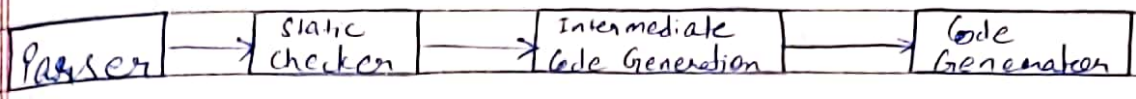
① Intermediate code is used to translate the source code into machine code. It lies between the high-level language and machine language.



- High Level → It can be represented as source code. To enhance performance of source code, we can easily apply code modification But to optimize the target machine, it is less preffered.
- Low Level → It is close to the target machine, which makes it suitable for register and memory allocation etc. It is used for machine dependent optimizations.

② A program is a collection of statements, the ordering and scheduling of which depends on dependence constraints.

- Data Dependencies → when statements compute data that are used by other st.
- Control Dependencies → are those which arise from the ordered flow of control in a program.

A dependence graph can be constructed by drawing edges connect dependent operations. These are impose a partial ordering among operations that prohibit a fully concurrent execution of program.

Importance & The basic idea behind is for compiler to look for various kinds if dependence among statements to prevent their execution in wrong order i.e the order that changes the meaning of the program. This helps it to identify various parallelisable componets in the program.

③ =	Synthesized	Inherited
i)	It is called, if its parse tree node value is determined by attribute value at child nodes.	i) It is called, if its parse tree node value is determined by the attribute value at parent or sibling node.
ii)	The production must have non-terminal at its head.	ii) The production must have non-terminal as a symbol in its body.
iii)	It can be evaluated during a single bottom-up traversal of parse tree.	iii) It can be evaluated during a single top down and sideways traversal of parse tree.
iv)	It can be contained by both the terminals or non-terminals.	iv) It cant be contained by both it is only contained by non-terminals
v)	Example → E.val → F.val E val F val	vi) Example → E.val = F.val E val F val

④ Optimization :- We can optimize the blocks with the help of given types -

i) Common Sub-Expression Elimination :- In this, we dont need to find it over and over. Instead we can find it once and kept in store from where it referenced when encountered again. For example

a = x+y
b = a-d
c = b+c

In above example we are computing x+y two times. Instead we can directly assign a to c

a = x+y
b = a-d
c = a

ii) Dead-Code Elimination :- Sometimes it happens that a program contains dead code. It is the block of the code which are not participating in the execution. Suppose we have written z = 4*d but later on we are not using that part of code. So in order to optimize our code we can simply remove that part of code.

iii) Sometimes it happens that we use a temporary variable to assign a simple multiplication and then we will use that variable for assignment. Instead we can directly assign the multiplication to the main variable. For example.

temp = 5 * z
a = b + temp
Instead of that we can use
a = b + (5 * z). It will help to optimize the code.

iv) Sometimes we are using the loops more than one time in order to perform same action. Then instead of using more loops we can reduced to one and perform the same action. For example

```
for (i=0; i<5; i++)
    a = z * i
for (i=0; i<5; i++)
    b = z * i + 5
```

In order to optimize we can use one loop instead of two loops

```
for (i=0; i<5; i++)
    a = z * i
    b = z * i + 5
```

v) Some times we are computing same statement n number of times. So in order to optimize we can use that one time.

For example

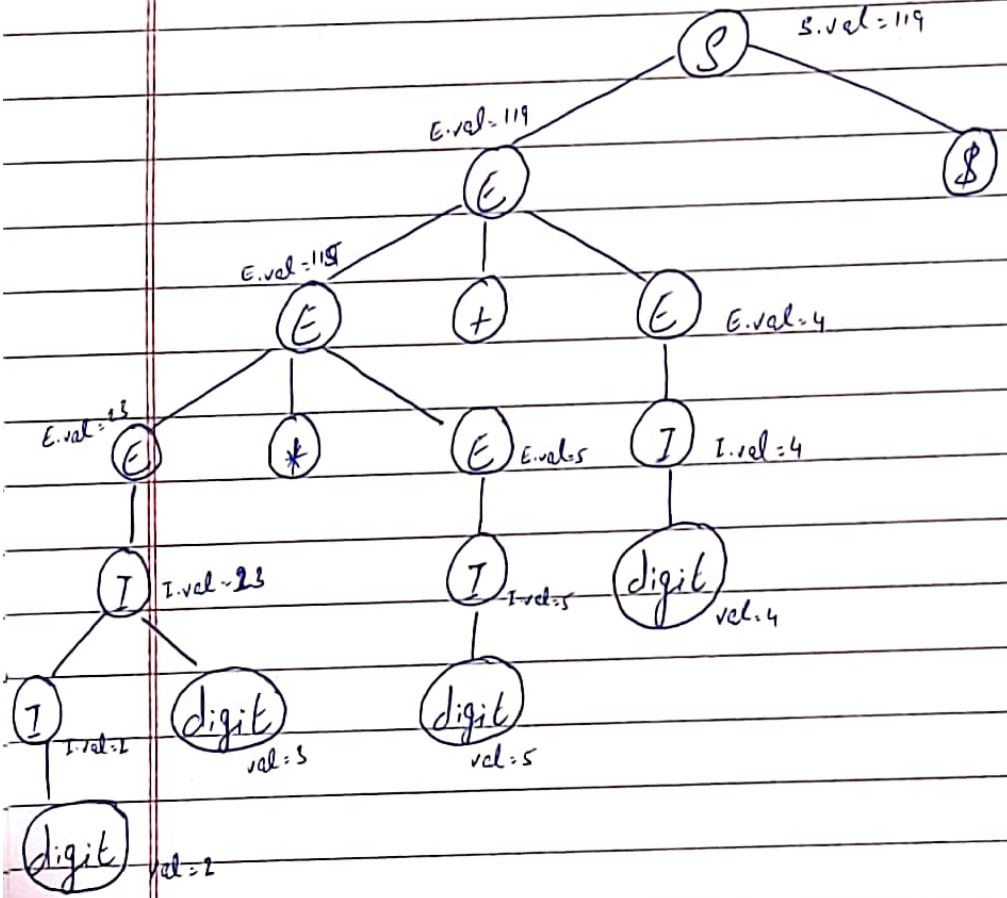
```
for (i=0; i<50; i++)
    z = 2 * c
    if (z > i)
        d = i + 5
```

Use this

```
z = 2 * c
for (i=0; i<50; i++)
    if (z > i)
        d = i + 5
```

(5) The syntax directed translation scheme is a context-free grammer. Syntax direct translation is implemented by construction a parse tree and performing the actions in left to right depth first order.

$S \rightarrow E\$$ { print E.val }
$E \rightarrow E + E$ { E.val = E.val + E.val }
$\quad / E * E$ { E.val = E.val * E.val }
$\quad / (E)$ { E.val = E.val }
$\quad / I$ { E.val = I.val }
$I \rightarrow I$ digit { I.val = 10 * I.val + lexval }
$\quad /$ digit { I.val = lexval }



⑥ Evaluate S-attributed grammer in bottom-up parsing →

- Evaluate it in any bottom-up order of the nodes in the parse tree.
- Apply postorder to the root of the parser tree:

```
void postorder(N) {
    for(each child C of N)
    do
        postorder(C);
    done
    evaluate attributes associated with N;
}
```

- Post order traversal of the parse tree corresponds to the exact order in which the bottom-up parsing builds the parse tree.
- Thus, we can evaluate S-attributed in one bottom-up (LR) pass.

⑦ i) Without using global data to create side effects, some of the semantic actions cannot be performed.

ii) Need to use a symbol table, global data to show side effects of semantic actions.

iii) A program with too many global variables is difficult to understand and maintain.

iv) Restrict the usage of global variables to essential items and use them as object.

(8) i) S-attribute :- • If an SDT uses only synthesized attributes, called S-attribute.
- These are evaluated as bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
- Semantic actions are placed in rightmost place of RHS

ii) L-attribute :- • If an SDT uses both synthesized attribute and inherited with a restriction that inherited attribute can inherit values from left sibling only, it is called as L-attributed.
- Attributes in L-attributed SDT are evaluated by depth first and left to right parsing.
- Semantic actions are placed anywhere in the R.H.S.

(9) Directed Acyclic Graph is a graph that is directed and without cycles connecting the other edges. That means, it is impossible to traverse the entire graph starting at one edge.

Process Of Creation :-

Rule 1 → Interior nodes always represent operator.
Exterior nodes always represent the names, identifiers or constants.

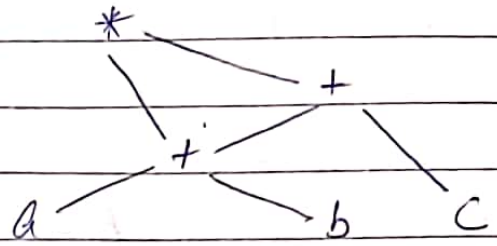
Rule 2 → A check is made to find if there exists any node with the same value.
- A new node is created only when there does

not exist any node with the same value.
- This action helps in detecting the common sub-expression and avoiding re-computation of same.

Rule 3 → The assignment instruction of form x := y are not performed unless they are necessary.

Example : (a+b) x (a+b+c)

T1 = a+b
T2 = T1 + c
T3 = T1 + T2



(10) Register Allocation And Assignment :- i) Register allocation is only within a block. It follows top-down approach.

ii) Assign registers to the most heavy use variable.

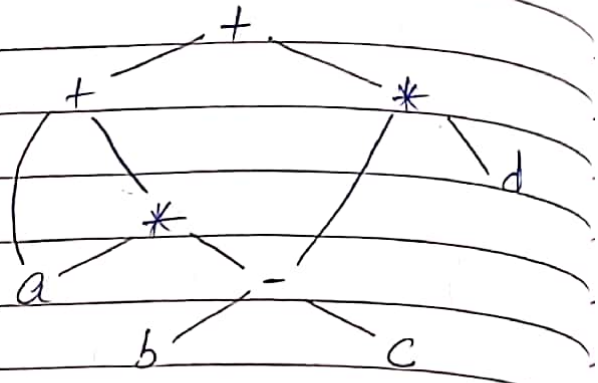
- Traverse the block
- Count uses
- Use count as a priority function.
- Assign registers to higher priority variables first.

Advantages- Heavily used values reside in registers.

Disadvantage :- Does not consider non-uniform distribution of uses.

(11) Expression → a + a * (b - c) + (b - c) * d

t1 = b - c
t2 = a * t1
t3 = a + t2
t4 = d * t1
t5 = t3 + t4



(12) Dead Code Elimination & It is possible that program contains dead code. Suppose the statement x := y + z appears in a block and x is dead symbol that means it will never used. Then without changing the value of block you can safely remove this statement.

(13) Peephole optimization is performed on the very small set of instructions in a segment of this code. It is basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without change in output. Peephole optimization is machine dependent.

Objectives & • To improve performance.
• To reduce memory footprint.
• To reduce code size.

(14) The problem in generating three address codes in a single pass is that we may not know the labels that control must go to.

at the time jump statements are generated. So to get around this problem a series of branching statements with targets of the jumps temporarily left unspecified is generated.

Back Patching is putting the address instead of labels when the proper label is determined. It performs three types of operation:-

i) Makelist(i) - Create new list containing i.
ii) Merge(i,j) - Concatenate i & j
iii) Backpatch(p,i) - Insert i as target for each of the statements on the list pointed to by p.

(8) ⇒ Basic Block :- It is a set of statements. The basic blocks dont have any in and out branches except entry and exit. It means the flow of control enters at beginning and will leave at the end without any halt. The set of instructions of basic block execute in sequence.

⇒ Flow Graph :- It is a directed graph. After partitioning an intermediate code into basic blocks, the flow of control among basic blocks is represented by a flow graph. An edge can flow from one block X to another block Y in such a case when Y blocks first instruction immediatly follows the X blocks last instruction.