source Code
Lexical Analyzer
Token
get next token
Parser
Symbol Table