Unit - 4

Intermediate Code Generation

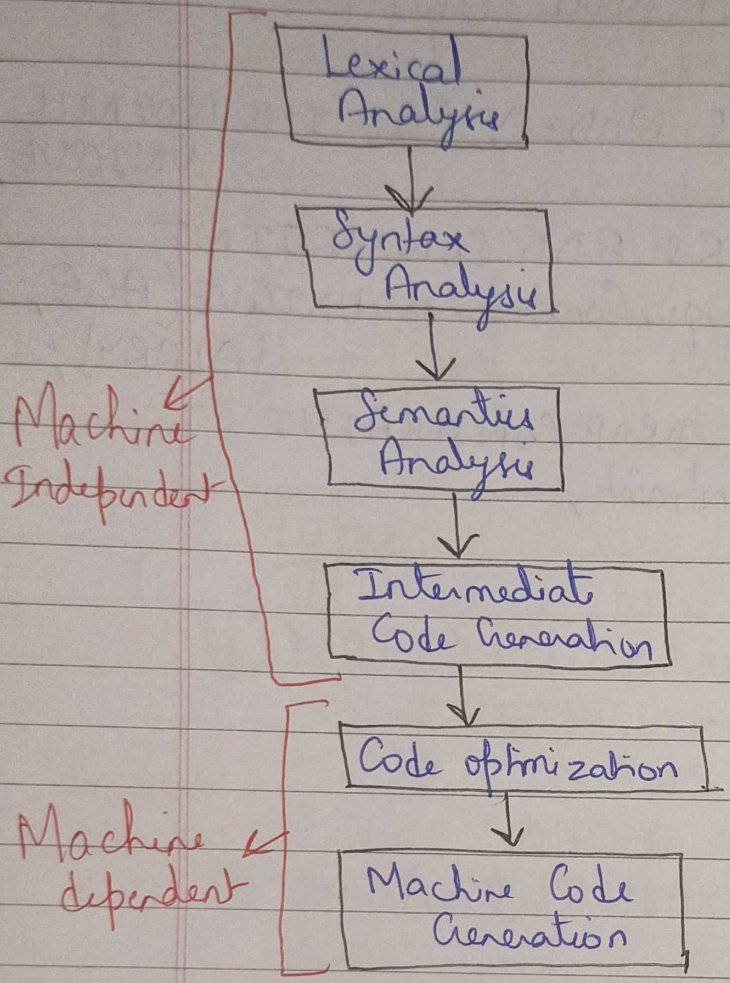


fig. Position of Intermediat Code Generation

Various forms :-

1. Linear form
   a. Prefix
   b. Postfix
   c. 3 address Code
2. Tree form
   a. Syntax Tree
   b. DAG

a. Prefix Notation :-

$a+b \Rightarrow +ab$

$(a+b)*c \Rightarrow *(+ab)c$

$(a+b)*(c-d) \Rightarrow *(+ab)(-cd)$

b. Postfix Notation :-

$a+b \Rightarrow ab+$

$(a+b)*c \Rightarrow (ab+)c*$

$(a+b)*(c-d) \Rightarrow (ab+)(cd-)*$

c. 3 address Code :-

1. x op z → (variable operator variable)
2. op z → (operator variable)

Ex → $u = a+b*c+d$

1. $T1 = b*c$
   $\Rightarrow a+T1+d$
2. $T2 = a+T1$
   $\Rightarrow T2+d$
3. $T3 = T2+d$
4. $u = T3$

Example 2 :- $a=(b*-c)+(b*-c)$

1. $T1 = -c$
2. $T2 = b*T1$
3. $T3 = -c$
4. $T4 = b*T3$
5. $T5 = T2+T4$
6. $a = T5$

Ans

Implementation of 3-address Code :-

A 3-address Code is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representation are :-

1) Quadruples
2) Triples
3) Indirect Triples

Quadruples :-

→ A quadruples is a record structure with four fields which are op, arg1, arg2 and result.

→ The op field contains an internal code for the operator. The 3-address statement x := y op z is represented by placing y in arg1, z in arg2 and x in result.

Triples :-

→ 3 fields op, arg1, arg2.

→ to avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of statement that comput it.

Indirect triples :-

Another implementation of 3-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.

So, lets see with the example. We take example of previous ques.

Quadruples :-

#	op	arg1	arg2	result
(1)	−	c		T1
(2)	*	b	T1	T2
(3)	−	c		T3
(4)	*	b	T3	T4
(5)	+	T2	T4	T5
(6)	—/=	T5		a

Triples :-

#	op	arg1	arg2
(1)	−	c	
(2)	*	b	(1)
(3)	−	c	
(4)	*	b	(1)
(5)	+	(2)	(4)
(6)	=	a	(5)

Indirect Triples :-

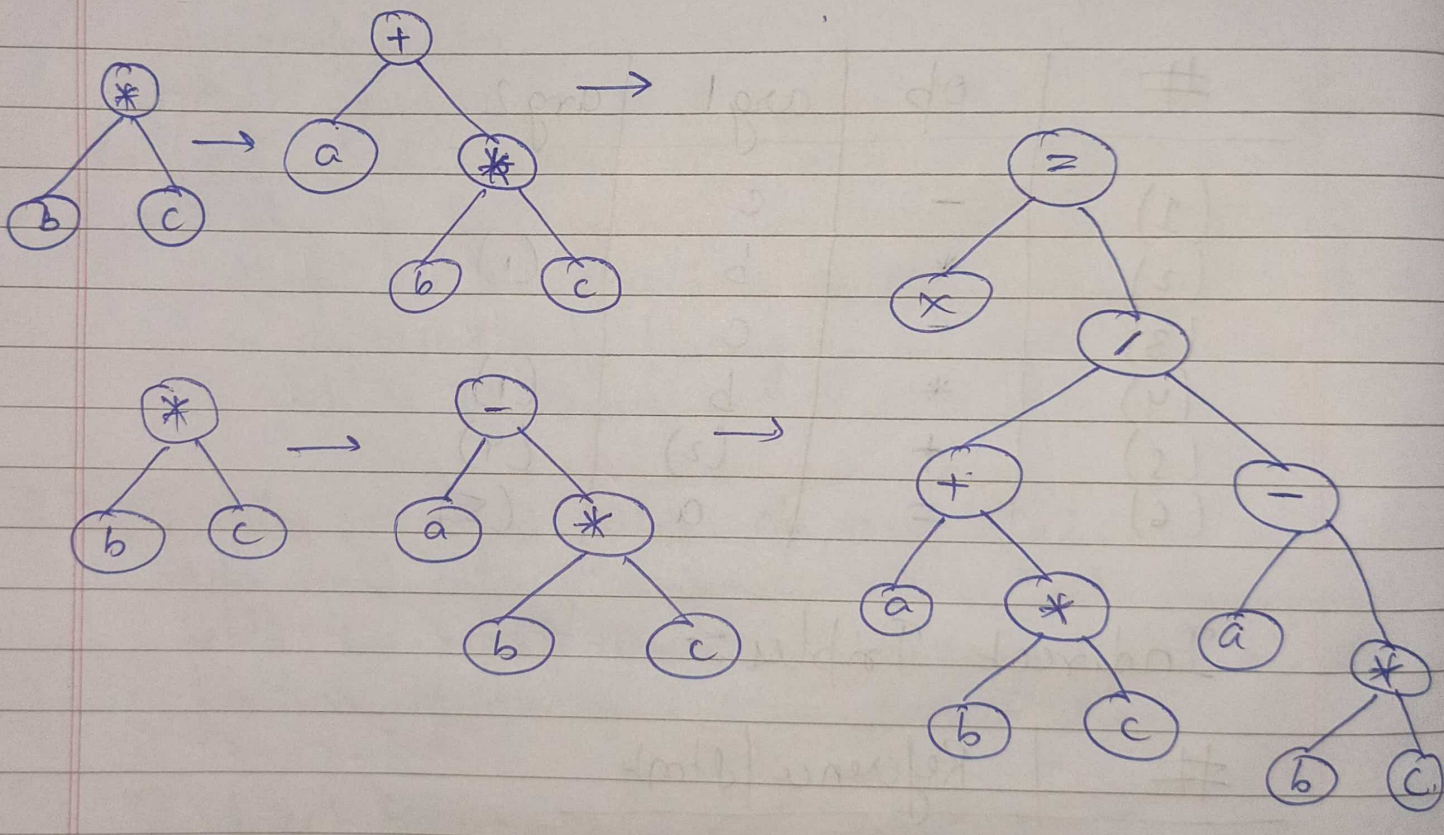
#	Reference/stmt
(1)	(13)
(2)	(14)
(3)	(15)
(4)	(16)
(5)	(17)
(6)	(18)

Question to practise :-

1) $-(a*b) + (c*d + e)$.
2) $(a+(b-c)) + (c-(d+e))$.

Syntax Tree :- A syntax tree depicts the natural hierarchical structure of a source program.

Q. $n = (a+b*c)/(a-b*c)$



Q. Basic Block & Control flow Graph :-

Partition Algorithm :-

1. First statement of a 3 addres Code is a leader.
2. Target instruction of conditional/unconditional branch is leader
3. Next Stmt of conditional/unconditional branch is leader.

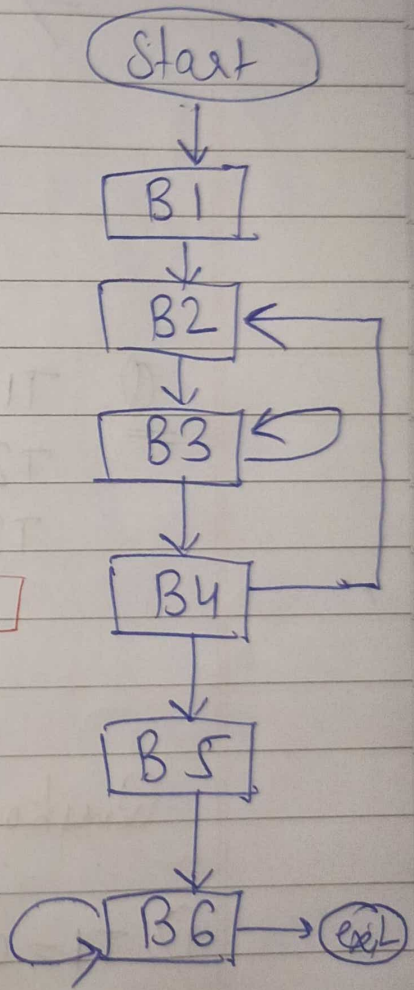
leader ← t1 = a * a
t2 = a * b
] B1

leader ← t3 = 2 + 1
t4 = t1 + t3
t5 = 2 * t2
t6 = t3 + t5
if i - - goto statement 3
] B2

leader ← t7 = a .
] B3 .

Q. L ← 1. i = 1] B1
L ← 2. j = 1] B2
L ← 3. t1 = 10 * i
4. t2 = t1 + j
5. t3 = 8 * t2
6. t4 = t3 - 88
7. a[t4] = 1.0
8. j = j + 1
9. If j <= 10 go to Stmt 3] B3
L ← 10. i = i + 1] B4
11. if i <= 10 goto Stmt 2
L ← 12. i = 1] B5
L ← 13. t5 = i - 1
14. t6 = t5 * 88
15. a[t6] = 1.0] B6 .
16. i = i + 1
17. if i <= 10 go to Stmt 13 .

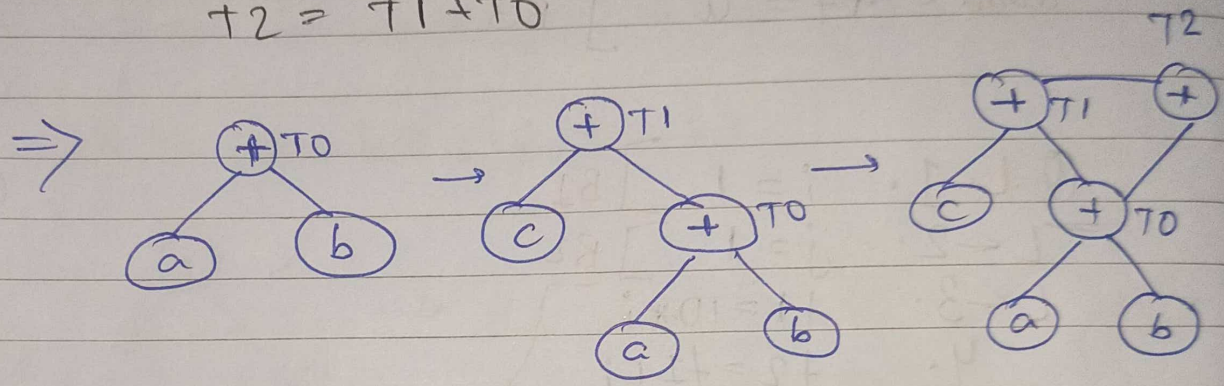
⇒



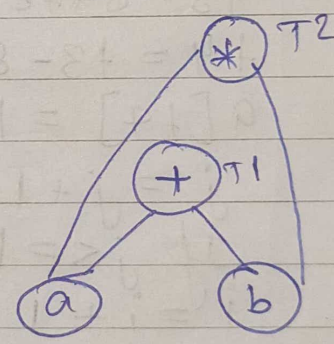
Directed Acyclic Graph :-

A DAG gives the same information as syntax tree but in a more compact way because common subexpressions are identified.

Q. T0 = a+b
T1 = c+T0
T2 = T1+T0



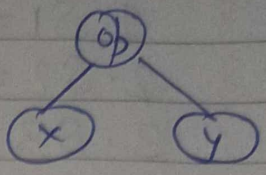
Q. T1 = a+b
T2 = a*b
T3 = a*b

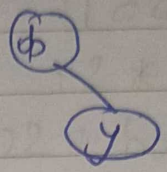


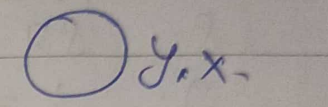
Question to practise :-

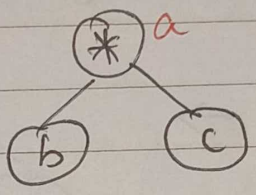
1) T1 = a-b
T2 = a+b
T3 = T1+T2
T4 = 4*T1
T5 = 4*T2
T6 = 4*T2.

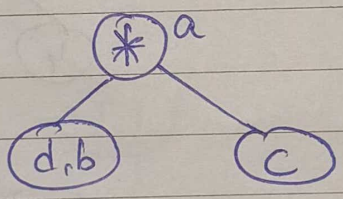
2) S1 = 4*i
S2 = a[S1]
S3 = 4*i
S4 = b[S3]
S5 = S2*S4
S6 = prod * S5

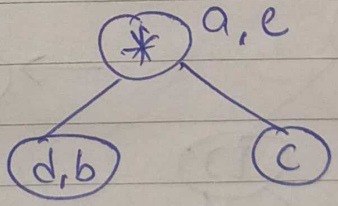
Rules :- 1) $a = x \text{ op } y \Rightarrow$ 

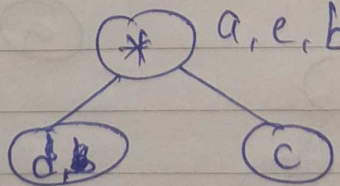
2) $b = \text{op } y \Rightarrow$ 

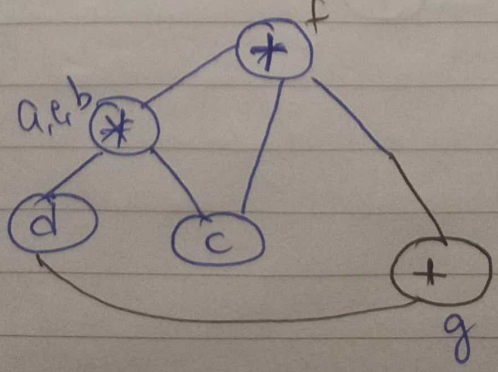
3) $u = y$
(i) x is a operand
(ii) y is a label $\Rightarrow$ 

1. $a = b * c$ 

2. $d = b$ 

3. $e = d * c$ 

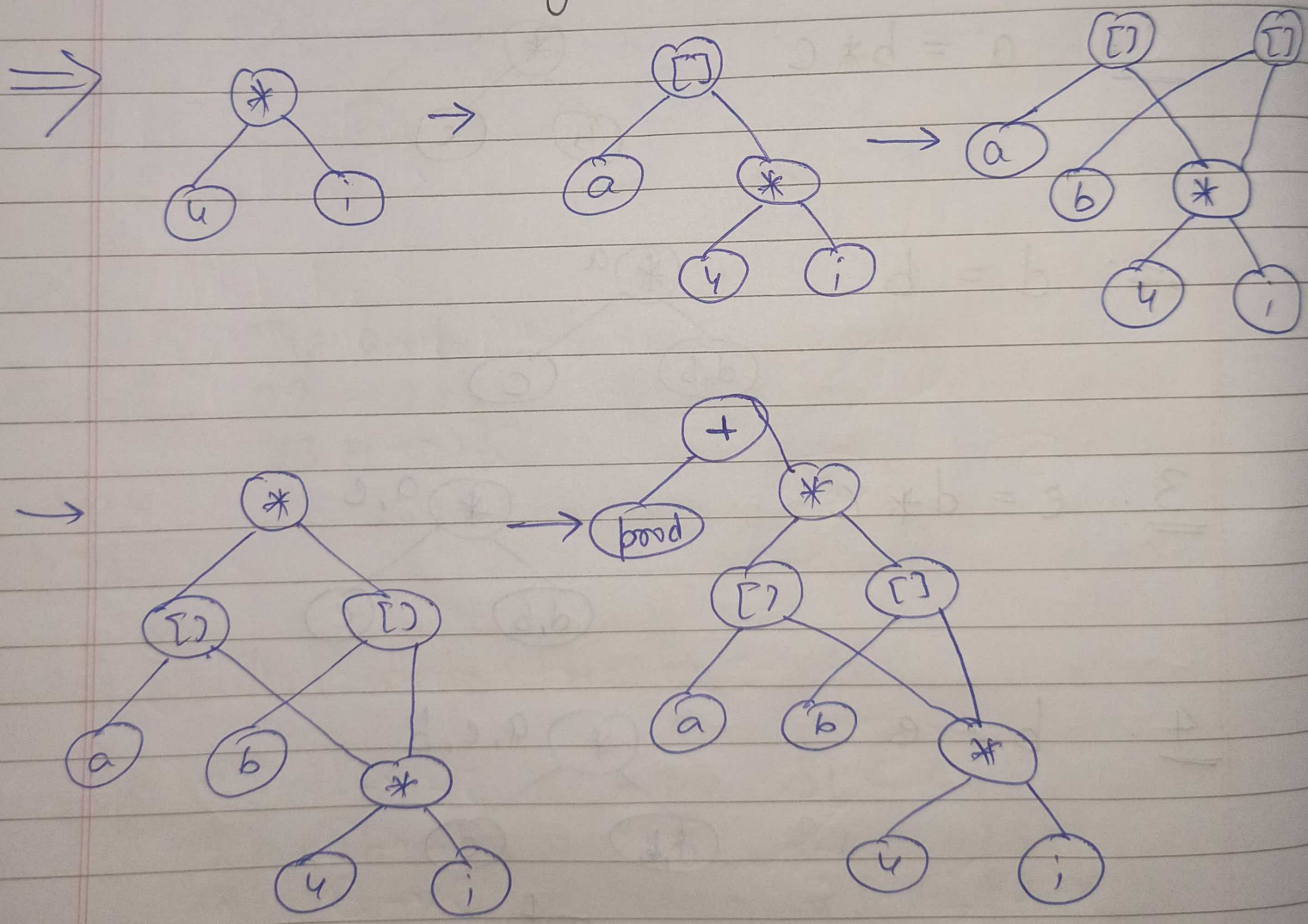
4. $b = e$ 

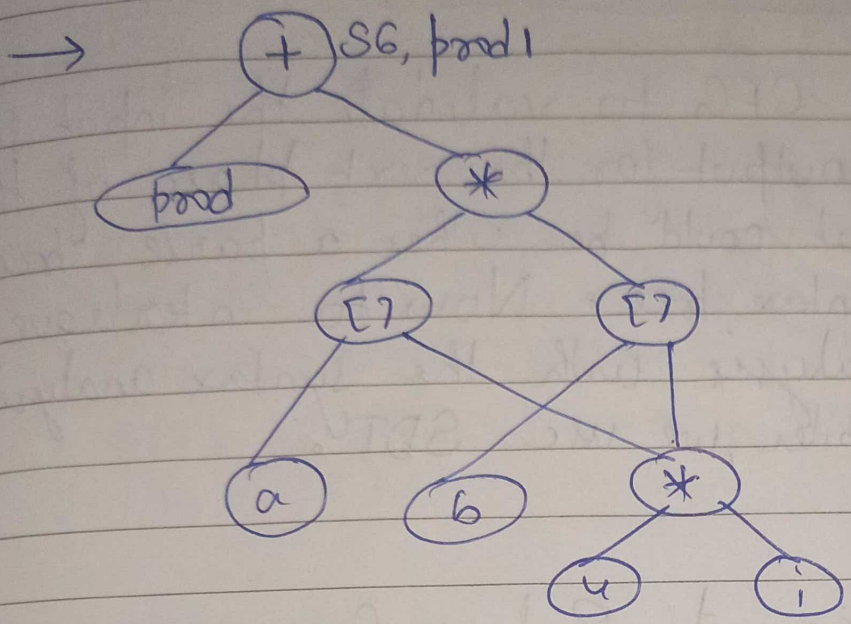
5. $f = b + c$ 

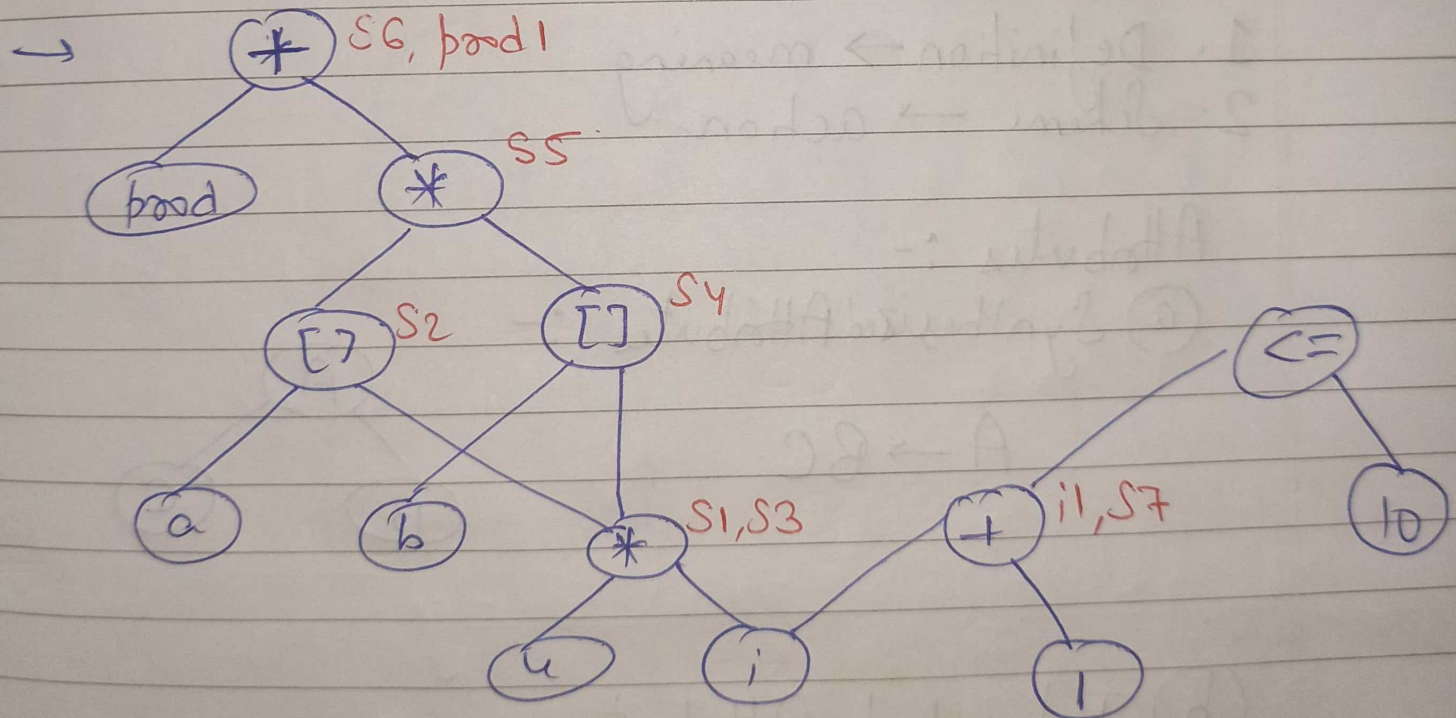
6. $g = d + f + d$

Q.

1. S1 = 4*i
2. S2 = a[S1]
3. S3 = 4*i
4. S4 = b[S3]
5. S5 = S2 * S4
6. S6 = prod + S5
7. prod = S6
8. S7 = i+1
9. i = S7
10. if i<=10 go to step 1




+
S6, prod1
prod
*
[]
[]
a
b
*
c
i


*
S6, prod1
prod
*
S5
[]
S2
[]
S4
a
b
*
S1,S3
c
i
<=
+
i1,S7
10
1

Syntax Directed Translation

Parser use a CFG to validate the input string and produce output for the next phase of the compiler. Output could be either a parse tree or an abstract syntax tree. Now to interleave semantic analysis with the syntax analysis phase of compiler, we use SDT.

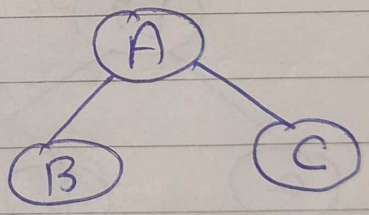
SDT = Semantic Rule + Grammar

1. Definition → meaning
2. Scheme → action

Attributes :-

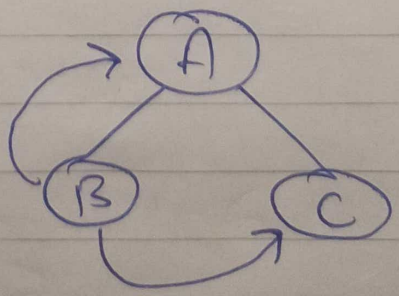
(a) Synthesized Attributes :-

$A \rightarrow BC$



(b) Inherited Attributes :-

$A \rightarrow BC$
$B \rightarrow A$
$B \rightarrow C$

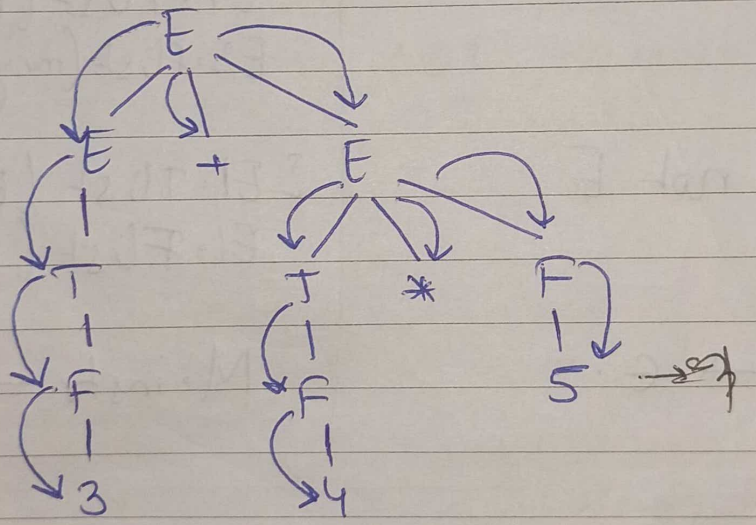


SDT Definition

Production Rule	Meaning	action
$E \rightarrow E+T$	E.value = E.value + T.value	print '+'
$E \rightarrow T$	E.value = T.value	{ }
$T \rightarrow T*f$	T.value = T.value + f.value	print '*'
$T \rightarrow F$	T.value = f.value	{ }
$F \rightarrow num$	F.value = num.lexvalue	print 'num'

SDT Scheme

Q. 3+4*5



⇒ print(3) → print 4 → print 5 → print * → print +

⇒ 345*+

Q. 3*4*5

Backpatching :-

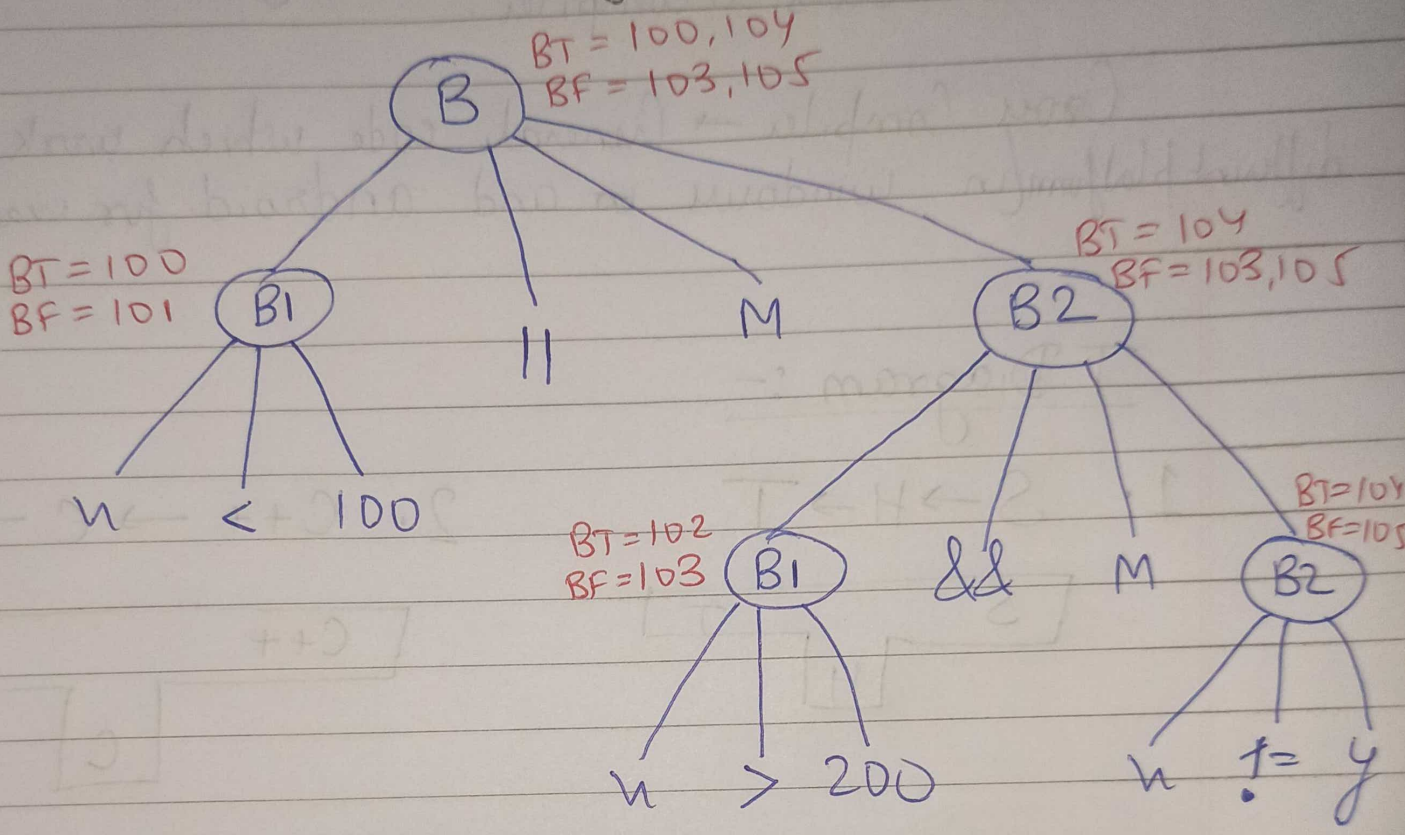
3 function :-
Backpatch ()
merge ()
makelist ()

Production Rule	Semantic Action
$E \rightarrow E_1$ or $M E_2$	{ backpatch(E1.falselist, M.instr) E1.Tlist(merge(E1.Tlist, E2.Tlist)) E2.Flist(E2.Flist) }
$E \rightarrow E_1$ and $M E_2$	{ backpatch(E1.tlist, M.instr) E1.Tlist(E1.Tlist) E2.flist(merge(E1.Flist, E2.Flist)) }
$E \rightarrow$ not E	{ E1.Tlist(E1.flist) E1.Flist(E1.Tlist) }
$E \rightarrow \epsilon$	M.instr → next instruction

Q. $x<100 || x>200 \&\& x!=y$.

⇒
100: if x<100 go to 106
101: else goto 102
102: if x>200 goto 104
103: else goto 107
104: if x!=y then goto 106
105: else goto 107
106: True statement
107: False statement

Backpatching



B → B1 || B2 :-

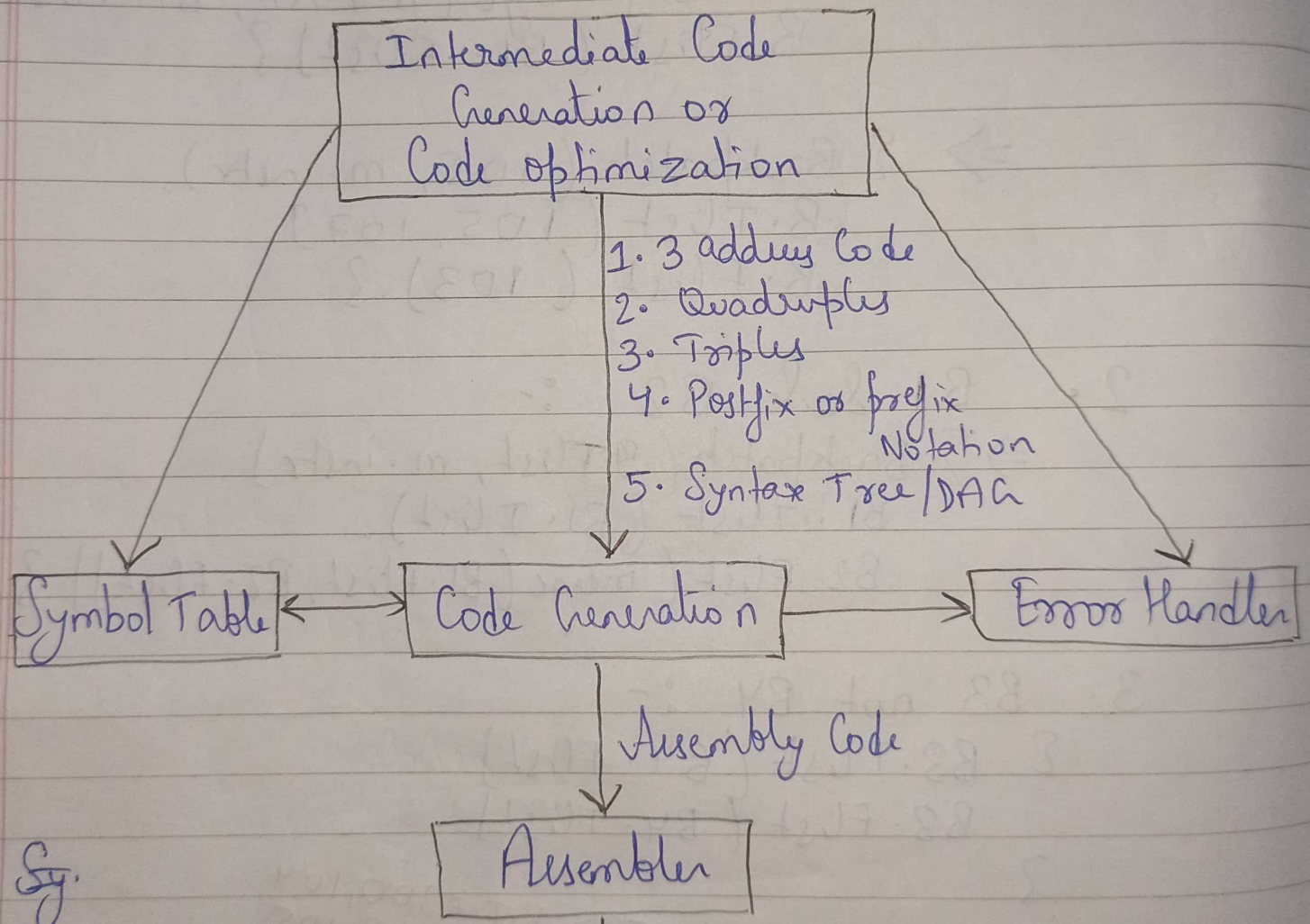
{ Backpatch(B.Flist, m.inst),
B.Tlist (merge (B1.Tlist, B2.Tlist))
B.Flist (B2.flist). }

B2 → B1 && B2 :-

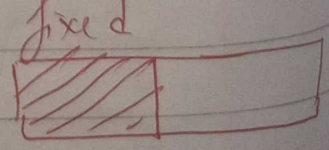
{
Backpatch (B.Tlist, m.inst).
B2.Flist (merge (B1.Flist, B2.Flist)),
B2.Tlist (B1.Tlist)) }

Code Generation

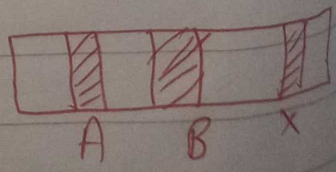
Properties :-
1. High performance
2. Correctness
3. Efficient use of resource of target machine
4. Quick code generation



Sy.



1. Absolute m/c Code
2. Relative m/c Code



Issues in Code Generation :-

1. Input target Problem
2. Code generation
3. Memory Management
4. Instruction Selection
5. Register allocation

Q. $x = y + z$

⇒
```
MOV y, R0
ADD z, R0
MOV R0, x
```

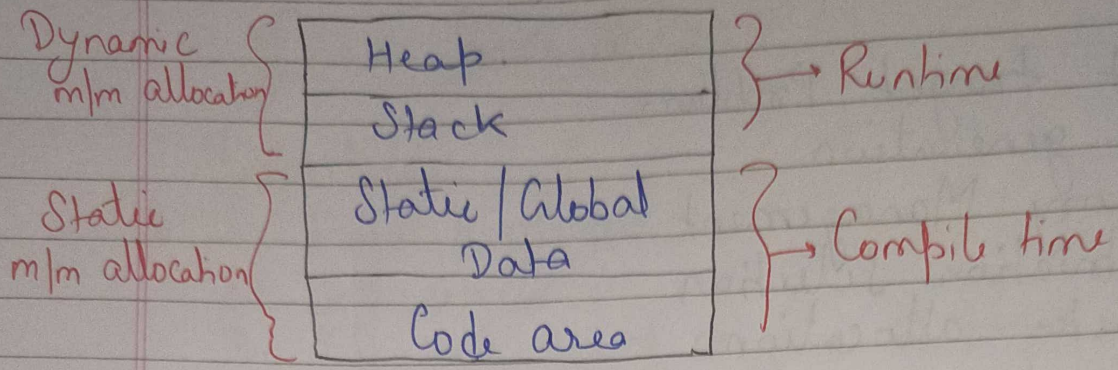
Q. $a = b + c$
$d = a + e$

⇒
```
MOV b, R0
ADD c, R0
MOV R0, a
MOV a, R0
ADD e, R0
MOV R0, d
```

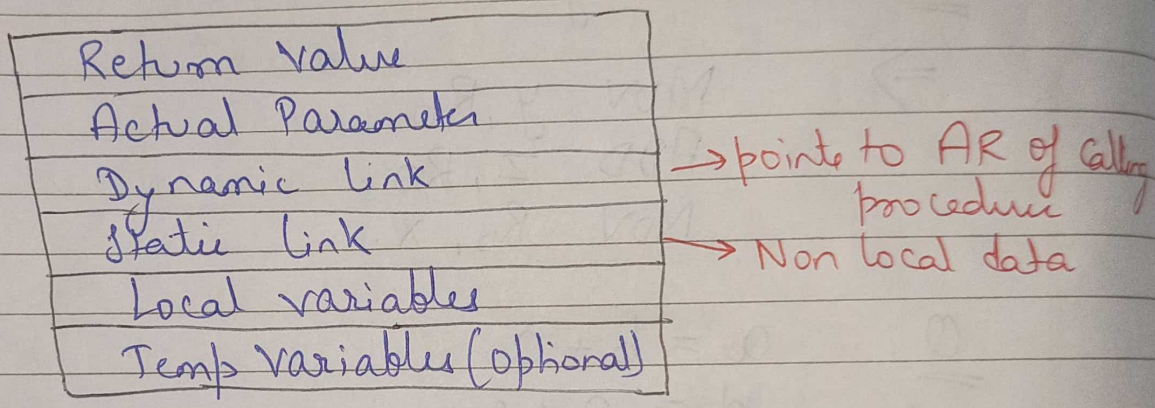
⇒
```
MOV b, R0
ADD c, R0
ADD e, R0
MOV R0, d
```

Q. $a = b * c$
$d = e * f$
$x = a + d$
$y = x + g$

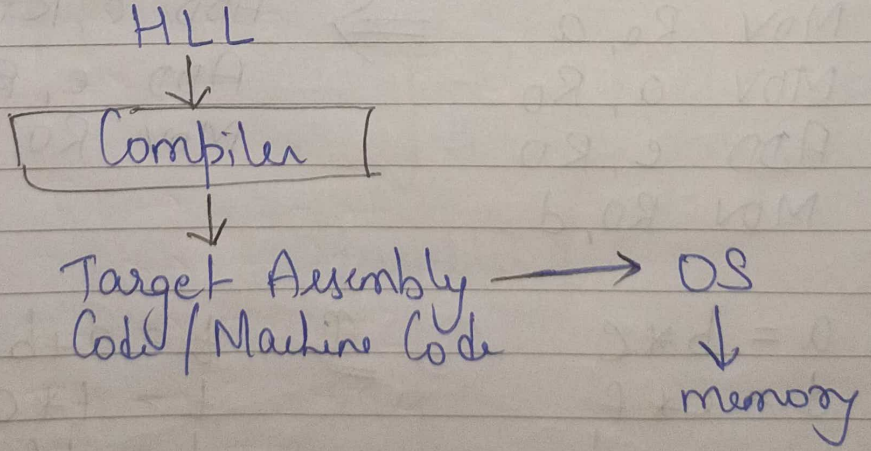
Q. $t = a + b$
$t = t * c$
$t = t / d$



Stack allocation Scheme :-



Runtime Storage Management :-



Activation Record (AR) :-

```
main() {
    a1()
}
```

```
a1() {
    b1()?
b1() {
  - -
}
```

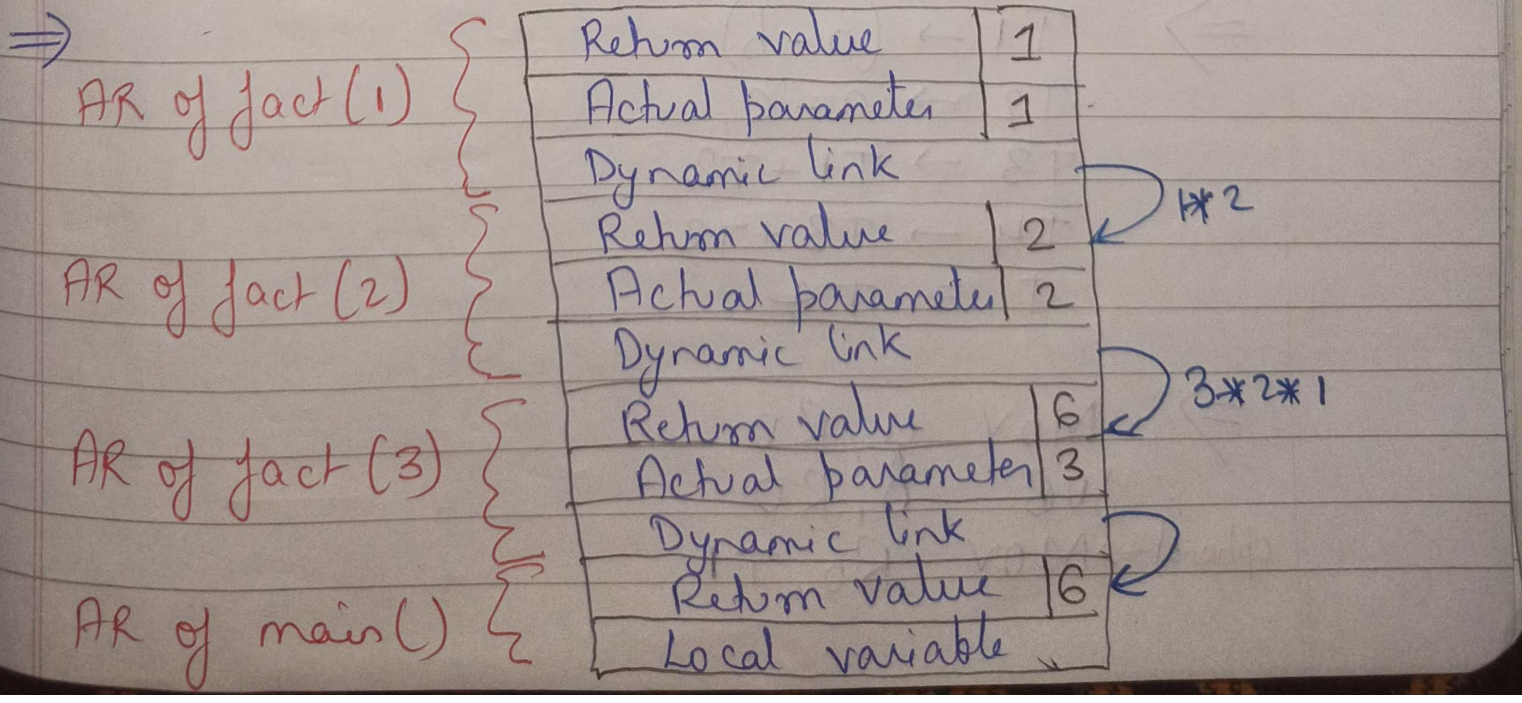
⇒ All segments have different activation Record

AR → main
a1
b1

Q.
```
main() {
    int f;
    f = fact(3);
}
int fact(int n)
    if(n == 1)
        return 1;
    else
        return n*fact(n-1);
```

⇒ main()
fact(3)
fact(2)
fact(1)

⇒



Code Generation Algorithm

1. Invoke function getreg() to store output into register L.
2. If value of y is already in addres descriptor then go ahead otherwise store addres of y in y' and store it in addres descriptor
3. Repeat step 2 for x
4. Update value of L, store output of x op y.
5. If there is no use of x and y in future store the value in register descriptor.

Register Descriptor → Store variables.

x	y
R_0	R_1

Addres Descriptor → stores addres

Q. ~~$T_1 \rightarrow a - b$~~
~~$T_2 \rightarrow a -$~~ $T = (a-b) + (a-c) + (a-c)$

⇒ $T1 \rightarrow (a-b)$
$T2 \rightarrow a-c$
$T3 \rightarrow T1+T2$
$T4 \rightarrow T3+T2$
$T = T4$

⇒ MOV a, R0
SUB b, R0
optional → MOV R0, T1

```
        MOV  a, R1
        SUB  c, R1
optional→ MOV  R1, T2
        ADD  R1, R0
        ADD  R1, R0
optional→ MOV  R0, T4
        MOV  R0, T
```

S.No	Statement	Code Generated	Register Descriptor	Address Descriptor
1.	T1 = a - b	MOV a, R0 SUB b, R0	R0 Contains T1	T1 in R0
2.	T2 = a - c	MOV a, R1 SUB c, R1	R0 Contains T1 R1 Contains T2	T1 in R0 T2 in R1
3.	T3 = T1 + T2	ADD R1, R0	R0 Contains T3 R1 Contains T2	T3 in R0 T2 in R1
4.	T4 = T3 + T2	ADD R1, R0	R0 Contains T4 R1 Contains T2	T4 in R0 T2 in R1
5.	T = T4	MOV R0, T		

Question to Practice :-

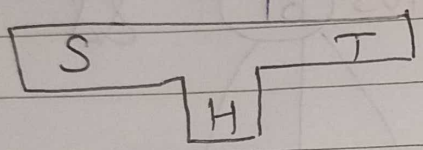
1) n = (a/b) + (c*d) - (c+a) + (f-c)
2) n = (a-b) + (c*d) - (e/f) + (g-a)
3) T1 = A - B
   T2 = C - D
   T3 = E - T1
   T4 = T1 + T2

Bootstrapping :- Process to generate compiler or cross compiler.

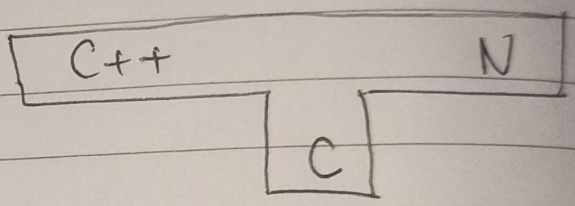
Cross Compiler → Generate code which work both in different platform (in windows and android for example).

T Diagram :-

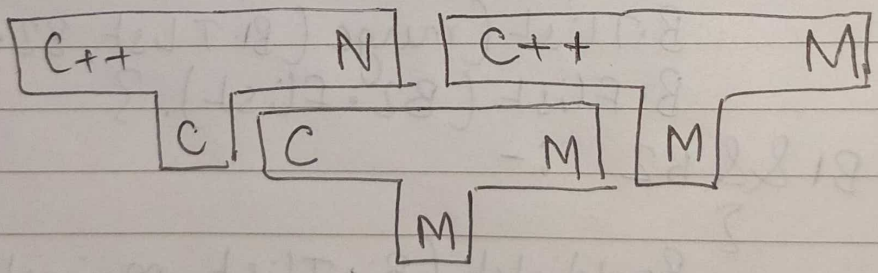
1. S → H → T



2. C++ → C → N

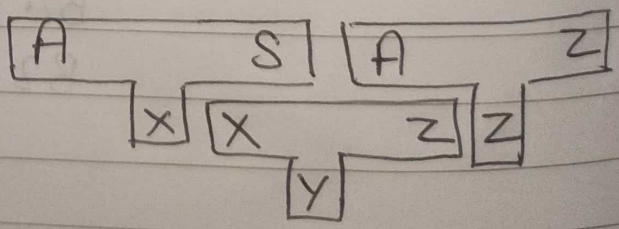


1. C → M → M
   C++ → C → N.



2. X → Y → Z
   A → X → S
   A → ? → Z

⇒



Q. 3. A → B → C
      D → A → X
      D → ? → C. ?