Chapter 7

Code Generation

* **Code Generation -**

- Code generation is the final activity of compiler.
- Code generation is a process of creating assembly language statements which will perform the operations specified by the source program when they run.
- It takes intermediate code as an input & generates target machine code as output.

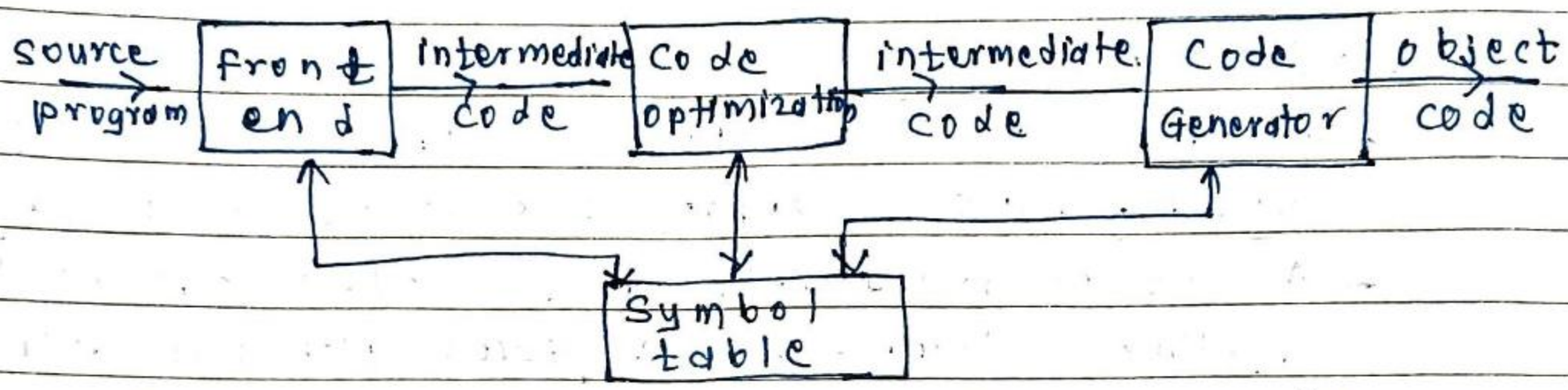


fig. position of code generator in compiler.

Properties of Code generation phase are

a) Correctness
b) high quality
c) efficient use of resources of the target machine
d) quick code generation.

* **Object Code forms**

The output of code generation is an object code or machine code. This code normally comes in following forms.

1) absolute code 2) relocatable machine code 3) assembler code

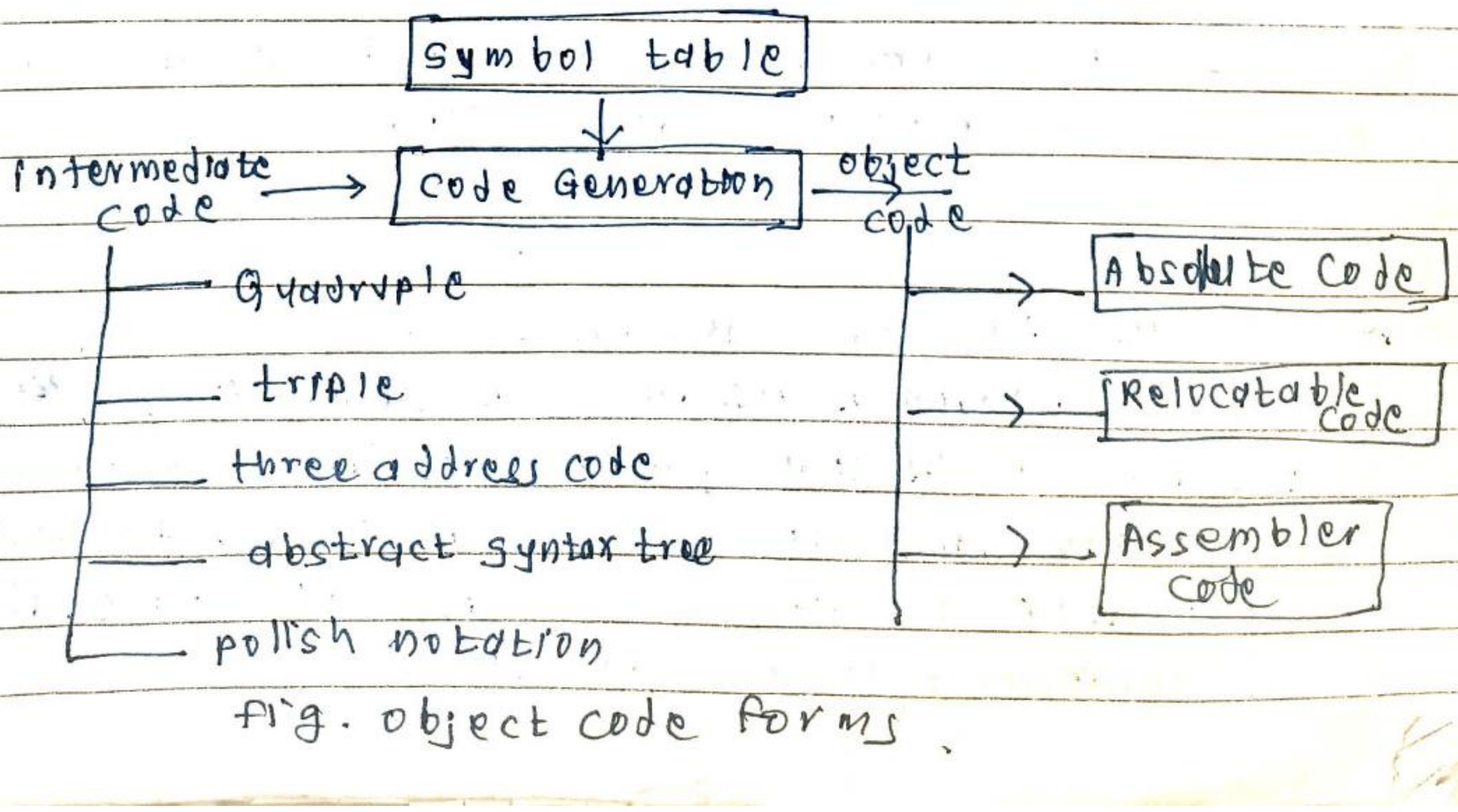


fig. object code forms

1) Absolute code -

- Absolute code is a machine code that contains reference to actual addresses within program's addr
- The generated code can be placed directly in the memory & execution starts immediately.
- Generally small programs can be compiled & executed quickly because of absolute code genera ex. WATFIIV & PL/C compilers.

2) Relocatable code -

- Producing a relocatable machine language program output. allows subprograms to be compiled sepa
- A set of relocatable object modules can be link together & loaded for execution with the help of linking loader.
- the advantage of generating relocatable machine language code is that we gain a great deal of flexibility in being able to compile subroutine separately & to call. other previously compiled programs from an object code.

3) Assembler code -

- Producing an assembly language program as output make the process of code generation somewhat eas
- We can generate symbolic instructions & use the ma facilities of the assembler to help in generati of code.
- But generating assembler code as an output ma code generation process slower.

* Issues in Code Generation -

some common issues in code generation.

1) Input to the code generator
2) Target programs.
3) Memory management
4) Instruction selection
5) Register allocation
6) choice of evaluation
7) approaches to code generation

1) Input to the code generator -

- The code generation phase takes intermediate code as input.
- The intermediate code along with the symbol table information is used to determine the runtime addresses of the data objects:
- These data objects are denoted by the names in the IR.
- The intermediate code generated by the front end should be such that the target machine can easily manipulate it, in order to generate appropriate target machine code.
- In the front end of compiler necessary type checking & type conversion needs to be done.
- The detection of the semantic errors should be done before submitting the input to the code generator.
- The code generation phase requires the complete error free intermediate code as input.

2) Target programs -

The output of code generator is target code. Typically the target code comes in three forms such as absolute machine language, relocatable machine language & assembly language.

3) memory management -

- Both the front end & code generator performs the task of performing the name in the source program to addresses to the data objects in run time memory.
- The names used in the three address code refer to the enteries in the symbol table.
- The type in a declaration statement determines the amount of storage needed to store the declared identifier.
- Thus using the symbol table information about memory requirements code generator determing the addresses in the target code.
- Similarly if three address code contains the labels then those labels can converted into equivalent memory addresses.

4) Instruction selection -

- The selection of instruction depends upon the instruction set of target machine.
- The speed of instruction & machine idioms are two important factors in selection of instruction.
- If we don't consider the efficiency of target code then the instruction selection becomes straightforward task.
- For each type of three address code the code skeleton can be prepared which ultimately gives the target code for the corresponding const

ex. $x = a + b$ then the code seqn can be generat

```
MOV   a, R0
Add   b, R0
Mov   R0, x
```

In above example the code is generated line by li Such a line by line code generation process gener the poor code because the redundancies can be achieved by subsequent lines & these redundanc cant be considered in the process of line by li code generation.

ex. $x := y + 2$

$a := x + t$

The code for the above statements can be generated

```
MOV   y, R0
Add   z, R0
MOV   R0, x
MOV   x, R0
Add   t, R0
MOV   R0, a
```

The above generated code is poor code because, MOV R is not used & statement MOV a, R0 is redunda

```
MOV   y, R0
Add   z, R0
Add   t, R0
Mov   R0, a
```

The quality of generated code is decided by its sp & size.

5) Register allocation -

- If the instruction contains register operands then such a use becomes shorter & faster than that of using operands in the memory.
- Hence, while generating a good code efficient utilization of register is an important factor.
- There are two important activities done while using registers.

1) Register allocation -
During register allocation, select appropriate set of variables that will reside in registers.

2) Register assignment -
During register assignment, pick up the specific register in which corresponding variable will reside.
Obtaining the optimal assignment of registers to variable is difficult.

6) Choice of evaluation order -

- The evaluation order is an important factor in generating an efficient target code.
- Some orders require less number of registers to hold the intermediate results than the others.
- Picking up the best order is one of the difficulty in code generation
- Mostly we can avoid this problem by referring the order in which the three address code is generated by semantic actions.

7) Approaches to code generation.

- The important factor for code generation is that it should produce the correct code. With this approach of code generation various algorithm for generating code are designed.

* Target machine Description -

- for designing the code generator it is necessary to have prior knowledge of target machine & instruction set used for this target machine.
- The target machine code is a register machine like minicomputers.
- Following assumptions are made for code generation.

a) we will assume that in target computer addresses are given in bytes & four bytes form a word.

b) There are n general purpose registers R0, R1, - - - Rn-1.

c) The two address instruction is of the form

op source destination

where op is an opcode & source & destination are data fields.

ex.

MOV - Moves from source to destination.

ADD - Add source to destination.

SUB - substracts ~~from~~ source from destination.

The source & destination are specified by registers & memory locations.

- The addressing modes used are as follows.

Addressing mode	Form	Address	Added cost
Absolute	M	M	1
Register	R	R	0
Indexed	c(R)	c + contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*c(R)	contents(c+contents(R)	1
Literal	#c	c	L

ex. The instruction MOV R1, M stores the contents of Register R1 into memory location M.

* Basic Blocks —

- The basic block is a sequence of consecutive statements in which flow of control enters at the beginning & leaves at the end without halt or possibility of branching.

ex. The basic block is as shown below.

```
t1 := a * 5
t2 = t1 + 7
t3 = t2 - 5
t4 = t1 + t3
t5 = t2 + b
```

Terminologies used in basic blocks -

1) Define & use -

The three address statement a := b + c is said to define a & use b & c

2) Live & dead -

The name in the basic block is said to be live at a given point if it's value is used after that point in the program.

The name (variable) is said to be dead at a given point if its value is never used after that point in the program.

* Algorithm for partitioning into blocks -

- Any given program can be partitioned into basic blocks by following algorithm.
- We assume that an intermediate code is already generated for the given program.

1) First determine the leaders by following rules.
   a) The first statement is a leader
   b) Any target statement of conditional or unconditional goto is a leader.
   c) Any statement that immediately follow a goto or unconditional goto is a leader.
2) The basic block is formed starting at the leader statement & ending just before the next leader statement appearing.

Q. Consider the following program code for computing dot product of two vectors a & b of length 10 & partition it into basic blocks.

```
prod = 0;
i = 1;
do
{
  prod = prod + a[i] * b[i];
  i = i+1;
} while (i<=10);
```

→ First we will write the equivalent three address code for the above program.

```
1. prod=0;
2. i=1;
3. t1 = 4*i
4. t2 = a[t1]
5. t3 = 4*i
6. t4 = b[t3]
7. t5 = a[t2] * b[t4]
8. t6 = prod + t5
9. prod = t6
10. t7 = i+1
11. i := t7
12. if i<=10 goto(3)
```

Block 1

```
1. prod := 0
2. i := 1
```

Block 2

```
3. t1 = 4*i
4. t2 = a[t1]
5. t3 = 4*i
6. t4 = b[t3]
7. t5 = t2 * t4
8. t6 = prod+t5
9. prod = t6
10 t7 := i+1
11 i = t7
12. if i<=10 goto (3)
```

According to the algorithm statement 1 is a leader by rule 1 (a)
statement 3 is also leader by rule 1 (b)

hence, statement 1 & 2 form the basic block.
similary statement 3 to 12 from another basic block.

Flow Graph –

- A flow graph is a directed graph in which the flow control information is added to the basic blocks.
- The nodes to the flow graph are represented by basic blocks
- The block whose leader is the first statement is called initial block.
- There is a directed edge from block B_1 to Block B_2 if B_2 immediately follows B_1 in the given seqn. We can say that B_1 is a predecessor of B_2
- The flow graph for the above code can be drawn as follow.

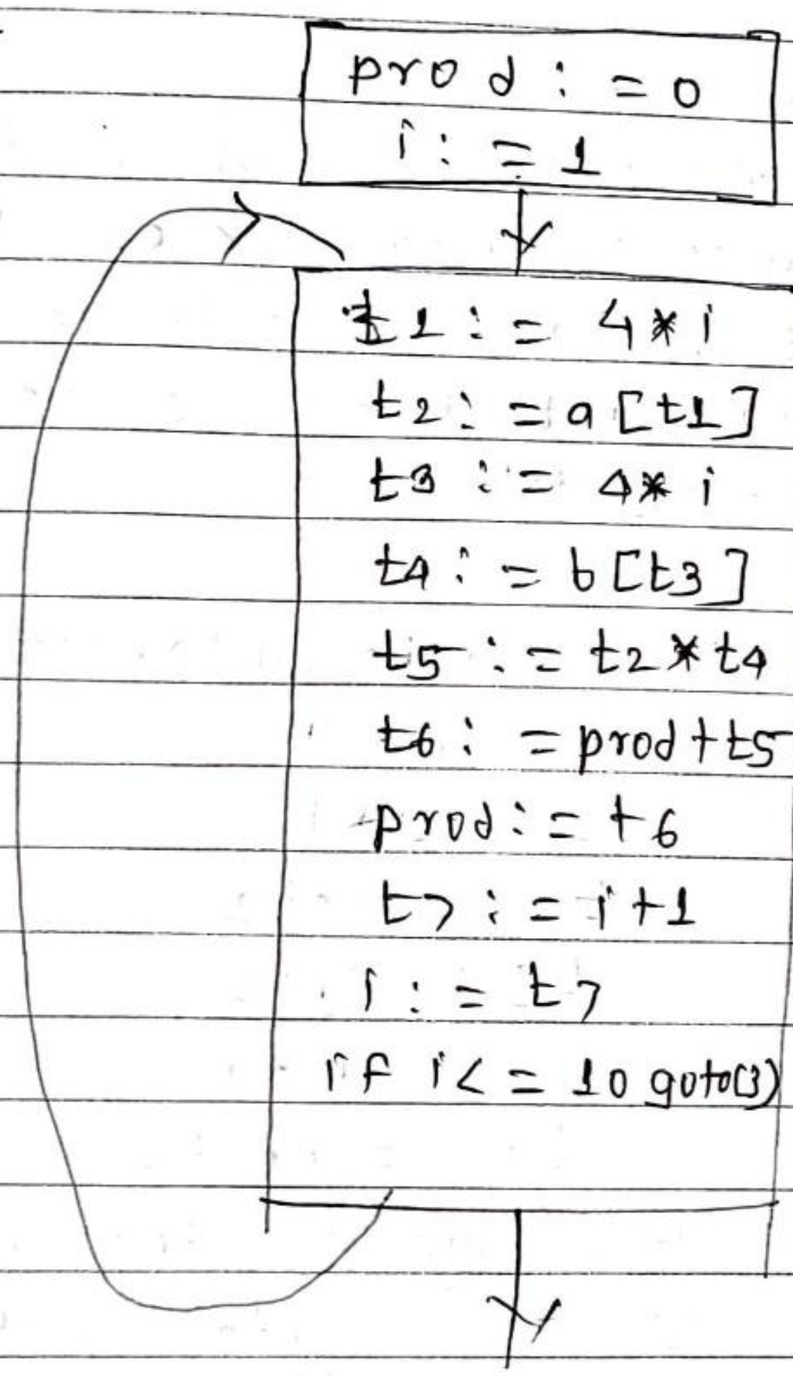


fig. flow graph.

* Next - Use information —

The next-use information is a collection of all the names that are usefull for next subsequent statem in a block.

ex. Consider a statement.

x := i

j := x op y

That means the statement j uses value of x

- The next use information can be collected by making the backward scan of the programming co in that specific block.

Storage for Temporary Names -

For the distinct names each time a temporary is n And each time a space gets allocated for each tem To have opthmization in the process of code generat we can pack two temporaries into the same location if they are not live simultaneously.

ex. Three address code as	This can be packed into two
$t_1 := a * a$	$t_1 := a * a$
$t_2 := a * b$	$t_2 := a * b$
$t_3 := 4 * t_2$	$t_2 := 4 * t_2$
$t_4 := t_1 + t_3$	$t_1 := t_1 + t_2$
$t_5 := b * b$	$t_2 := b * b$
$t_6 := t_4 + t_5$	$t_1 := t_1 + t_2$.

Many times the temporaries be packed into registers than memory locations.

* Ex. Computing next use information L: a := b op c

-> i) The currently found information in symbol table regarding the next use & liveness of a, b & c is associated with the statemet L

ii) In the symbol table set 'a' to not live & no next use.

iii) set 'b' & 'c' to live & next use of b & c in symbol table

* Register Allocation & Assignment -

- The use of register operands instead of memory operands is always faster & shorter it also help for generating good code.
- There are certain strategies can adopted by compiler for register allocation & assignment.
- The most commonly used strategy to register allocation & assignment is to assign specific values to specific registers.

ex. for base address of separate register can be used arithmatic computations can be done by separate registers.

Advantage -

The design process of compiler for code generation becomes simplified.

Disadvantage -

Design of compilers become complicated because of restrictive use of registers. because at ~~certain~~ same time certain set of registers remain totally unused over substantial portions of code & some set of registers get overloaded.

The strategies used in register allocation & assignment

1) Global register allocation -

While generating the code the registers are used to hold the values for the duration of single block. All the live variables are stored at the end of each block. for the variables that are used consistenly we can allocate specific set of registers.

Hence, allocation of variables to specific registers that is consistent across the block boundaries is called global register allocation.

The strategies adopted while doing the global register allocation

1) The global register allocation has a strategy of storing the most frequently used variables in fixed registers throughout the loop.
2) Another strategy is to assign some fixed number of global registers to hold the most active values in each inner loop.
3) The registers not already allocated may be used to hold values local to one block.

2) Usage Count -

The usage count is the count for the use of some variable x in some register used in any basic block. The usage count gives the idea about how many units of cost can be saved by selecting a specific variable for global register allocation.

The approximate formula for wage count for the loop L in some basic block B can be given as

$$\sum_{\text{block } B \text{ in } L} (use(x, B) + 2 * live(x, B))$$

where use (x, B) is number of times x used in block B prior to any definition of x & live(x, B) = 1 if x is live on exit from B, otherwise live(x) = 0.

ex. Consider, a block B_1, B_2, B_3, B_4 & count the wage count for block B in following loop L.

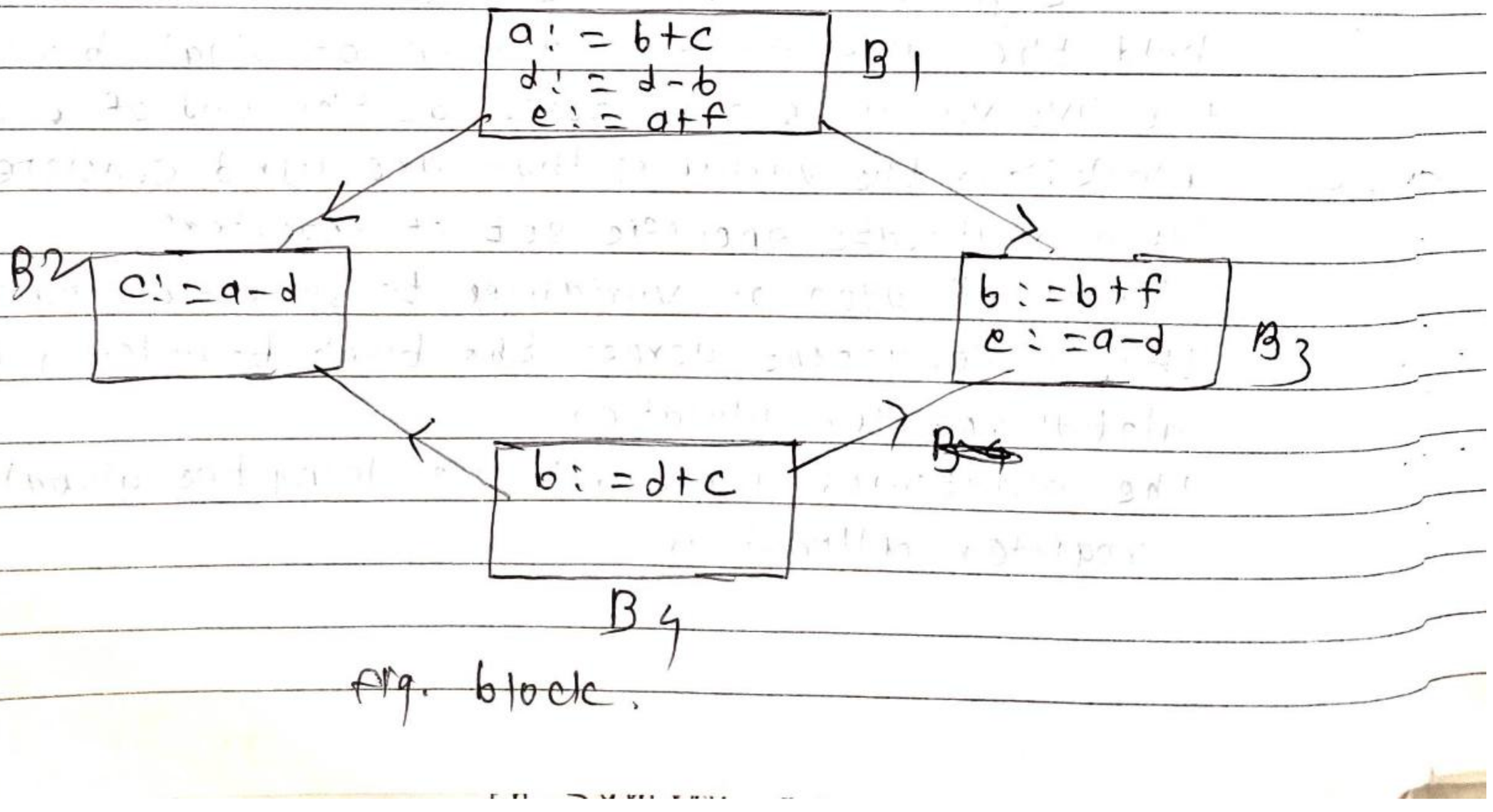


fig. block.

The usage count for block B_1 for variable a is -

$use(a, B_1) = 0$ ∵ a is defined in B_1 before used in define.

$2 * live(a, B_1) = 2$ ∵ a is live on exit of B_1 hence $(a, B) \neq 1$

$\therefore (use(a, B_1) + 2 * live(a, B)) = 2$

The usage count for block B_2 & B_3 for variable a is

$use(a, B_2) = use(a, B_2) = 1$ ∵ a is used in B_1 before define

$2 * live(a, B_2) = 0$ ∵ a is not live on exit of B_2

$$\sum_{B \text{ in } L} use(a, B) = 2$$

$$\therefore \sum_{B \text{ in } L} use(a, B) + 2 * live(a, b) = 2 + 2 = 4$$

∴ count of a is 4. That means compiler can save 4 units of cost by selecting for the global register allocation.

3) Register Assignment for outer loop

Consider that there are two loops L_1 is outer loop & L_2 is an inner loop. And allocation of varidble a is to be done to some register. The approximate scenario is as given

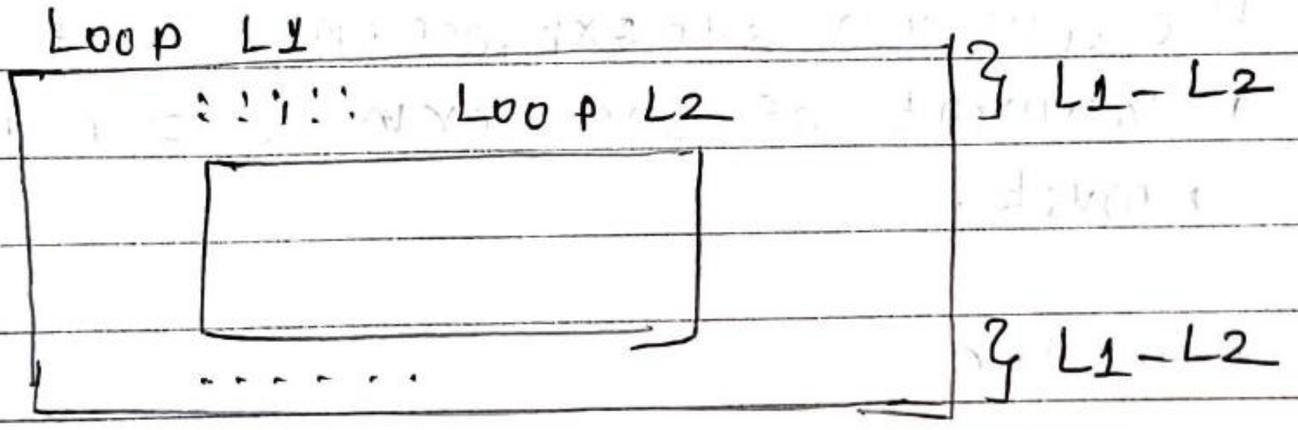


fig. Nested loops

Following criteria should be adopted for register assignment for outer loop.

1) If a is allocated in loop L_2 then it should not allocated in $L_1 - L_2$.

2) If a is allocated in L_1 & it is not allocated in L_2 then store a on a entrance to L_2 & load a while leaving

3) If a is allocated in L_2 & not in L_1 then load a on entrance of L_2 & store a on exit from L_2

* DAG Representation of Basic blocks

- The directed acyclic graph is used to apply transformat on the basic block.
- To apply the transformations on basic block a DAG is constructed from three address statement.
- A DAG can be constructed for the following type of labels on nodes.
  a) Leaf nodes are labelled by identifiers or variable names or constants.
  b) Interior node store operator values.
- The DAG & flow graphs are two different pictorial representations. Each node of the flow graph can be represented by DAG. because each node of the flow graph is a basic block.

Applications of DAG -

- Determining the common sub-expressions.
- Determining which names are used inside block & computed outside the block.
- Determining which statements of the block could have their computed value outside the block.
- simplifying the list of quadruples by eliminating the common subexpressions & not performing the assignment of the form $x := y$ unless & until it is a must.

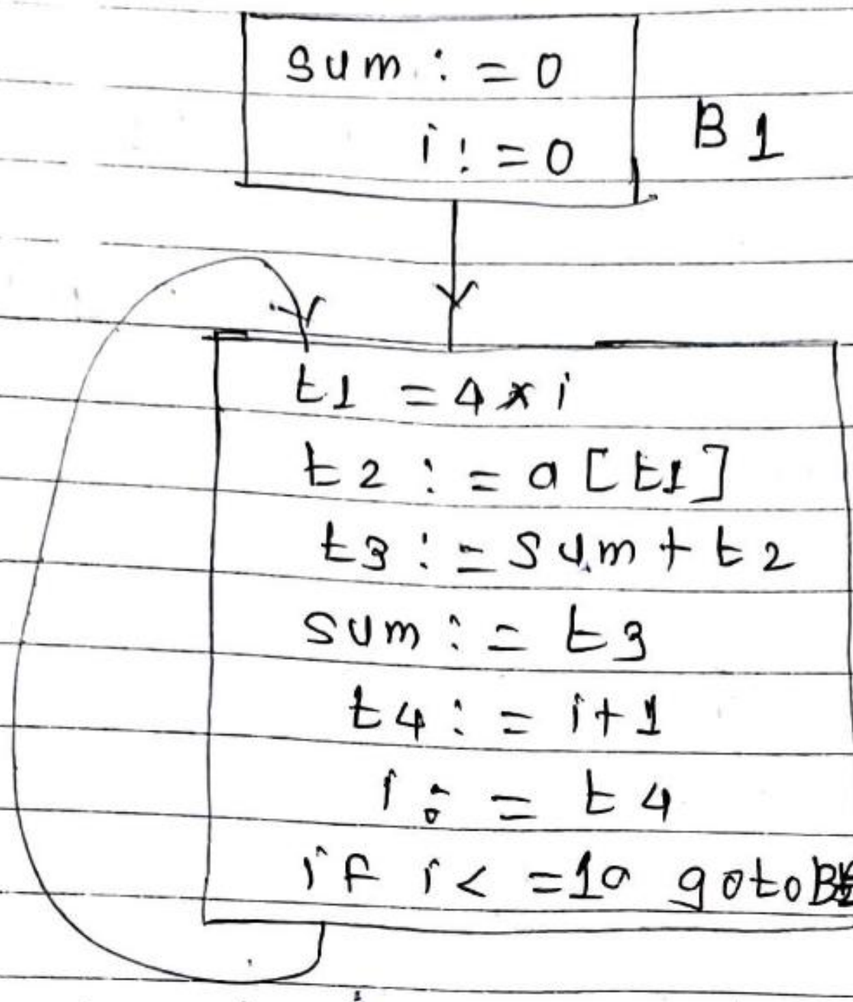
ex. Consider

```
sum := 0;
for(i=0; i<=10; i++)
    sum = sum + a[i];
```

→ The three address code for above is

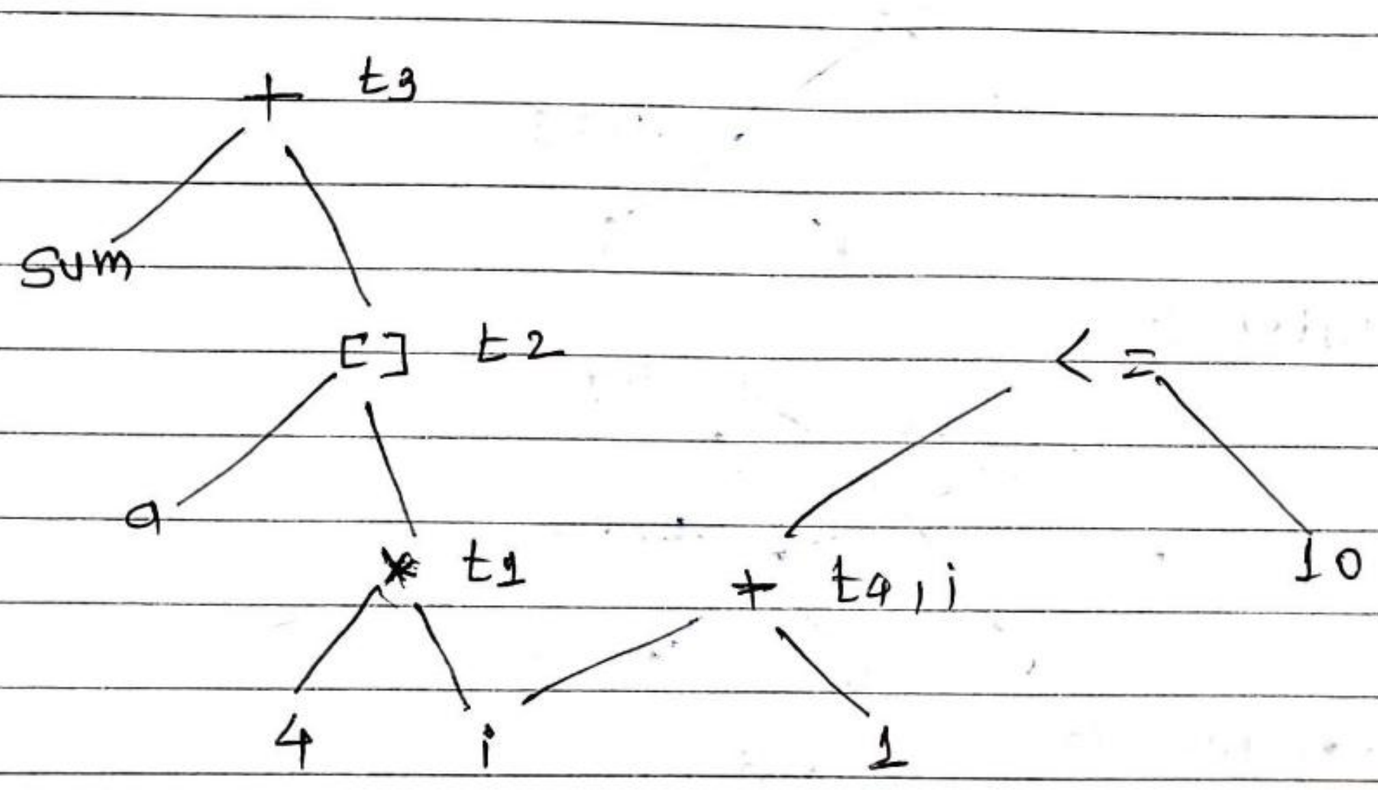
1) sum := 0
2) i = 0
3) $t_1 = 4 * i$
4) $t_2 := a[t_1]$
5) $t_3 := sum + t_2$
6) sum := t_3
7) $t_4 := i + 1$;
8) i := t_4
9) if i <= 10 goto (3)

We can partition above code into basic blocks as follows.



basic blocks

Now, let us, consider block B_2 for construction of DAG by numbering it.



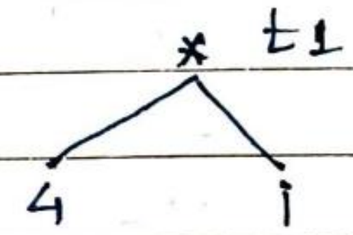
2.2. Write an algorithm for constructing a DAG for the following statements.

1) $t_1 = 4 * i$ 2) $t_2 = a[t_1]$ 3) $t_3 = 4 * i$
4) $t_4 = b[t_3]$ 5) $t_5 = t_2 * t_4$ 6) $p + t_5$
7) $p = t_6$ 8) $t_7 = i + 1$ 9) $i = t_7$
10) if $i < 20$ goto (1)

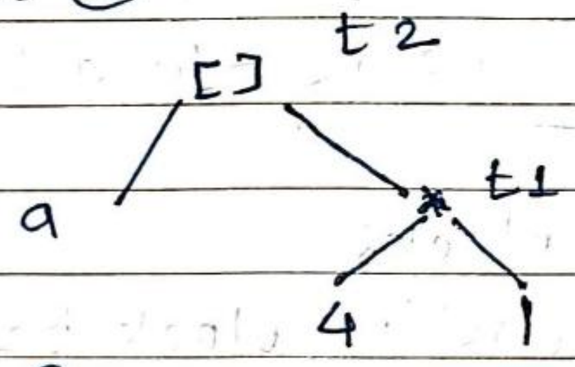
→ Letus we construct dag for following code

$t_1 = 4 * i$ —(1)
$t_2 = a[t_1]$ —(2)
$t_3 = 4 * i$ —(3)
$t_4 = b[t_3]$ —(4)
$t_5 = t_2 * t_4$ —(5)
$t_6 = p + t_5$ —(6)
$p = t_6$ —(7)
$t_7 = i + 1$ —(8)
$i = t_7$ —(9)
if $i < 20$ goto (1) —(10)

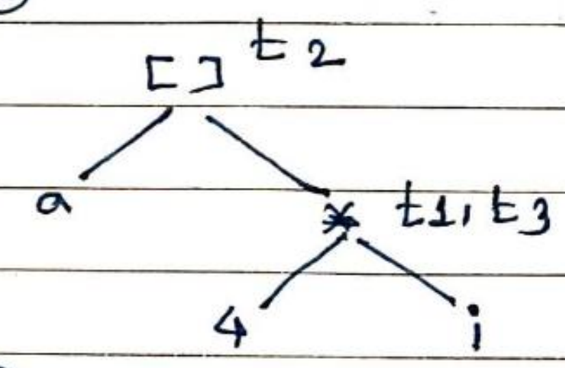
After statement (1)



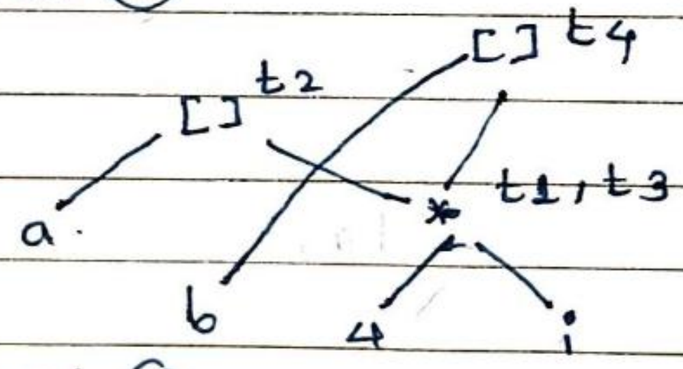
After statement (2)



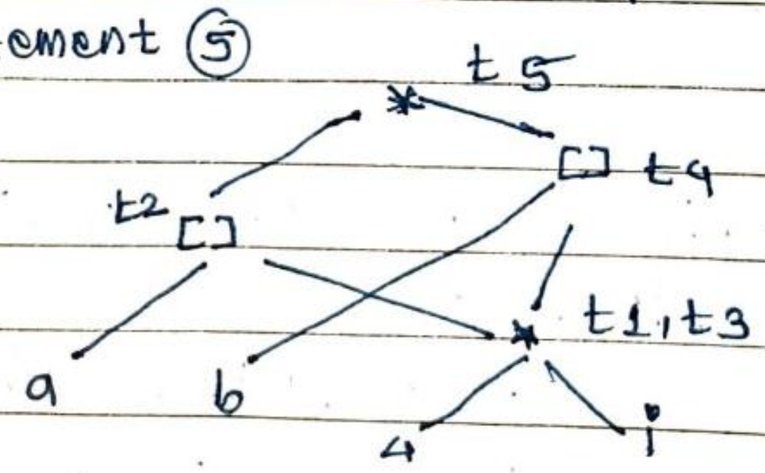
after statement (3)



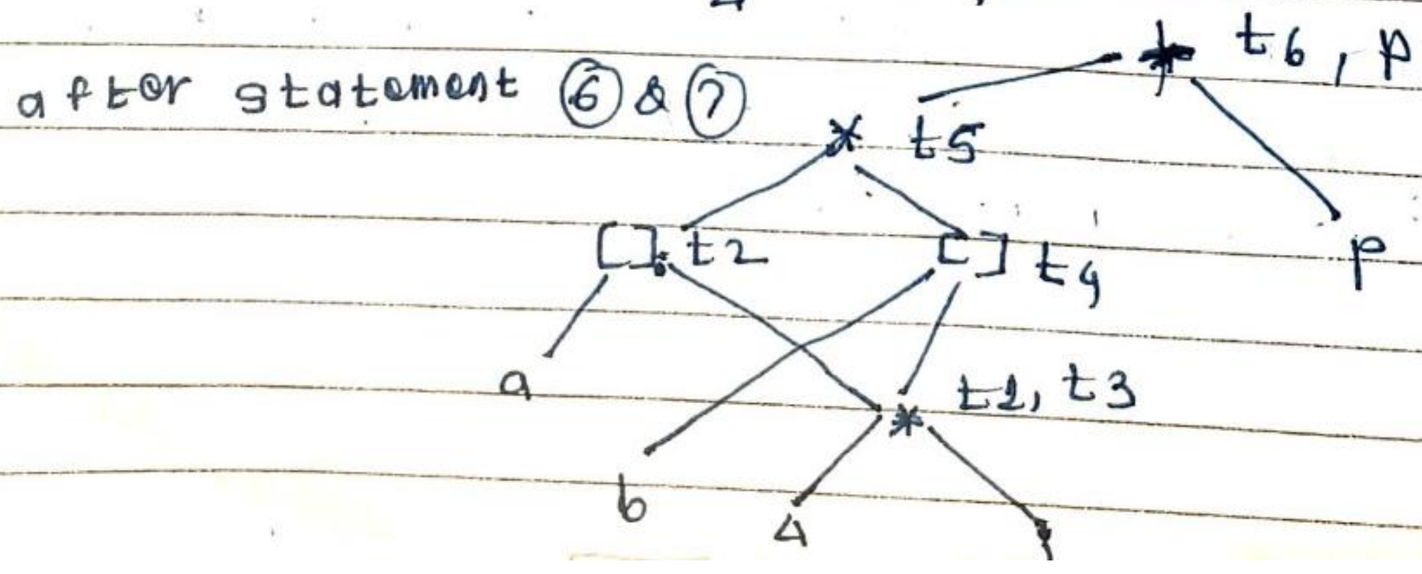
after statement (4)



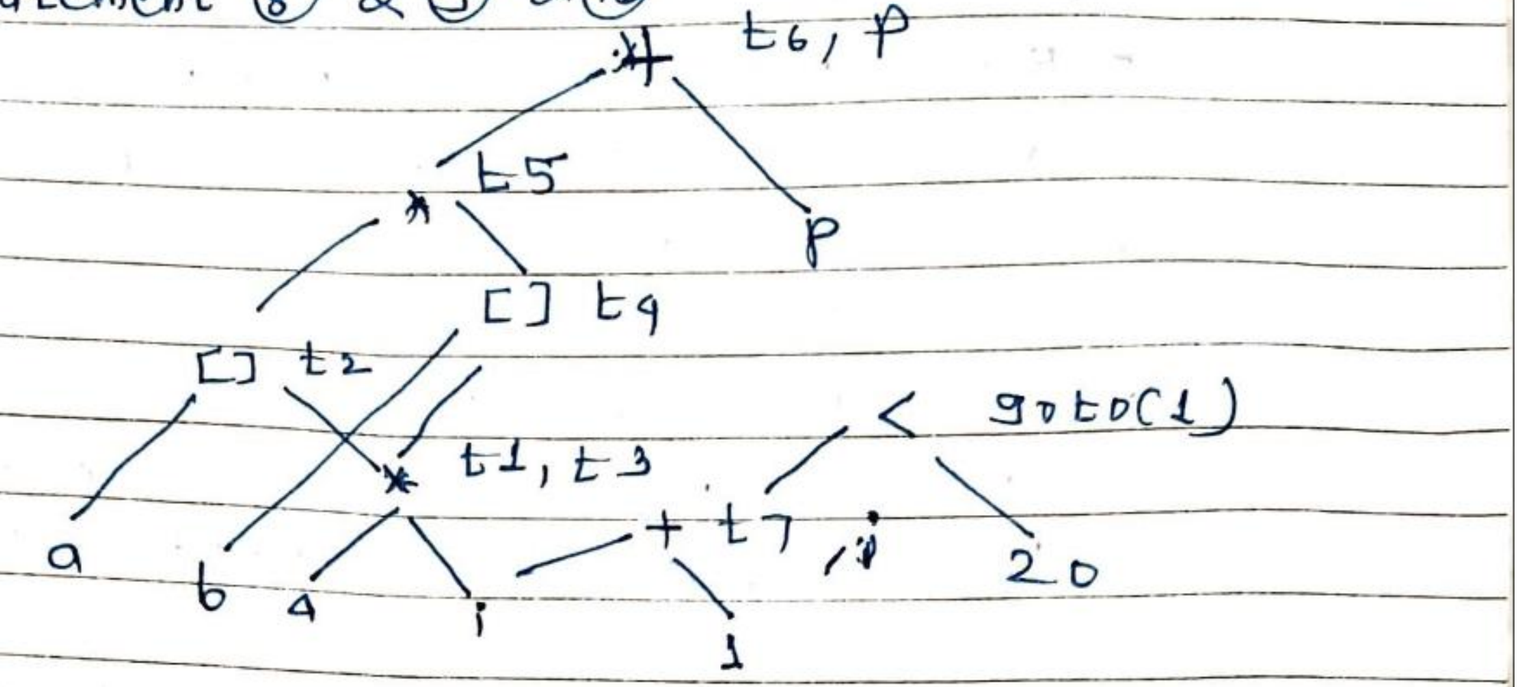
after statement (5)



after statement (6) & (7)



after statement (8) & (9) & (10)



* Peephole Optimization -

Need of peephole optimization -

If we apply statement by statement code generation strategy then the generated target code may contain many redundant instructions. The quality of such code is very poor. To optimize such target code certain transformations need to be applied on the target code. These transformations ultimately will result in getting the significant improvement over the running time or space requirement of the target program.

•Definition -

Peephole optimization is a simple & effective technique for locally improving target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions & replacing these instructions by shorter or faster sequence.

Characteristics of Peephole Optimization -

- The peephole optimization can be applied on the target code using following characteristics
  1) Redundant Instruction elimination
  2) Flow of control optimization
  3) Algebric simplification
  4) use of machine idioms

1) Redundant instruction elimination -
- Especially the redundant loads & stores can be eliminated in this type of transformations

ex. MOV R0, x
MOV x, R0

We can eliminate the second instruction since x is already in R0. But if (MOV x, R0) is a label statement then we cannot remove it.

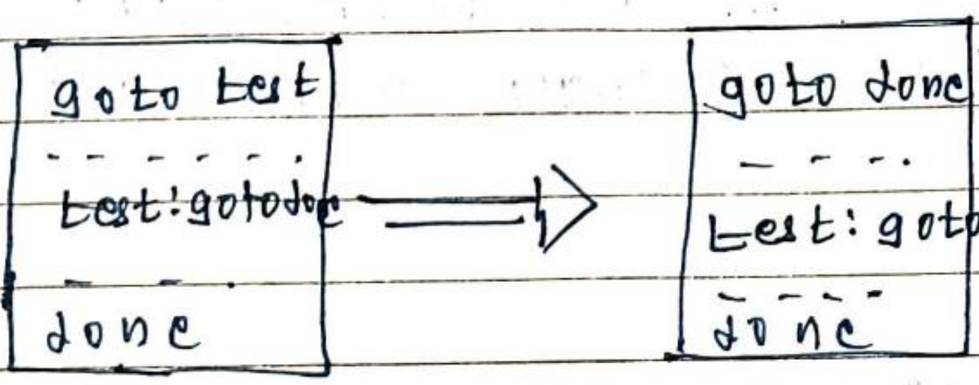
- We can remove unreachable instructions.

ex. sum = 0
if(sum)
printf ("%d", sum);

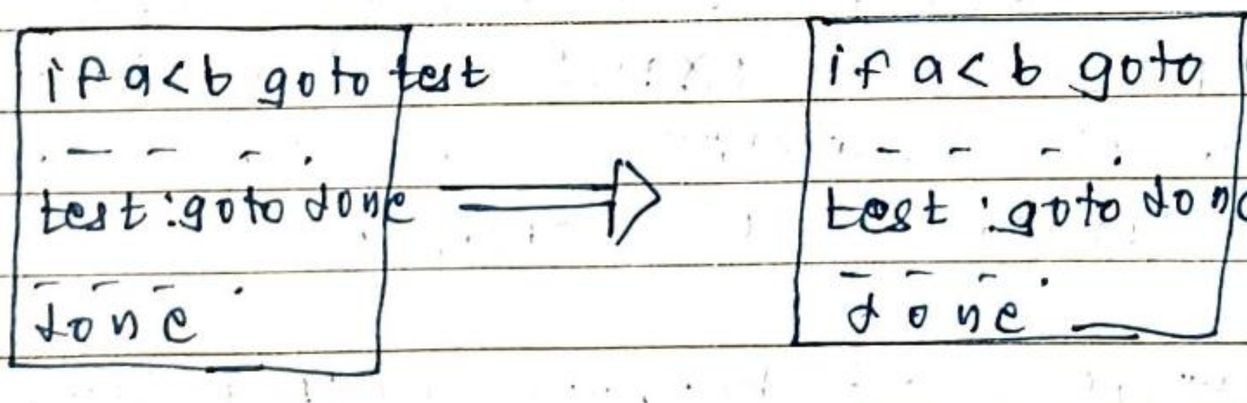
Now, this if statement will never get executed hence, we can eliminate such a unreachable code

2) Flow of control optimization -
- Using peephole optimization unnecessary jump on jumps can be eliminated

ex.



Thus, we reduce one jump by this transformation
Another example could be



3) Algebric simplification -
Peephole optimization is an effective technique for algebric simplification.
The statements such as
x := x+0 or x := x*1
can be eliminated by peephole optimization

4) Reduction in strength -
- Certain machine instructions are cheaper than the other.
- In order to improve the performance of intermedi code we can replace these instructions by equivalent cheaper instruction
ex. x^2 is cheaper than $x * x$

5) Machine idioms -
- The target instructions have equivalent machine instructions for performing some operations.
- Hence, we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.
ex. Some machines have auto increment or auto decrement addressing modes that are used to perform addor substract operations.

* Simple Code Generator -
- In this method computed results can be kept in registers as long as possible.
ex. $x = a + b$
The corresponding target code is,

```
ADD  b, Ri    here Ri holds value of a
              here cost = 2
        OR
MOV  b, R1    here R0 holds value of a
ADD  R1, R0   here cost = 2
```

- The code generator algorithm uses descriptors to keep track of register contents & addresses for names
1) A register descriptor is used to keep track of ~~which~~ what is currently in each register. The register descriptor show that initially all the registers are empty. As the code generation for the block progresses the registers will hold the values of computations.

The address descriptor stores the location where the current value of the name can be found at run ti
The information about locations can be stored in the symbol table & is used to access the variables.
- The address descriptor has following fields.

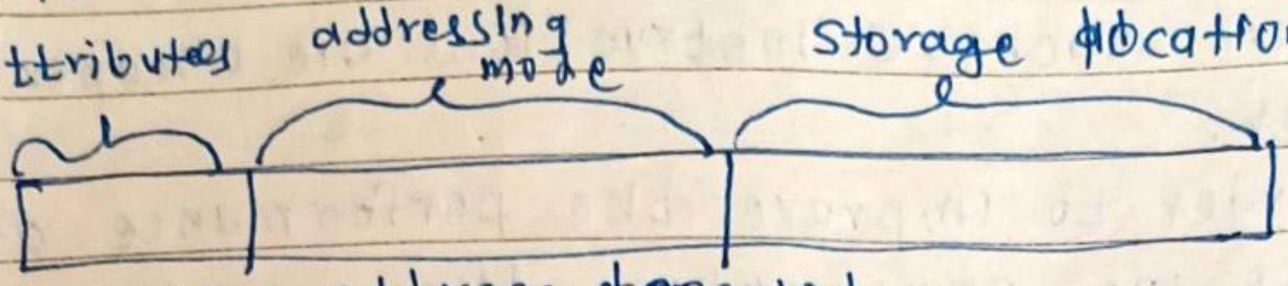


fig. address descriptor.

Attributes - mean type of the operand. It generall refers to the name of temporary variables.
Addressing mode - Indicates whether the addresses are type 'S', 'R', 'IS', 'IR'
The third field is location field which indicates whe the address is in storage location or in register.

The modes of operand addressibility are as given belo
S - is used to indicate value of operand in storage.
R - is used to indicate value of operand in register.
IS - Indicates that the address of operand is stored i storage.
IR - indicates that the address of operand is stored i register.

2) A Register descriptor -
A Register descriptor is used to keep track of what currently in each register. The register descriptors sho the initially all the registers are empty. As the code generation for block progresses the registers will hol the values of computations.

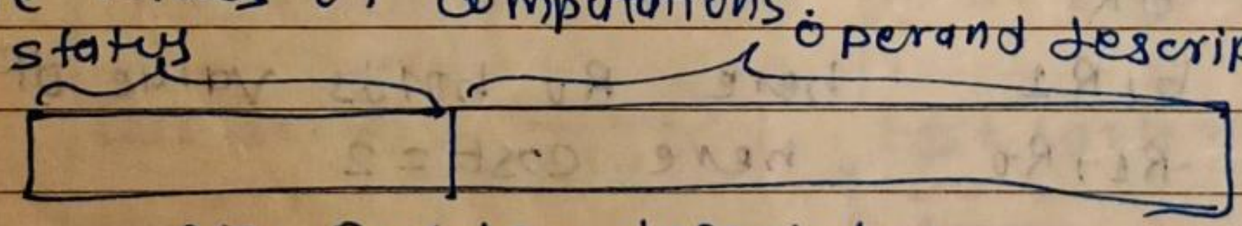


fig. Register descriptor.

- The status field is of Boolean type which is used to check whether the register is occupied with som data or not. When the status field holds the valu 'True' then operand descriptors with some data not. When the status field holds value 'True' then ope descriptor field contains the pointer to the operand descriptor who is having the latest value in the register.

* Algorithm for code generation -

Read the expression in the form of operator, operand1, operand2 & generate code using following algorithm.

```
Gen_Code (operator, Operand1, Operand2)
{
  if (operand1.addressmode = 'R')
  {
    if (operator = '+')
      Generate ('ADD operand2, R0');
    else if (operator = '-')
      Generate ('SUB operand2, R0');
    else if (operator = '*')
      Generate ('MUL operand2, R0');
    else if (operator = '/')
      Generate ('DIV operand2, R0');
  }
  else if (operand2.addressmode = 'R')
  {
    if (operator = '+')
      Generate ('ADD operand1, R0');
    else if (operator = '-')
      Generate ('SUB operand1, R0');
    else if (operator = '*')
      Generate ('MUL operand1, R0');
    else if (operator = '/')
      Generate ('DIV operand1, R0');
  }
  else
  { Generate ('MOV operand2, R0');
    if (operator = '+')
      Generate ('ADD operand2, R0');
    else if (operator = '-')
      Generate ('SUB operand2, R0');
    else if (operator = '*')
      Generate ('MUL operand2, R0');
    else if (operator = '/')
      Generate ('DIV operand2, R0');
  }
}
```

1) Generate the code sequence using code generation alg. for the following ~~gram~~ expression.

$x := (a+b) * (c-d) + ((e/f) * (g+b))$

→ The corresponding three address code can be give

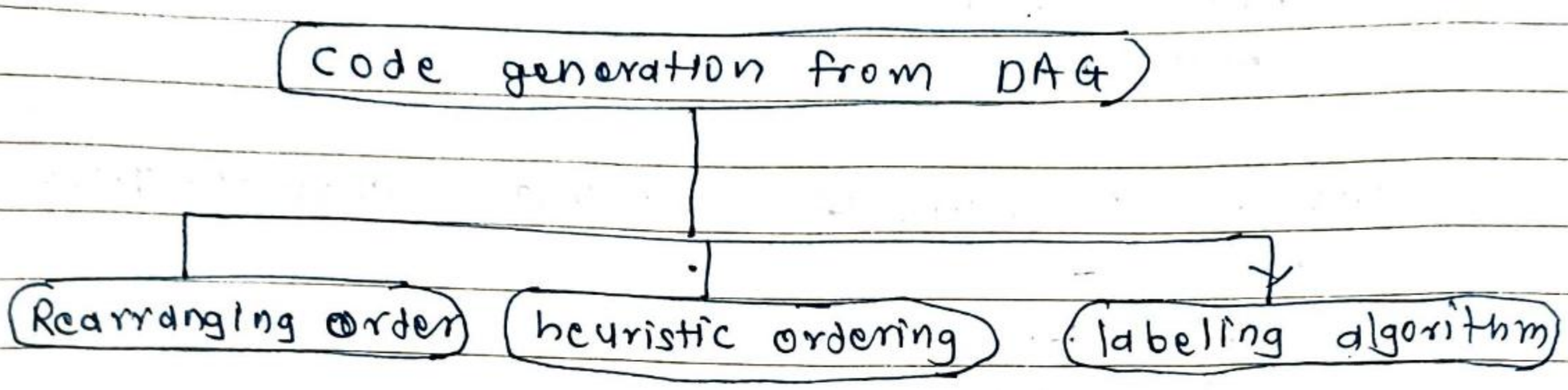
$t1 := a + b$
$t2 := c - d$
$t3 := e/f$
$t4 := t1 * t2$
$t5 := t3 * t1$
$t6 = t4 + t5$

using simple code generator algorithm the sequence targ code can be generated as,

Three address code	Target code sequence	Register Descriptor	Operand descript
$t1 = a + b$	MOV a, R0	Empty	t1 | R | R0
	Add b, R0	R0 contains t1	
$t2 = c - d$	MOV c, R1	R1 contains c	t2 | R | R1
	SUB d, R1	R1 contains t2	
$t3 = e/f$	MOV e, R2	R2 contains e	t3 | R | R2
	DIV f, R2	R2 contains t3	
$t4 = t1 * t2$	MUL R0, R1	R0 contains t1	t4 | R | R1
		R1 contains t2	
		R1 contains t4	
$t5 = t3 * t1$	MUL R2, R0	R2 contains t3	t5 | R | R0
		R0 contains t1	
		R1 contains t5	
$t6 = t4 + t5$	ADD R1, R0	R1 contains t4	t6 | R | R0
		R0 contains t5	
		R0 contains t6	

* Generating Code from DAG -

- Generating code from DAG is much simpler than the linear sequence of three address code.
- Using DAG we can rearrange some sequence of instructions & generate an efficient code.
- There are various algorithms used generating code from DAG.



1) Rearranging order -

- The order of three address code affects the cost of the object being generated.
- In the sense that by changing the order in which computations are done. we can obtain the object code with minimum cost.

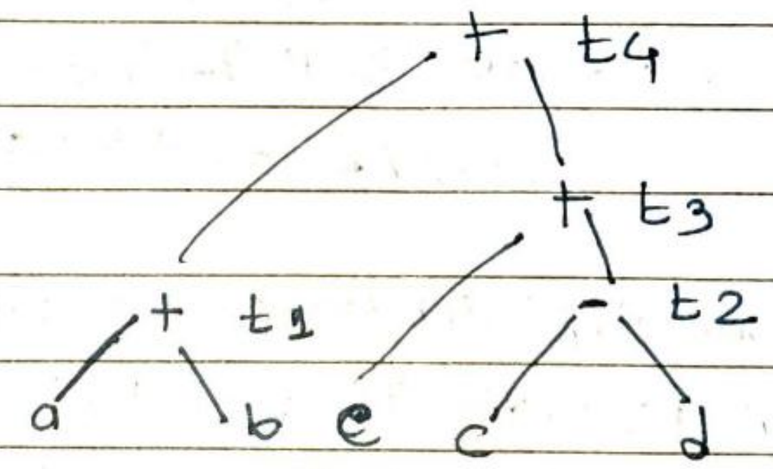
ex. $t_1 = a + b$

$t_2 = c - d$

$t_3 = e + t_2$

$t_4 = t_1 + t_3$

for the expression $(a+b)+(e+(c-d))$ a DAG can be constructed for the above sequence as follows.



The code can be generated by translating the three address code line by line.