Chapter - 5

26.6.2024
Zinia ma'am

Syntax Directed Translation

Syntax ⟶ construction (Ex : Sub (S) + verb (V) + obj (O). ⟶ syntax)

Semantic ⟶ meaning

I eat rice.

↓ translation ⟶ to understand the meaning

आणि जाऊ शकते

Syntax Analyzer ⟶ Semantic analyzer

(Parse tree) (Parse tree with additional info that has a meaning)

SDD (Syntax Directed Definition)

⟶ CFG with attributes & rules

E ⟶ E + T

⇓

E. type

↓

Example of attribute

X ⟶ symbol

a ⟶ one of X's attribute

⟶ X.a

* attributes may be number, type, table reference, values for instance

Ex : S_n ⟶ SVO ⟶ O.eng
V.eng
S.eng

S_n ⟶ S_n.eng

English sentence

I eat rice

↓ ↓ ↓

$S_n.eng = S.eng \,||\, V.eng \,||\, O.eng$

by concatenating

Scanned with CamScanner

$S_n \longrightarrow S\ V\ O$

S_n.bangla ↓ S.bang, V.bang, O.bang

S_n.bang

আমি ভাত খাই

$S_n.bang = S.bang \,||\, O.bang \,||\, V.bang$

↳ This rule is called semantic rules.

attribute for non-term :

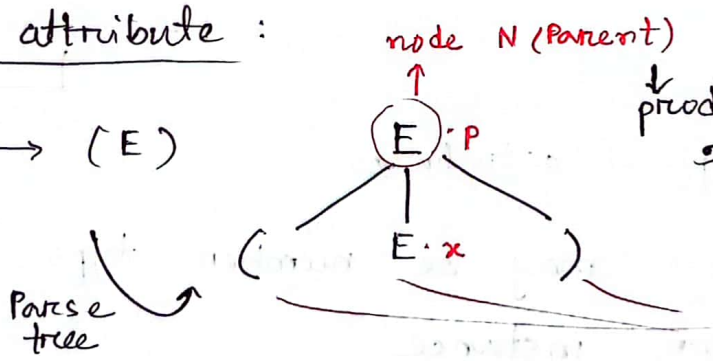
→ synthesized
→ inherited

$E \longrightarrow (\ E\ )$

terminal (( )) non-terminal (E)

Synthesized attribute :

$E \longrightarrow (\ E\ )$

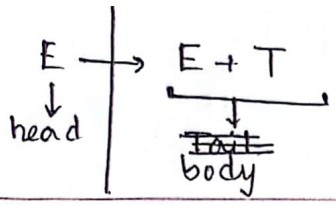


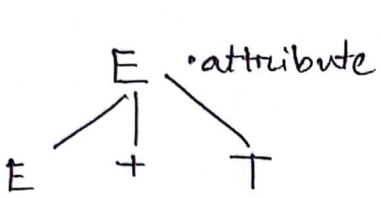
production এর head এ থাকবে

child থেকে যে node N নিজের থেকে যদি attribute value (here val) পায়, then that's synthesized attribute

$E.p = E.x + E.p$

parent এর, child এর, নিজের

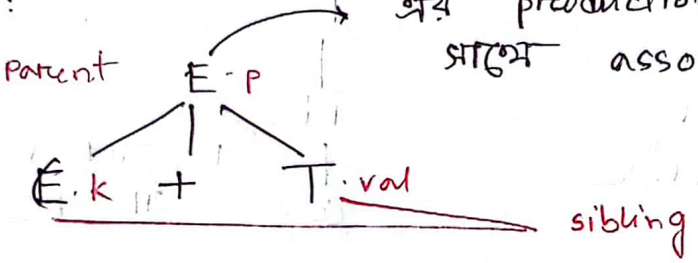




will be synthesized if it's from itself or it's children.

Inherited :

Ex :

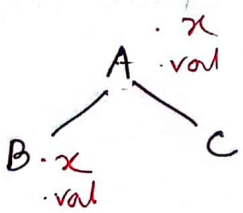


এর production rule যখন তার parent এর সাথে associated.

$$E.k = E.p + T.val$$

Ex :

$A \longrightarrow BC$



semantic rule :

$A.val = A.x$

↓

A, head এ আছে.

নিজের থেকে att. নিচ্ছে,

↳ synthesized attribute

semantic rule :

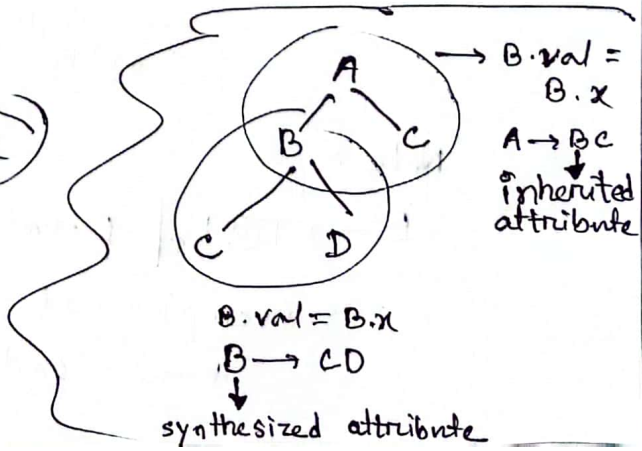
$B.val = B.x$

↓

B, production এর body তে আছে ;

and নিজের থেকে attribute নিচ্ছে ,

↳ inherited attr



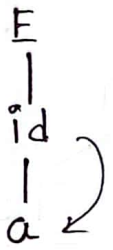
attribute for terminal :

$E \longrightarrow id$

- terminal এর attribute is always synthesized which comes from lexical analyzer. (lexical value)

$E \longrightarrow id$

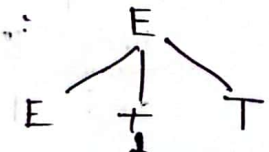
id . lex val
attribute



lex থেকে value না পাই

- There are no semantic rules in SDD itself for computing the value of an attribute for terminal.

Ex :



$E \cdot val = E.val + T.val$

(terminal)

Ex : $E \longrightarrow E@T$ | $E.val = E \cdot val + T.val$

Production rule

Semantic rule

SDD

Assume, This is addition sign

meaning, আমরা semantic rule এ লিখি

Note :

$E \longrightarrow T@E_1 \mid E \cdot val = E_1 \cdot val + T.val$

- Subscript add করার to understand which E is in head and which E is in body.

Ex 5.1 :

Terminal → 6 ✗

Non - " → 4 ✗ (L, E, T, F)

L → E n (n → newline) → L.val = E.val

F → (E) → F.val = E.val

F → digit → F.val = digit.lexval

attribute → val, lexval

L.val → synthesized

E.val →

T.val →

F.val →

all are synthesized attributes → S-attributed SDD

↓

all are in the head of the production rule → synthesized.

Attribute Grammar :

Just attribute value নিয়েই কাজ করছি, calculation করে রেখে দিচ্ছি ⟶ attribute grammar

Side effects ⟶ symbol table এ value update, printing something

⟶ Not a attribute grammar

- Ex 5.1 ⟶ attribute grammar.

Evaluating an SDD at the Nodes of a Parse tree

⟶ প্রতিটা node এর attribute value as additional info in the parse tree

অনুবাদ ⟶ ANNOTATED PARSE TREE

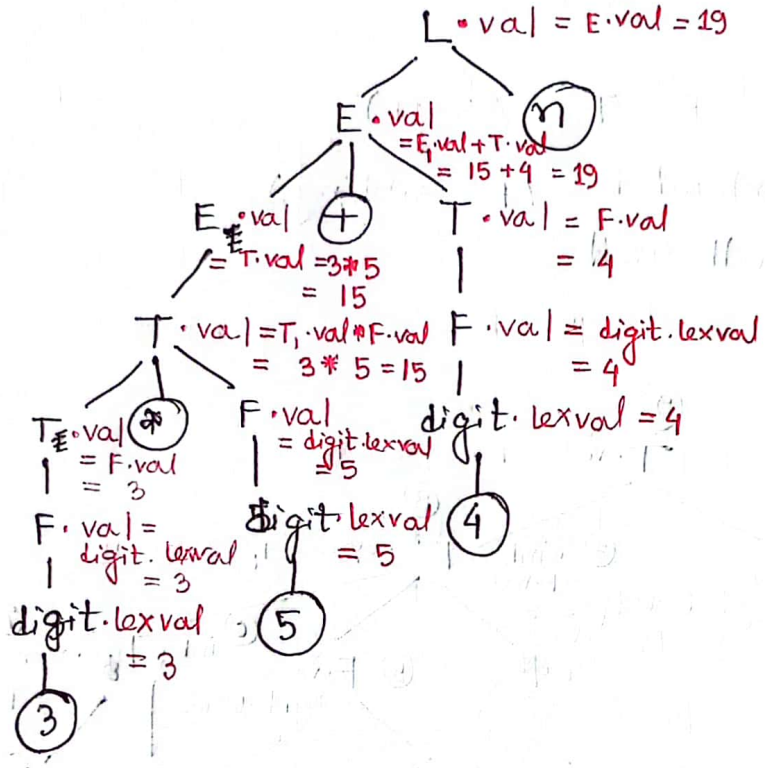
how to evaluate (in which order)?

Ex : 5.1 ⟶ bottom to up evaluate করবো (Post order traversal)

Scanned with CamScanner

Ex : 5.2

Annotated Parse tree for 3*5+4n (From Ex : 5.1)



Circular Dependency :

SDD with inherited & synthesized attribute.

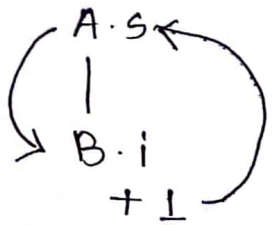
$A \rightarrow B$

$A.S = B.i$ (B is child) — synthesized

$B.i = A.S + 1$ — inherited (A is parent)

In this case,

* we need to redesign the SDD.
* we need to know evaluation order here.



Scanned with CamScanner

Ex : 5.3

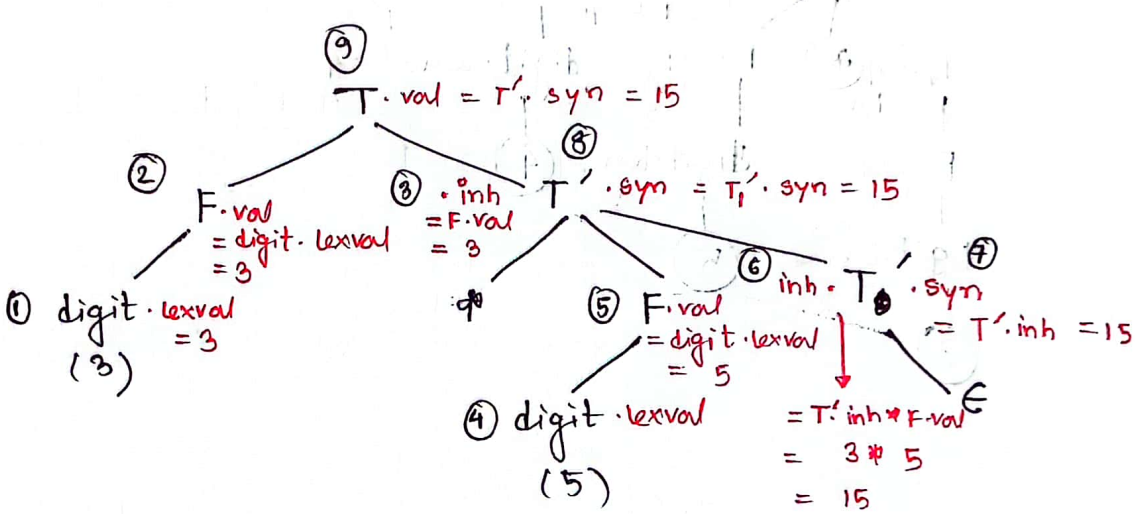
Annotated Parse tree for 3 * 5

Non term → T, F, T′ (3 ↑)

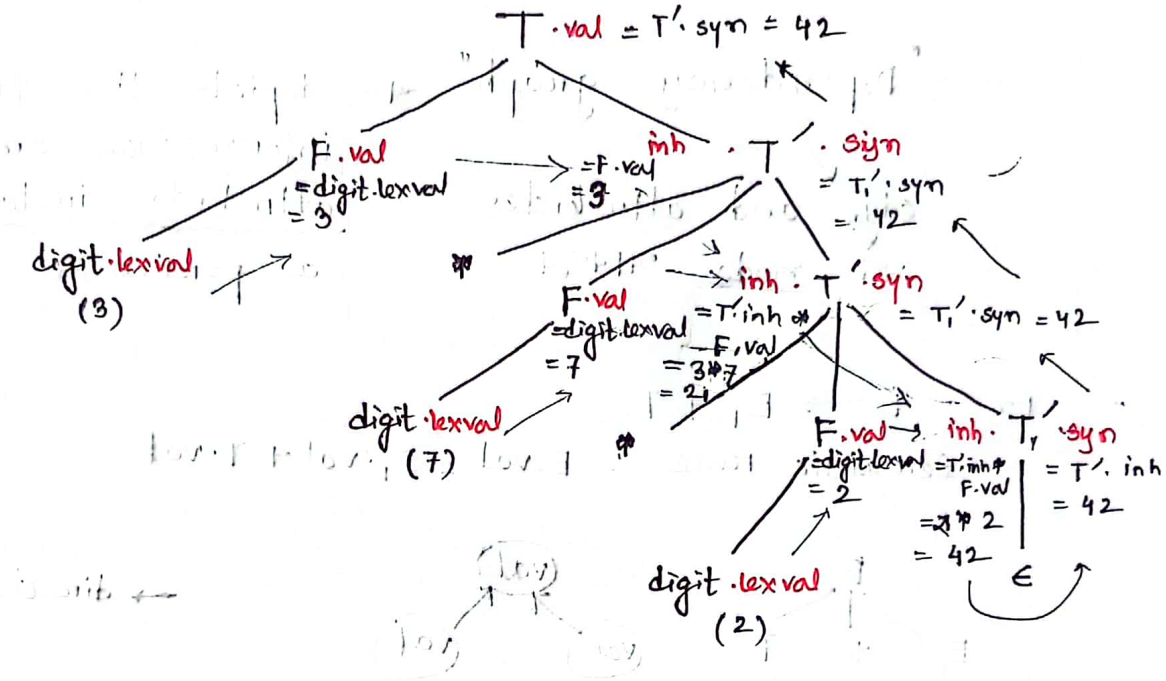
T′.inh ⟶ inherited (F sibling)

T.val → synthesized

Two types of attributes exist. So dependency check is a must.



3 * 7 * 2 :


T .val = T'.syn = 42
F.val
=digit.lexval
=3
digit.lexval
(3)
=F.val
=3
inh . T' .syn
= T1'.syn
= 42
F.val
=digit.lexval
=7
digit.lexval
(7)
inh . T' .syn
=T'.inh *
F.val
=3*7
=21
= T1'.syn = 42
F.val
=digit.lexval
= 2
digit.lexval
(2)
inh . T1' .syn
=T'.inh *
F.val
=21 * 2
= 42
= T'.inh
= 42
ε

27.06.2024

Evaluation Order of SDD's :

↳ কোন value আগে calculate করব ← to understand this.

→ "Dependency graph" ← depicts the flow of information among the attribute instances in a particular parse tree

↙ edge and attributes এর node থাকবে

Ex : $E \rightarrow E_1 + T$

Semantic rule : $E.val = E_1.val + T.val$

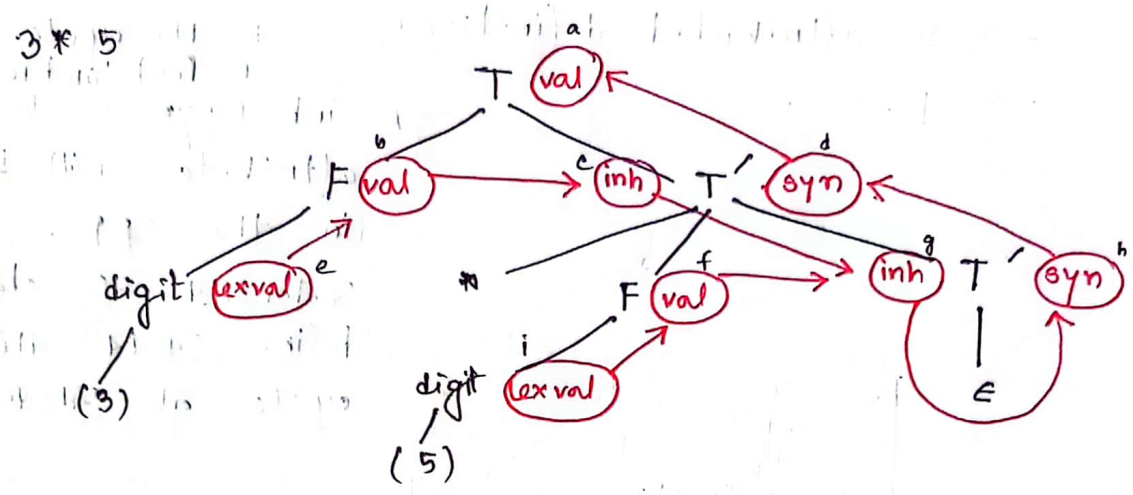
→ directed graph

$X \rightarrow AB$, semantic rule : $B.c = 2 \times (X.a)$

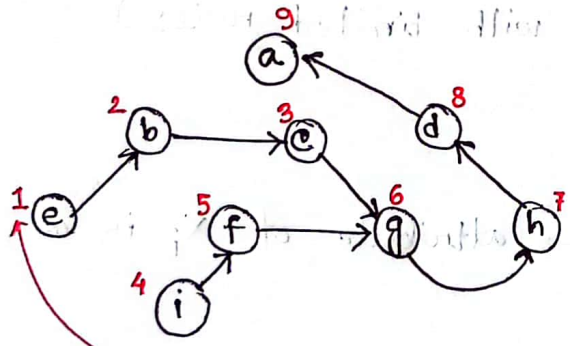
Scanned with CamScanner

Note : If there's any cycle in the graph, then we need to change the semantic rules.

Ex : 5.5 :



To sort the nodes → Topological sort ← we need to identify which node isn't dependent on anyone.



Note :
↳ e b i f c g h d a
This is also correct order. So, we can draw more than one dependency graph using topological sort.

In order যার 0 থাকে নিবো

	1	2	3	4	5	6	7	8	9
Topological sort :	e	b	c	i	f	g	h	d	a

↳ after sorting, from the dependency graph. we can easily evaluate the annotated parse tree.

Scanned with CamScanner

Classes of SDD :

→ S - attributed definitions ⟶ No cycle
⟶ Post ordering

→ L - " " ⟶ inh + syn → both attribute will be there in the SDD.

এমনভাবে এই class define করব যাতে কোনো cycle না থাকে ,

L - attributed definition :

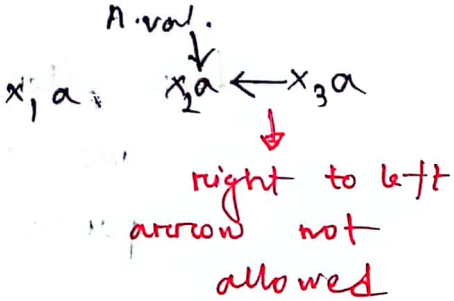
* dependency graph edges can go from left to right, but not from right to left.

* Syn + (inh attributes with limited rules)

Rules :

$A \longrightarrow X_1 X_2 X_3$ ⟶ attribute of X_i is a.

- $X_1 \cdot a = A \cdot val$ ✓
- $X_3 \cdot a = A \cdot val + X_1 \cdot a + X_2 \cdot a$ ✓
- $X_2 \cdot a = A \cdot val + X_3 \cdot a$ ✗ →



- $X_3 \cdot a = A \cdot val + X_3 \cdot val$ ✓

Scanned with CamScanner

inh → from sibling, head or itself
left এ থাকতে হবে to be L-attributed

Example → S-attributed SDD
↳ all grammar symbols have only synthesized attribute.
(child থেকে calculate হচ্ছে values)

L-attributed SDD
↳ all S-attributed SDD are L-attributed SDD because they both support synthesized attributes

Example 5.8 → Not an S-attributed SDD
(inh + syn)

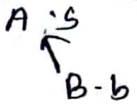
L-attributed SDD

Ex: ② $T' \rightarrow * F T_1'$ $\quad T_1'.inh = T'.inh \times F.val$
F, T_1 এর left এ
T' → head
F → sibling (left এ আছে)

① $T \rightarrow F T'$ $\quad T'.inh = F.val$
left এ
left এ sibling

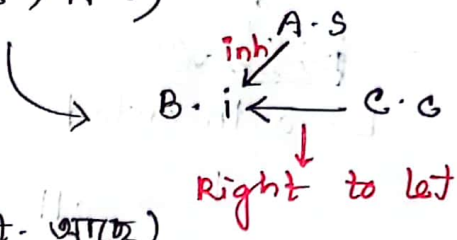
Scanned with CamScanner

Example 5.9 →



$A \rightarrow BC \qquad A.s = B.b$ (Syn)

$B.i = f(C.c, A.s)$



So, not an S-attributed (inh att. আছে)

not " L- " SDD (right sibling থেকে value নিচ্ছে)

3.07.2024

Semantic rule with controlled side effects :

SDD of desk calc ⟶ print a result
↳ side effect

" " semantic analyzer ⟶ enter type of ID into symbol table

Attribute grammar ⟶ No side effect
Translation scheme ⟶ left-right evaluation

incidental side effect ⟶ root node এ side effect যার জন্য কোনো problem হয় না।

L → En print (E.val)

production ⟶ side effect ⟶ dummy node ⟶ ঐ production এর head এর সাথে associate করবো।

5.10 SDD for simple type declarations :

int a
int a, b, c
T L

float a
float a, b, c
T L

D ⟶ T L
↓ ↓ ↓
declaration type list

T → int
T → float

L → L1, id
L → id

Scanned with CamScanner

⑧ SDD given → semantic rule generate করবা

$D \rightarrow TL$

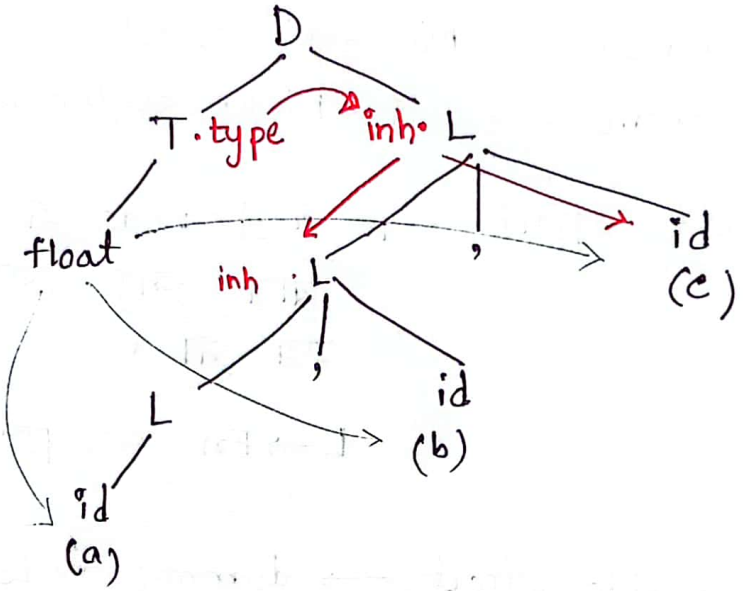
→ int L

→ int L, id

→ int L, id, id

→ int id, id, id → int a, b, c

float a, b, c :



* a, b, c এর type another branch এ (float)

↳ inherited att. ব্যবহার L এ

L.inh = T.type

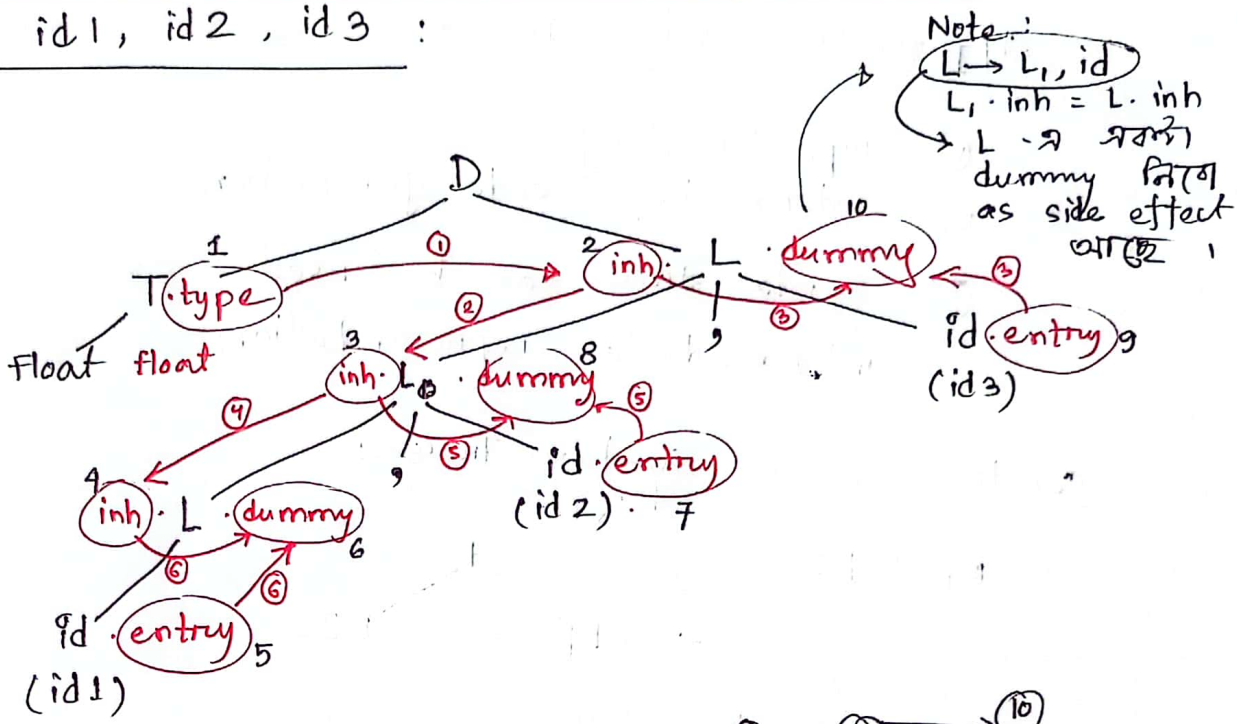
* $L_1.inh = L.inh$ (L head)

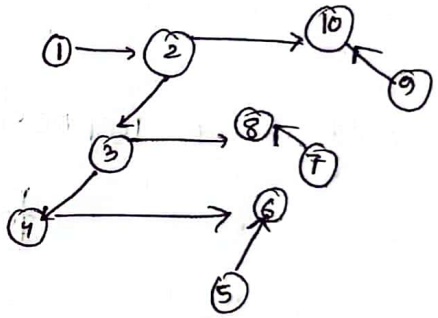
* (id)c এর type symbol T. entry করতে,

addType (id.entry, L.inh)

inherited attr. থাকলে → dependency graph

float id1, id2, id3 :





↓

we can make annotated parse tree from this.

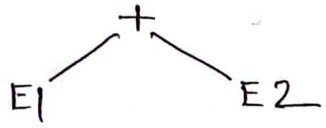
Scanned with CamScanner

Application of SDT :

1. Type checking
2. intermediate code generation
3. Syntax tree can be used as intermediate representation

construction of Syntax Trees :

$E_1 + E_2$



node ⟶ represented as record

op. field	other field

leaf ⟶ value lex. analyzer fn(a

op. fied	val. field

Interior node with two children

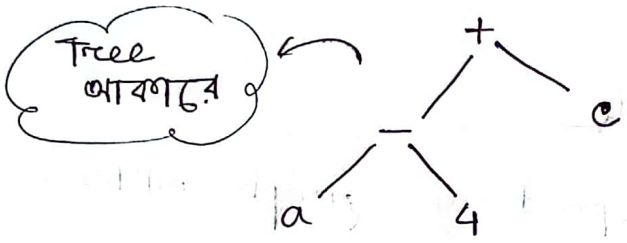
op. field	left child	right child

Ex : 5.11

- node creation ⟶ 'new' ⟶ side effect
- S-attributed definition
- Ex : a + b + 6 − 7 + c
- dependency graph লাগবে না (inh attr. নাই) ↳ S-attributed

Syntax tree for a − 4 + c :

Note: left থেকে right operation আগে হবে



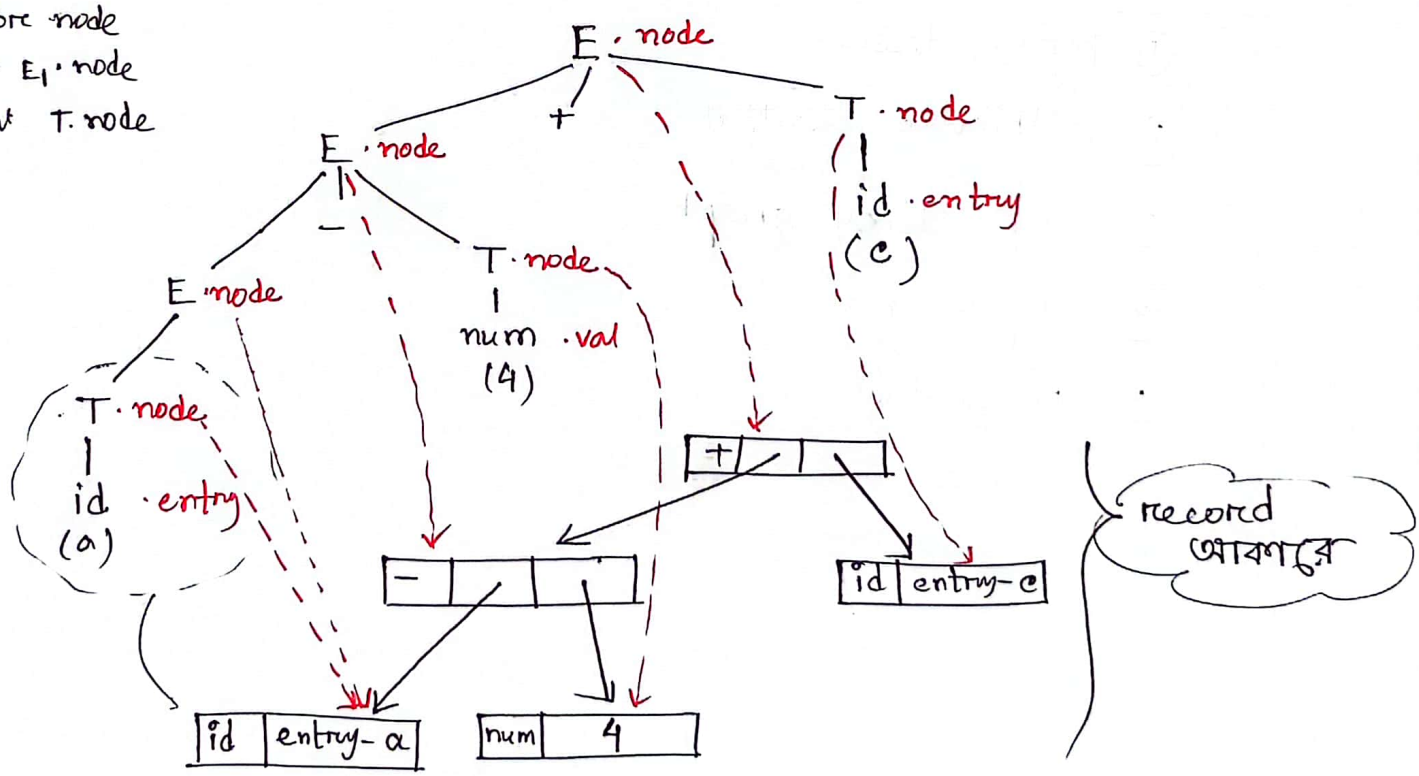
Note:

⟶ parse tree

- - - ⟶ for node reference / node pointer

⟶ ⟶ syntax tree

E ⟶ interior node
↳ left E_1 · node
⟶ right T · node



Scanned with CamScanner

parallelly this steps to be have written

```
p1 = new Leaf ( id , entry-a)
p2 = new Leaf ( num , 4 )
p3 = new Node ( '-' , p1, p2 )
p4 = n Leaf (id , entry-c)
p5 = new Node ('+' , p3 , p4)
```

5.12 –

(syn + inh) attribute

additional thing ⟶ dependency graph লাগবে.

Steps :

① Parse tree

② attribute বসাবো

③ dependency graph

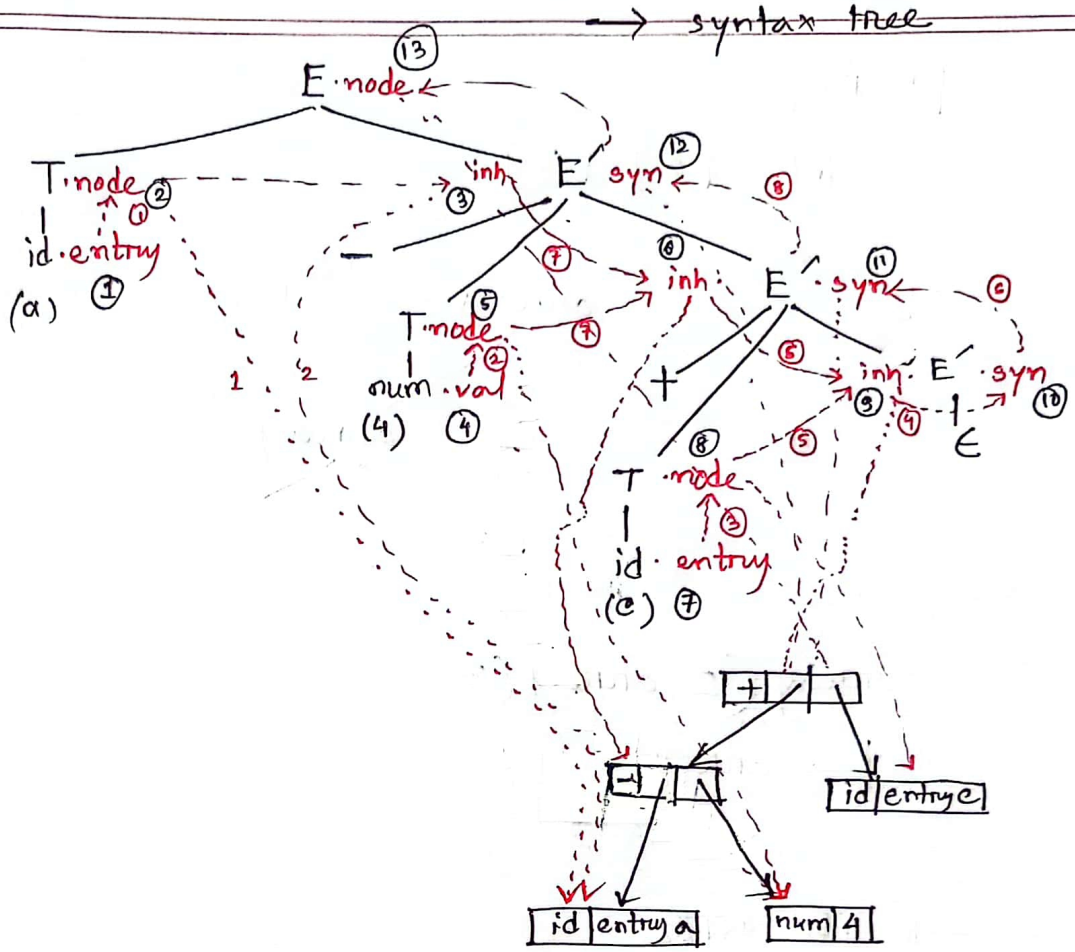
④ Topological

⑤ Evaluation

Scanned with CamScanner

a - 4 + c :

—— parse tree
---> dependency graph
•••> node ref
——> syntax tree



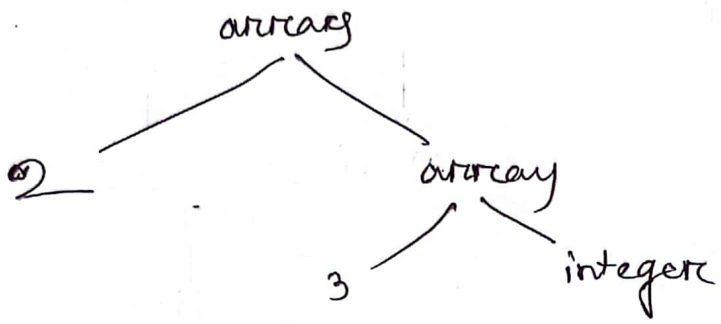
Steps of node creation :

1. P1 = leaf(id , entry - a)
2. P2 = leaf (num , 4)
3. P3 = Node (- , P1 , P2)
4. leaf (id , entry - c)
5. Node (+ , P3 , P4)

Scanned with CamScanner

5.13

int [2] [3]



array (2, array (3, integer))

↳ array of 2 arrays of 3 integers

→ inh आएगा

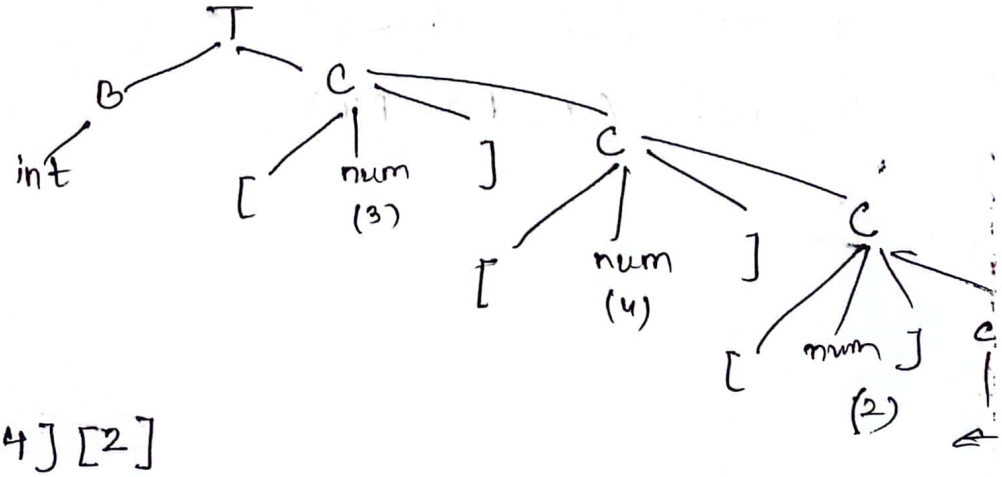
→ side effect आएगा

T → Type

B → basic type

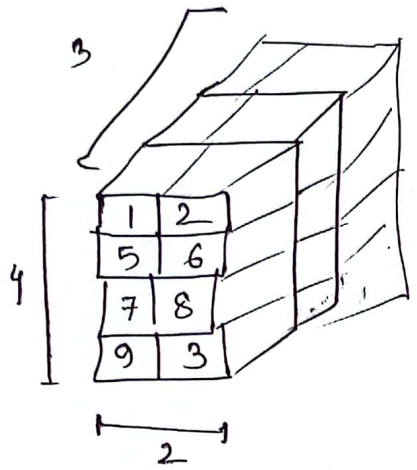
C → component

[num] → size

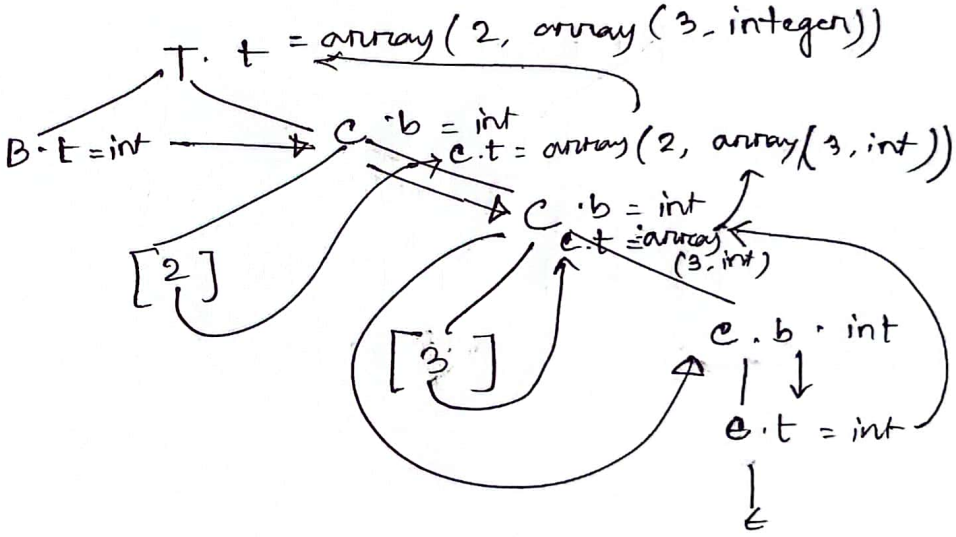


int [3] [4] [2]

Scanned with CamScanner


3
1
2
5
6
7
8
9
3
4
2

for int [2] [3] :


T. t = array (2, array (3, integer))
B. t = int
C. b = int
C. t = array (2, array (3, int))
C. b = int
C. t = array (3, int)
[2]
[3]
e. b. int
e. t = int

4.7.2024

Syntax Directed Translation Scheme :

$E \rightarrow E_1 + T$ | $E.code = E_1.code \,||\, T.code \,||\, '+'$

Production (4+5) | semantic rule (45+)

Specification → SDD

Implementation → SDT

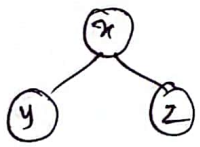
$E \rightarrow E_1 + T \; \{ print\ '+' \}$

semantic action within production body.

SDT →

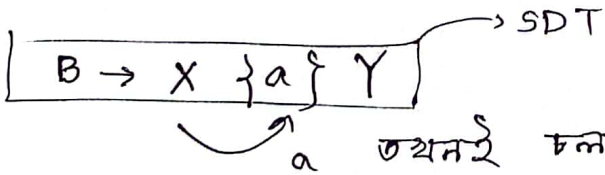
① parse tree

② preorder on left to right depth order



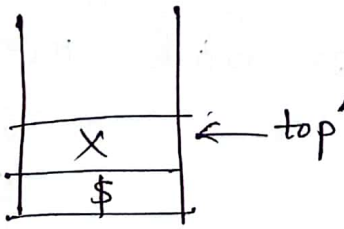
Preorder : x y z

Post " : y z x



a তখনই চলবে যখন X - terminal পেয়ে যাব

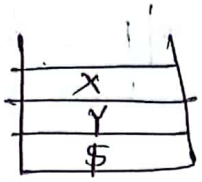
more precisely,



X top এ থাকলে action a execute হবে।

Post order = leftmost derivation

$B \rightarrow XY$



> reverse order এ stack এ যাবে।

Postfix scheme ⟶ synthesized attr. all

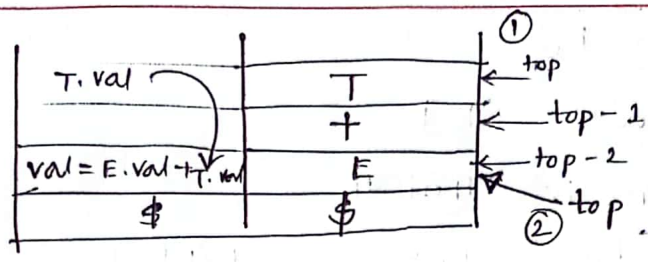
Parser stack implementation of Postfix SDT's :

⟶ LR parsing
- ⟶ rightmost derivation
- ⟶ left to right scan

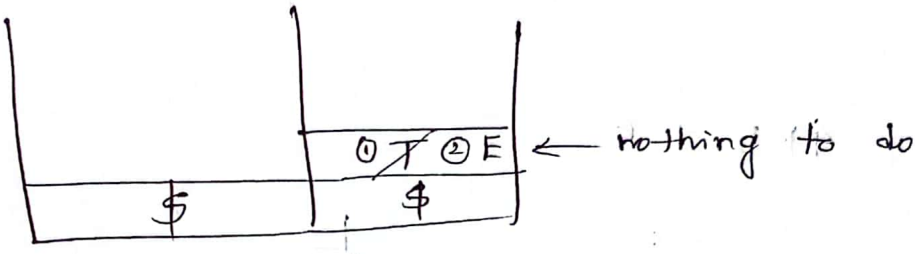
$L \rightarrow En$
$\rightarrow E+Tn$
$\rightarrow E+Fn$
$\rightarrow E+digit\ n$
$\rightarrow T+digit\ n$
$\rightarrow F+digit\ n$
$\rightarrow digit+digit\ n$

* shift
* Reduction ⟶ production এর head দিয়ে reduce করা

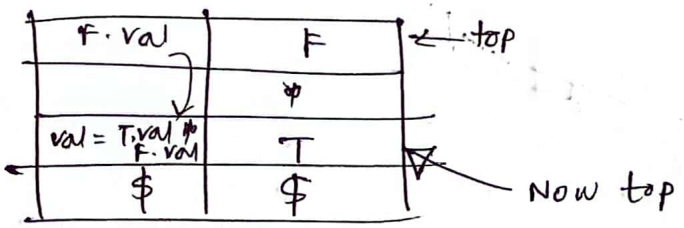
$E \rightarrow E + T$



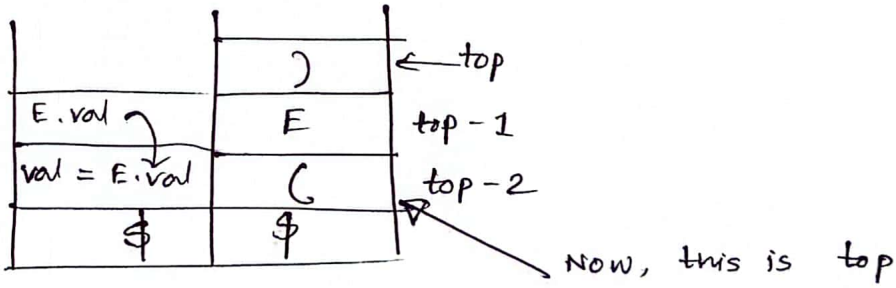
$E \rightarrow T$



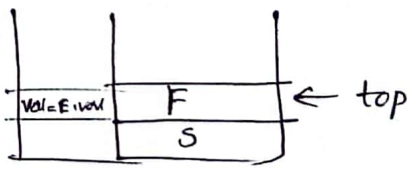
$T \rightarrow T * F$



$F \rightarrow (E)$



-) appeared on top of the stack
- Perform the semantic action (} action {)
- Reduce the production



as $F \rightarrow (E)$

Scanned with CamScanner

Ex : 5.16

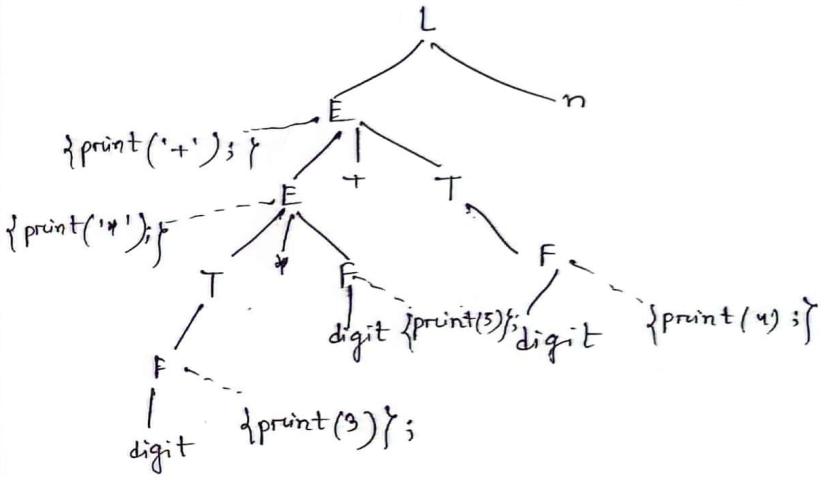
Problematic SDT

↳ Prefix বানাবো

(Ex : 4+5 ⟶ +45)

4*5 ⟶ *45

4*5 + 6



Scanned with CamScanner

10.7.2024
Zinia ma'am

Chapter 6
Intermediate-code Generation

Parser → Static checker → Int. code generator → (int. code) → code generator

(Int. code generator: very close to target code)

front end | backend

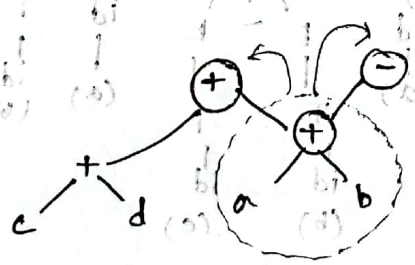
* No lexical error
* No syntax "
* No semantic "

Source prog. (C++) → High level Int. representation (C) ----> Low level Int. representation (assembly) → Target code

then intermediate code generator চালু হয়।

Variants of Syntax trees :

- DAG (Directed acyclic graph)
  - → leaves → operands
    interior node → operator
  - → common subexpression → multiple parent



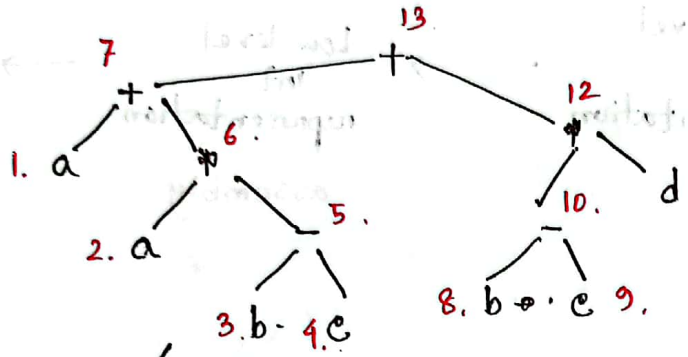
→ efficient code generation → help করে
→ same subexpression 2 বার use করে।

6.1

$$a + a * (b - c) + (b - c) * d$$

→ b − c common subexpression

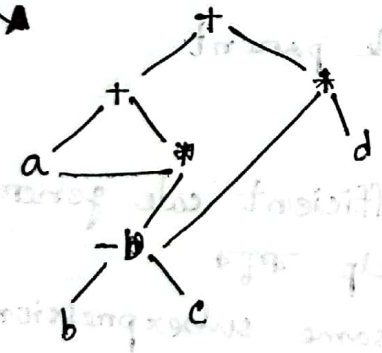
→ terminal এ duplication থাকবে না

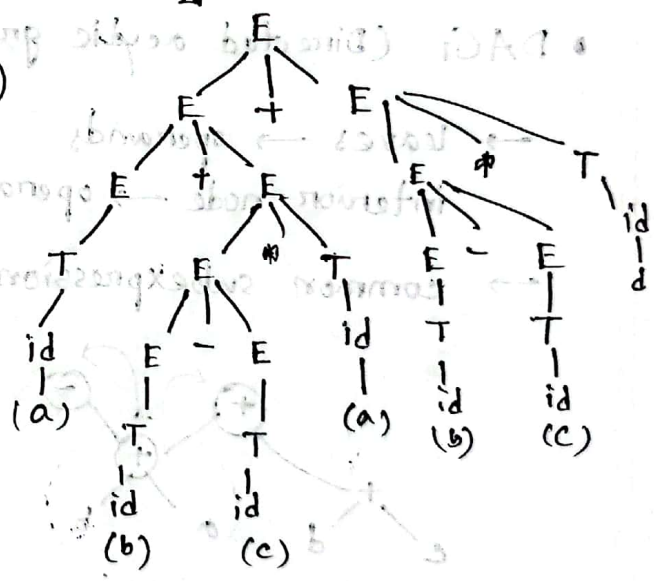


E → E + E
→ E − E
→ E * T

T → id
→ (E)

Not atomic
(a − 2, b − 2, c − 2, d − 1 বার)



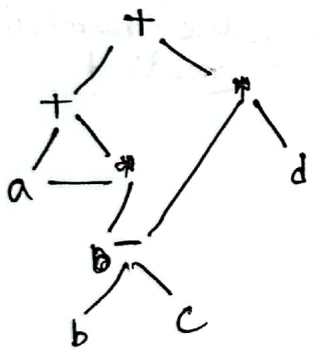


↳ atomic operands

→ Syntax tree in DAG form

6.1

$E \to E_1 + T$	E.node = new node ('+', E_1.node, T.node)
$E \to E_1 - T$	E.node = " " ('-', " ")
$E \to T$	E.node = T.node
$T \to T_1 * F$	T.node = new node ('*', T_1.node, F.node)
$T \to T_1 / F$	
$T \to F$	
$F \to (E)$	
$F \to id$	F.node = new leaf (id, id.entry)
$F \to num$	F.node = " " (num, num val)



1. a প্রথমবার leaf
   P_1 = leaf (id, entry-a)
2. P_2 = leaf (id, entry-a) = P_1 (No physical existance)
   ↓ as a আগে create করা আছে
3. P_3 = leaf (id, entry-b)
4. P_4 = " " (", " -c)
5. P_5 = Node ('-', P_3, P_4) → (b-c)
6. P_6 = Node ('*', P_1, P_5) → a*(b-c)
7. P_7 = Node ('+', P_1, P_6) → a+a*(b-c)
8. P_8 = leaf (id, entry (b)) = P_3
9. P_9 = leaf (", " -c) = P_4
10. P_{10} = Node ('-', P_3, P_4) = P_5
11. P_{11} = leaf (id, entry-d)
12. P_{12} = Node ('*', P_5, P_{11})
13. P_{13} = Node ('+', P_7, P_{12})

P_2, P_8, P_9, P_{10}
→ No physical existance
→ previous created nodes are used.

Scanned with CamScanner

DAG এর construction :

Value - Number method :

→ nodes → stored in an array of records

→ | operation | val |

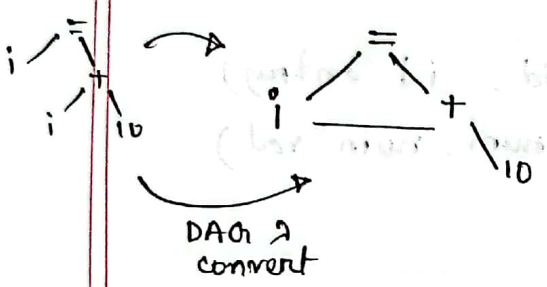
| op | left child | right child |

↘ multidimensional

↓

index number দিয়ে refer করি।

DAG for i = i + 10

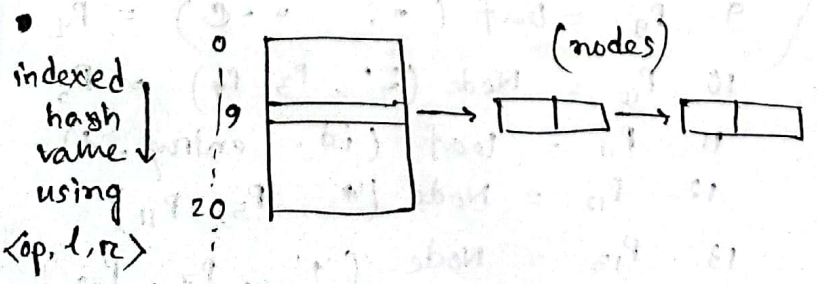


1.	id	→ to entry for i	
2.	num	10	
3.	+	1	2
4.	=	1	3
5	. . . . .		

id	entry
num	10

↳ এর array construction method → value - number method.

- op, l, r → <op, l, r>
  - op → operation
  - l → left child
  - r → right child
- value number means → index number
- search করে <op, l, r> পেয়ে গেলে → return index number
- for faster searching → HASH TABLE
-



Three address code :

→ at most 1 operator at right side of an expression

$x + y * z \longrightarrow t1 = y * z$
$t2 = x + t1$

* এর precedence বেশি.
So, $y * z$ then $x + (y * z)$

compiler - generated temporary name

→ linearized → sequential

6.4

$a + a * (b - c) + (b - c) * d$

$t1 = b - c$
$t2 = a * t1$
$t3 = a + t2$ → three address code
$t4 = d * t1$
$t5 = t3 + t4$

two concept → address
→ instruction

- addresses can be name (Ex : a, b, c, d), constant (10, 20) compiler generated temp. variable (t1, t2, etc)
- instruction :
  → assignment inst : $x = y$ op z
  → binary, arithmatic / logic operator

Scanned with CamScanner

→ assignment of form $x = op\ y$ (op: unary, logical negation, shift operator or conversion operator)

Ex : $a = -c$

$a = (float)\ c$

→ copy instruction → $x = y$ Ex : $a = b$

→ unconditional jump → goto L

↓

যেখানে jump করতে চাই

→ conditional jump

- if x goto L → x true হলে L এ যাবে
- If False x goto L → x false " L " "

→ condition jumps → if x relop y goto L

3 address

→ Procedure call and return

return করুক না করুক

- param x
- call p, n → no. of parameter
  ↓ function name
- y = call p, n

  ↳ function কিছু return করলে

  return y

add (5,6) → add (int a, int b)

param 5
param 6
t1 = call add, 2

return a+b;

$p(x_1, x_2, \ldots x_n)$

↳ param x_1
param x_2
⋮
param x_n
} কোনো parameter redundant হবে না
call p, n

→ indexed copy instruction ⟶ $x = y[i]$

↓
let, 100 থেকে start

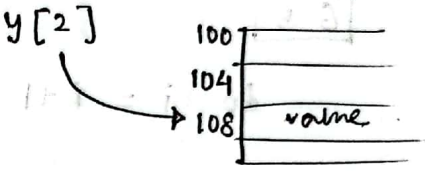
$x[i] = y$

↳ let $i = 1$

$x[1] = y$

↓
if it's float
108 এ y store হবে

y[2]



int হলে y[1] → 104 এ
y[2] → 108 "

Example 1

$n = f(a[i])$, a ⟶ array of int (4 byte করে ধরে নিচ্ছি)

three address code :

$t1 = i * 4$
$t2 = a[t1]$
param t2
$t3 = call\ f, 1$
$n = t3$

Example 2

$n = f(a, g(a))$

↳

param a
$t1 = call\ g, 1$, param t1
$t2 = call\ f, 2$
$n = t2$

→ address and pointer assignment

$x = \&y$ $\quad x = *y$ $\quad *x = y$

↓ address आवश्यक

(sets the r(real)-value of x ← pointer to be the location (l value) of y ← name)

$x = *y$ → y is pointer and r-value is location

6.5

do i = i+1 ; while (a[i] < v) ;

→ L: t1 = i + 1 let, a[i] → double type

i = t1

t2 = i * 8

t3 = a[t2]

if t3 < v goto L

symbolic label

→

100 : t1 = i+1

101 : i = t1

102 : t2 = i * 8

103 : t3 = a[t2]

104 : if t3 < v goto 100

Scanned with CamScanner

HOME TASK

(9) do {

a = x[i];

p = add (a, 5);

i = i+2;

} while (a<100 && i<6);

↳ L : t1 = i * 4

t2 = x[t1]

a = t2

param a

param 5

t3 = call add, 2

p = t3

t4 = i + 2

i = t4

if t2 <100 goto L1

goto L6

L1 : if t4 <6 goto L

L6 :

11.7.2024

Quadruples :

4 field

op, arg1, arg2, result

* x = y + z → arg1, arg2; result ← x; op ← +

op	arg1	arg2	result
+	y	z	x

unary operation

* x = minus y → don't use arg2

minus	y		x

copy statement

* x = y

=	y		x

* param x

param	x		

not used

Unconditional jump

* goto L

goto			L

Conditional jump

* if x goto L

goto	x		L

* if x < y goto L

<	x	y	t1
goto	t1		L

Scanned with CamScanner

$a = b * -c + b * -c$

three address code.

$t1 = \text{minus } c$

$t2 = b * t1$

$t3 = t2 + t2$

$a = t3$

this is also correct (optimization)

$t3 = \text{minus } c$

$t4 = b * t3$

$t5 = t2 + t4$

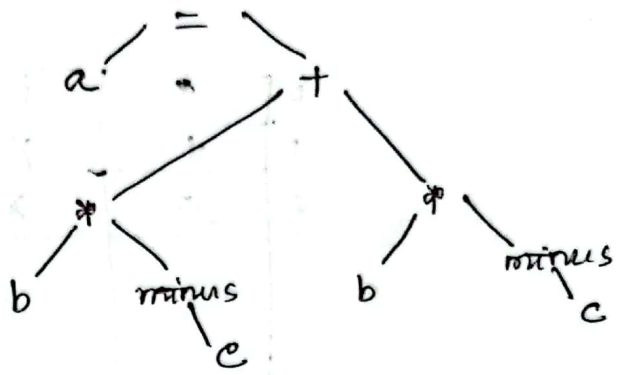
$a = t5$

	op	arg 1	arg 2	result
0	minus	c		t1
1	*	b	t1	t2
2	minus	c		t3
3	*	b	t3	t4
4	+	t2	t4	t5
5	=	t5		a

quadruple

Triple :

op, arg 1, arg 2
↳ result refer করার



Syntax tree

0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

* copy statement (different)

a = t5

=	a	t5

(যেখানে copy করে রাখবো)

(যাকে copy করবো)

x [i] = 5

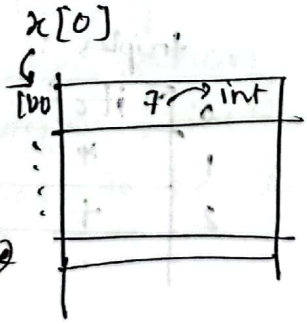
x [2] = 5

2 * 4 (for int)

2 * 8 (for double)

add with

x[0] → x এর base



* x[i] = y

	op	arg 1	arg 2
0	*	i	4
1	+	x	(0)
2	=	(1)	y

Scanned with CamScanner

* $x = y[i]$

0	*	i	4
1	+	y	(0)
2	=	x	(1)

Quadruple is more efficient
↳ optimizing compiler.

* Before optimizing :

$t1 = itof(60)$
$t2 = a * t1$
$t3 = t2 + 40$

quadruples :

0	itof	60		t1
1	*	a	t1	t2
2	+	t2	40	t3

triples :

0	itof	60	
1	*	a	(0)
2	+	(1)	40

Scanned with CamScanner

optimizing :

$t2 = a * 60.0$

$t3 = t2 + 40$

0	*	a	60.0	t2
1	+	t2	40	t3

0	*	a	60.0
1	+	(0)	40

Before optimizing,

2 → +, (1), 40

Now, 1 → +, (0), 40

reference is changes

↓

So, we use Indirect triples.

INSTRUCTION field is used

↓

optimization এর জন্য, instruction has to be deleted. (but it also has a problem; creates hole).

Scanned with CamScanner

Static single assignment form (SSAF):

একটা variable কে 2 বার store করবে না।

```
p = a+b          P1 = a + b
p = q * d        p2 = q1 * b
   ↓                 ↓
Three address      SSAF
   code
```

same position

Problem:

```
if (flag)  x = -1 ;  else  x = 1
   ↓
y = x * a
```

x_1, x_2 → but যাবে কোন x.

```
if (flag)  x1 = -1  else  x2 = 1
x3 = φ (x1, x2)   ← right result pick করে।
y = x * a
```

Practice problem 1 ⟶ Do it yourself.

Scanned with CamScanner

5.9.2024
Zinia ma'am

Types & Declarations :

* ensures the types of operand ~~math~~ match the type expected by operator.

```
boolean x ;
string  y ;
~~boo~~
```

x = y → syntax ✓ but, meaning isn't right (boolean ≠ string)

boolean = string ⟶ type expression

equivalency
- Name
- struct
  - strict
  - less strict

Type expressions :

→ set of basic types & constructors.

int [2][3]
- → array of 2 arrays of 3 int.
- → Type expression :

array(2, array(3, int))
- array → type constructor
- int → basic type

operator array takes
- number
- a basic type

- boolean, char, int, float void ⟶ basic type

Ex :

float x ; T.E : float

int i ; T.E : int

if I write x = i ⟶ it will check float = int

" " . i/x ⟶ (int) i / (float) x

res will be in float.

- Type name is a T.E.

array (2, array (3, int))

number T.E

- record is a data structure with named field.

float x ⟶ T.E : float

struct { float x ; float y ; } P ; ⟶ record (x : float, y : float)

- function এর জন্য we use ⟶ .

s ⟶ t

parameter ← return type

function int f(float x) ⟶ T.E : float ⟶ int

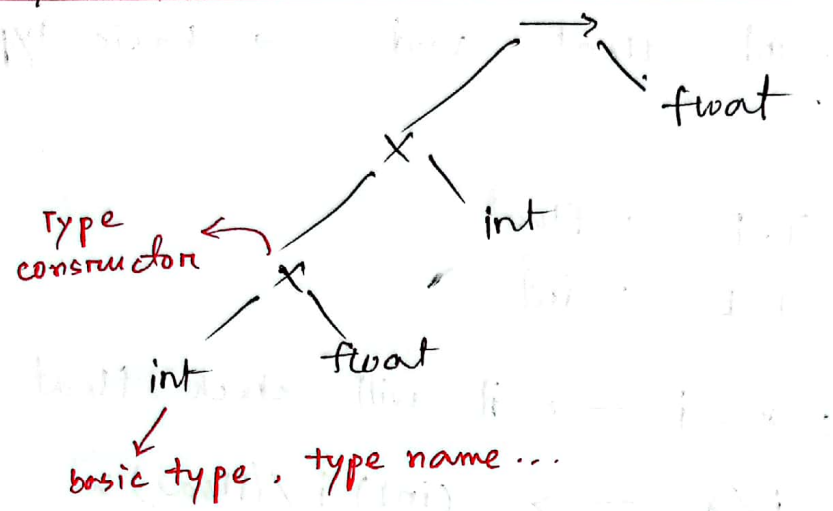
" float g(int a, float b, int c) ⟶ T.E : int x float x int ⟶ float

X ⟶ left associativity

int x float x int

① ②

Type Expression Tree : int × float × int → float.



Type equivalence :

if two type expression are equal, then return a certain type else error ;
↳ Semantic error.

- Structurally eq.
- Name equi.

Structurally eq :

→ same basic type (int = int)
→ applying same constructor (array(1) = array(3))
→ one is a type name that denotes the other.

Name equivalence :

```
struct ST {
    int a ;
    float b ;
} x , Y ;
```

```
struct T {
    int a
    float b
} M , N ;
```

ST = ST → Name Eq ✓

Type name ↑ ↑

X = Y

M = N

↓ ↓ Type name

T → Name equiv ✓

X = M

↓ ↓

ST ≠ T → Not equivalent (doesn't support name equivalence)

strict eq :

ST → int a
float b

T → int a
float b

→ প্রথম member এর order same and name same (int then float & a ↔ int, b ↔ float)

→ তখন strict equiv.

less eq :

member এর order same থাকলেই হবে ,

struct. eq :

X = M

↓ ↓

ST = T

Declarations :

$D \rightarrow T\ id\ ;\ D \mid \epsilon$

$T \rightarrow B\ C \mid record\ \{\ D\ \}$

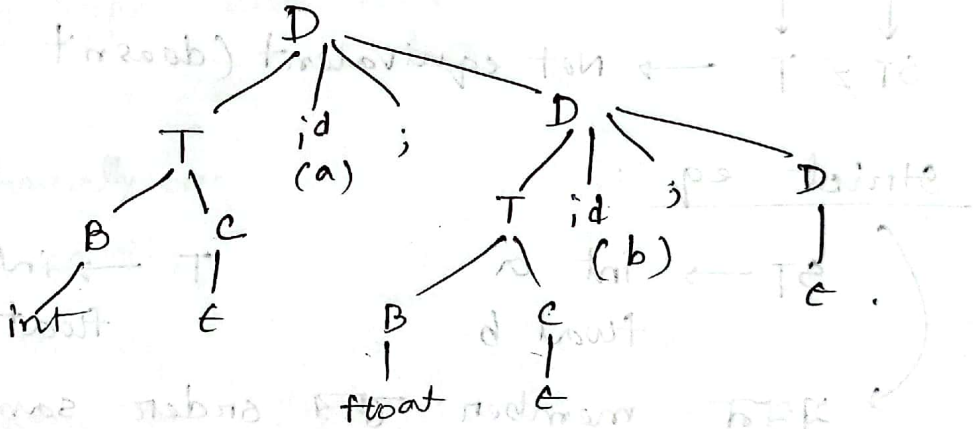
record ⟶ for struct

C → component
T → type
B → basic type

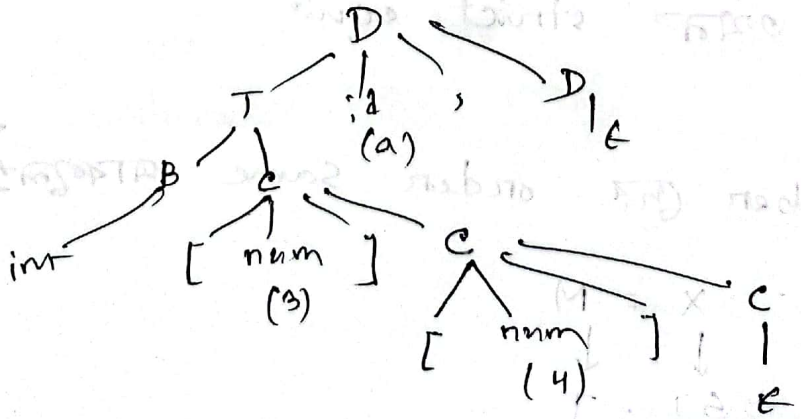
$B \rightarrow int \mid float$

$C \rightarrow \epsilon \mid [num]\ C$

int a ; float b ;



int [3] [4] a ;



Scanned with CamScanner

11 . 9 . 2024
Zinia ma'am

Storage layout for local names :

From type of name ⟶ determine the amount of storage needed in run time
↓
at compile time

varying length ⟶ Ex : dynamic
↳ fixed amount of storage reserve আবশ্যক ছোয়া হয় ।

- storage ⟶ blocks of contiguous bytes
  (8 bits)
  বহুবচন byte (generally 2 byte) ⟶ word

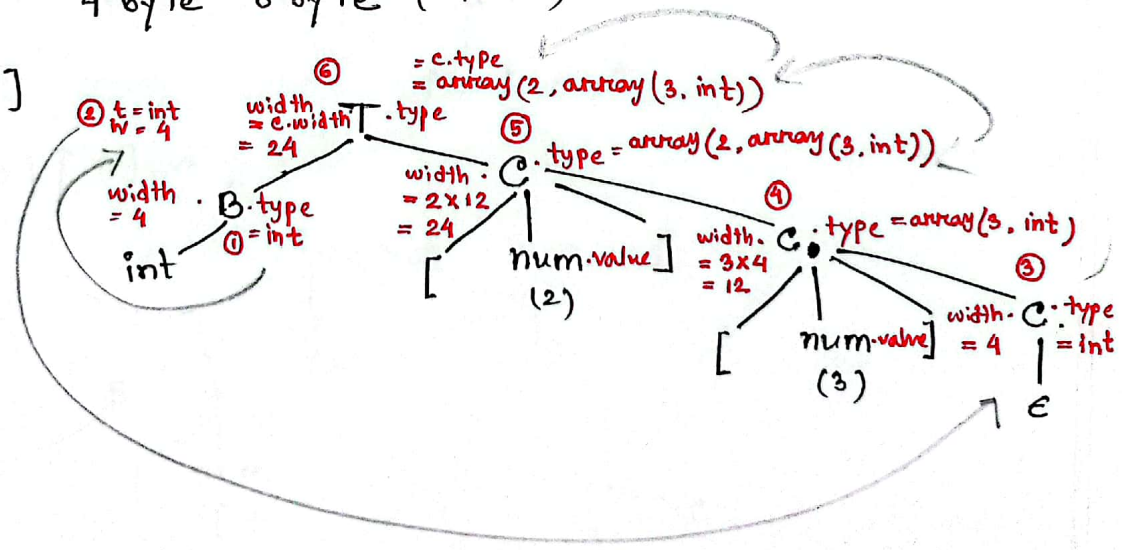
Ex. 6.9 :

SDT of array types

$T \rightarrow B \{ \ldots \} C \{ \ldots \}$

type ⟶ int , float
↓ ↓
4 byte 8 byte (width)

int [2][3]



Ex 6.9 vs Ex : 5.13 :

- No inherited attr.
all synthesized.
- Type & width declare করে
- S attributed grammar

- basic type or array type declare করে।
- T → B · C ; C.b = B.t.
here b is inherited attr. as it's being taken from it's sibling.
Not an S attr. grammar

Sequence of declaration :

relative address বের করে

P → ϵ | D → প্রতি program এর first এ offset = 0 বসবি।
Program, declaration

int a ; ⟶ (a, int, 0) → A symbol table এ যাবে
type
symbolTable.insert(a, int, 0)
id.lexeme, offset

then, offset = 0 + 4
= 4

int → width 4 byte
float → " 8 "
char → " 1 "

float b ; → SymbolTable.insert(b, int, 4)
Now, offset = 4 + 8 = 12

char c ; → SymbolTable.insert(c, char, 12)
Now, offset = 12 + 1 = 13

Scanned with CamScanner

Three address code for expression :

concate করা যাবে না

$a = 5 + 4 - 3$ ✗ $\quad e = a + (b + c) + (-d)$ ✓

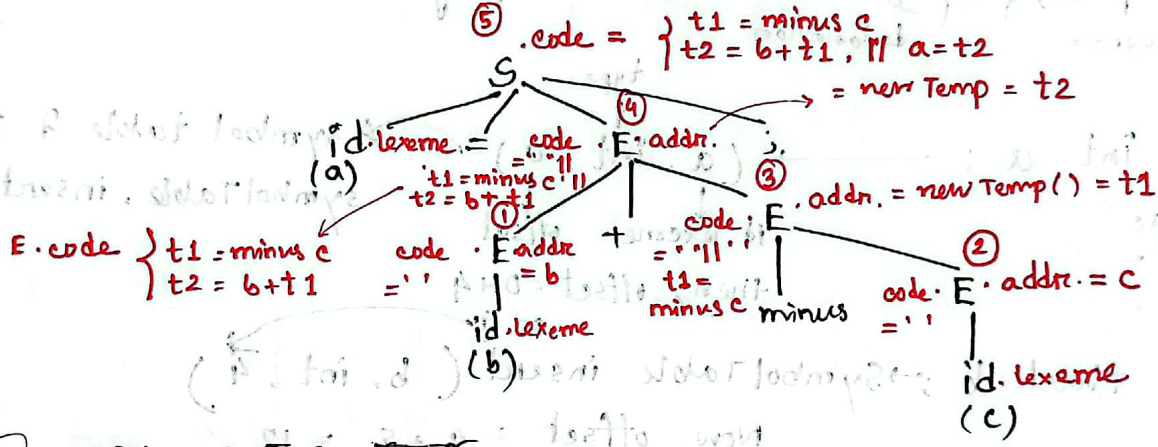
$a = b + c + d$ ✓

.code ⟶ three address code

expression $a = b + -c$ ⟶ Three address code ⟶ $t_1 = \text{minus } c$

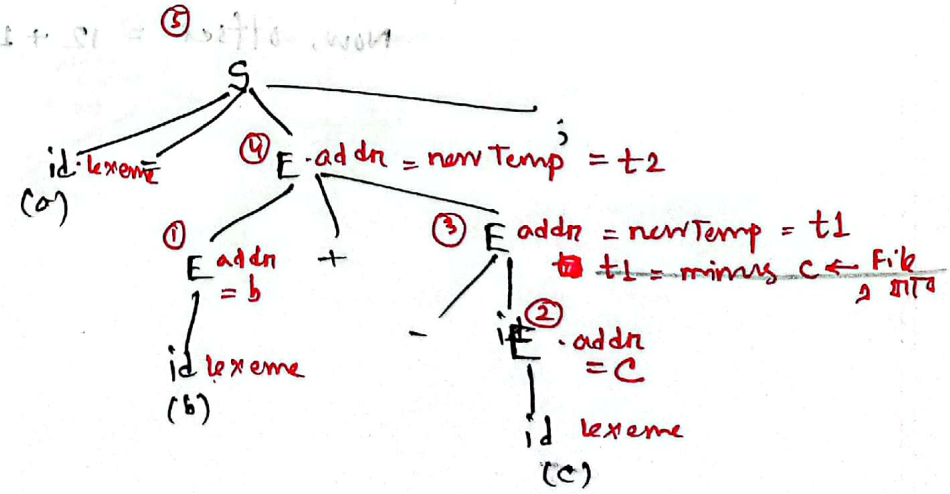
$t_2 = b + t_1$

$a = t_2$



Incremental Translation ⟶ File এ TAC থাকে

⟶ directly লেখা হয়



gen function সব file এ লিখে রাখবে

TAC File :

```
t1 = minus c
t2 = b + t1
a = t2
```

Arr [5] → base add. → Arr = & Arr[0]

two dimension → base + (i x width) + (j x width)

Translation of Array References - SDT

a [i] → a = int array

↳ t1 = i * 4

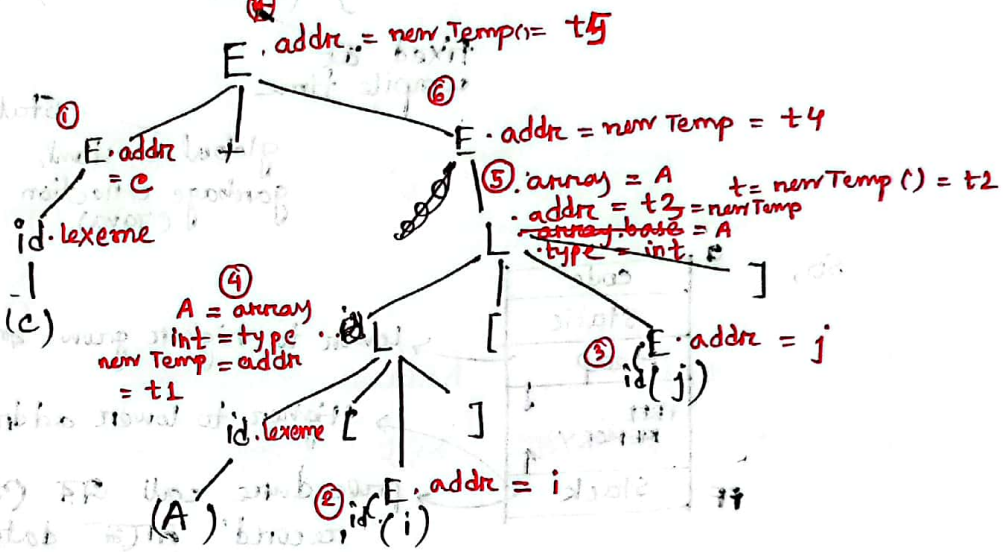
i. y = a [t1]

a [i+t] or b [i+t] [j+t] or a [b [i+t]]

Incremental translation

x = A [i] [j] TAC →

let, A → int array



File for TAC

int type ka width

① → t1 = i * 4

⑤ → t2 = j * 4 ← t = j * 4 = t2

t3 = t1 + t2

⑥ → t4 = A [t3]

⑦ → t5 = c + t4

Home task → Practice

12.9.2024
Zinia ma'am

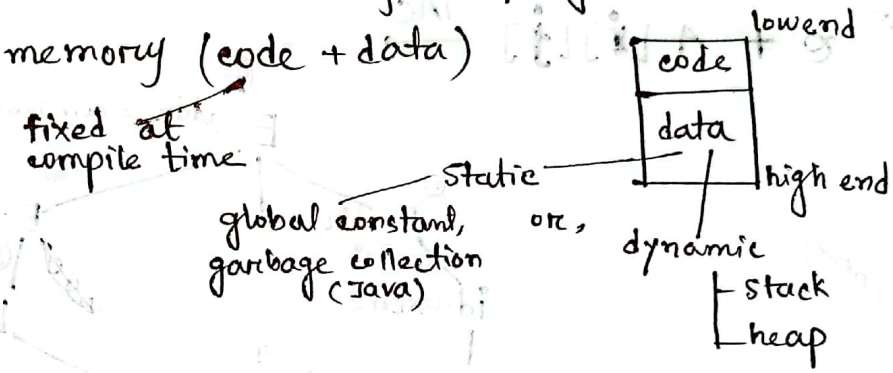
CHAPTER 7
Run-time Environment

Storage organization :

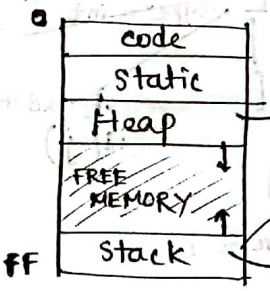
* Target program এর own logical address space এ run হয়.
* compiler, OS & target machine এই space share করে
* OS map করে physical address এর সাথে।

Run-time representation of object program

↳ run-time memory (code + data)



So,



→ Lower to higher grow করে address

→ Higher to lower address grow করে

→ procedure call এর ক্ষেত্রে এখানে 'activation record' নামে data structure store হয়

Activation record
→ value of program counter
→ machine regs. when proc. call occurs

Heap ⟶ malloc, new etc dynamic data এর ক্ষেত্রে use হয়

Padding :

* integer placed at an address divisible by 4.
* char (10 char) ⟶ 12 byte assign করে compiler
* space left due to alignment consideration

↳ Padding → যখন space তখন compiler data pack করে যেখানে where no padding is left.

Scanned with CamScanner

Static vs Dynamic storage Allocation :

- compile time এই storage কতটুকু needed জানা যাবে
- compile time এ জানা যায় না।

Stack → local to procedures, those names are allocated space on stack.

Heap → calling function এর বাইরেও data থাকলে

Stack Allocation Space :

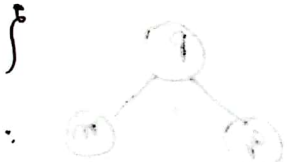
- Push → when proc calls
- Pop → when proc terminates

Activation Tree :

node → একেকটা activation record

* একটা func. এর কাজ শেষ হলে আরেকটা start হবে।

```
add (5,6)}
get();  → at first কাজ করবে
          then add func. কাজ করবে।
```

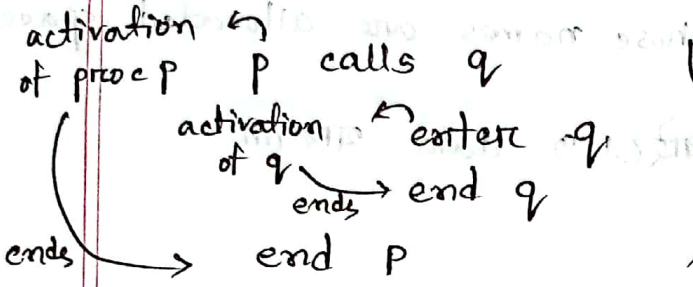


Ex : 7.1 :

→ readArray() ; A[1] to A[9]
→ sentinel set করা
→ call Quicksort func. → partition func. call.
→ recursively Q.S. " "

17.9.2024

qs (1, 9)
↓ P=4
qs (1, 3) qs (5, 9)

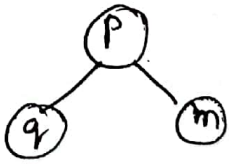


* q ~~normally~~ normally terminates
* program failure হলে q ends, p also ends.

- nodes represent activation
  root: starting of program
  (For C, it's main)

node এর child হবে তারা যারা node এর proc থেকে call হয়েছে।

Ex: P() {
  q();
  m();
}

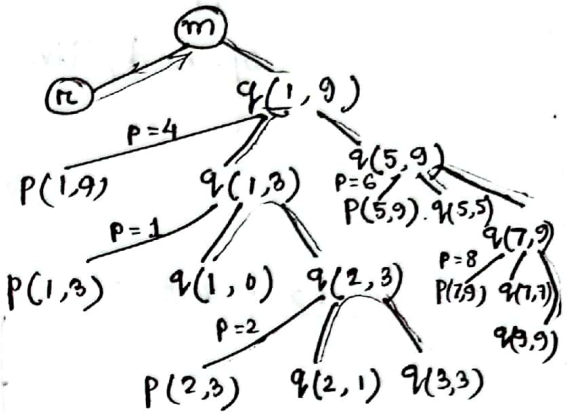


→ left to right
→ in order they're called.
→ q end হলেই m এর activation হবে।

n>m হলেই partition হবে ;

root → main
partition, QS, readArray → nodes
edge → যা return করবে

```
int a[];
void readArray() {
    int i ;  → local
{
int partition() {
}
void QS() {
    if (n> m) {
        i = partition(m, n);
        QS(m, i-1);
        QS(i+1, m);
    }
}
main() {
    readArray();
    =
    =
    QS(1,9);
}
```



** Preorder Traversal → root left right
↳ calling sequence পাই।

** post order (L,R,Root)
↳ return

* q(7,9) activated means → q(5,9)
q(1,9)
m

↳ এই node এর ancestors activated থাকবে

17.9.2024
2nd Part
(Zinia ma'am)

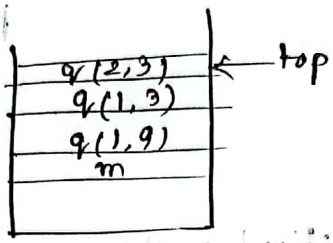
~~opp~~

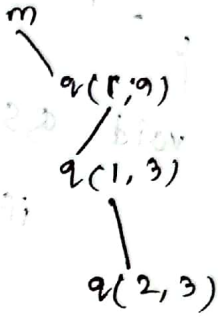
opened / live node n ⟶ তার ancestor node ও alive.

↙

stack এর top এ থাকবে

Ex : 7.3 :





Data struc with 7 items

Contents of activation record :

* Temporaries ⟶ compiler generated temporary
* Local data ⟶ int sum(int , int)

int sum = a+b;
return sum;

local for sum().

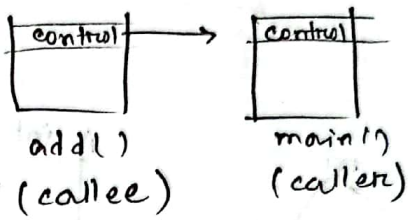
* Saved machine status ⟶ contents of register

main() {
add(5,6); ⟶ PC, SP, IP
↳ store করতে হবে in these register
}

* Access link ⟶ a pointer to locate data from another activation record.

* control link ⟶ a pointer to A.R of caller

```
main( ){
    add( );
}
```



* returned value ⟶ ↓ না থাকলে খালি

```
int sumf(_ , _ )
{
    int sum = a+b;
    return sum;
}
```

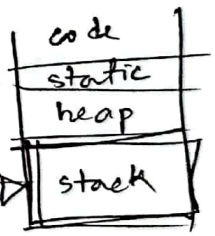
returned value return value রাখতে হবে,

for efficiency ⟶ register এ

* actual parameter ⟶ a and b.

Ex : 7.4

activation record = frame

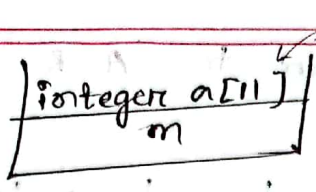


{ actual param
control link

p(1, 3), q(, , 9), m ⟶ 3 টা live node ⟶ stack
⇒ 3 টার activation record থাকবে,

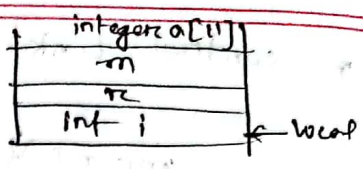
Scanned with CamScanner

Downward growing stack of Activation Record

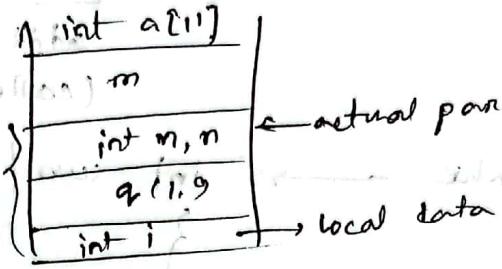


global

m



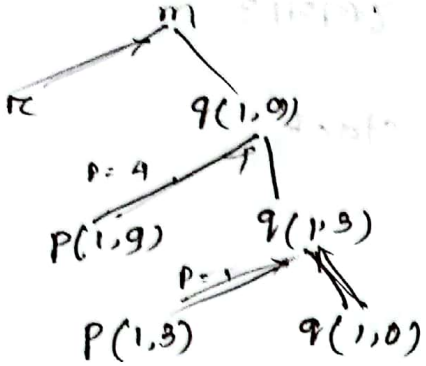
rc → return
↓
popped

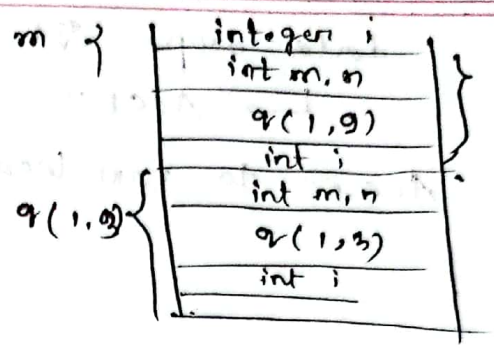


top of stack → bottom of diagram

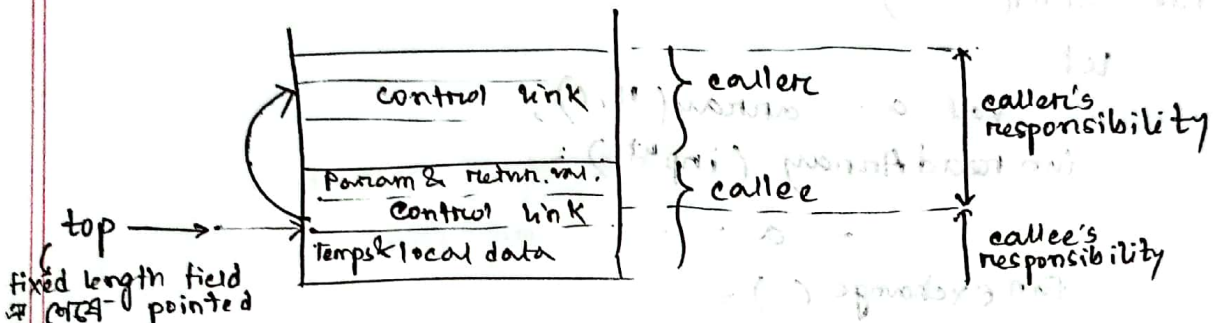
18.9.2024

74



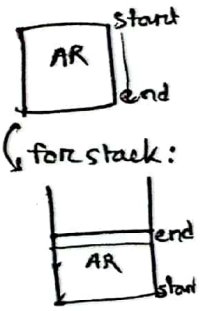


calling sequences :



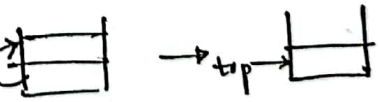
caller
↑
main }

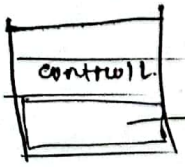
add(5,6);
↓ ↓
caller এ actual parameter set করে



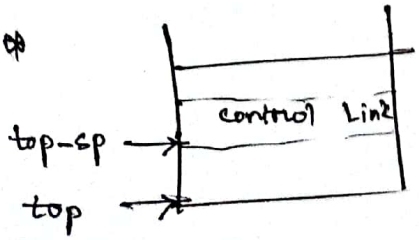
fixed length item
→ control link
→ Access link
→ Machine status field
} middle এ রাখা হয়

call এর ক্ষেত্রে downward grow
return " " "





→ variable length data থাকবে



· data scope এর বাইরে অন্য কোথাও
  ↳ ACCESS LINK

· Access to non local data on stack :

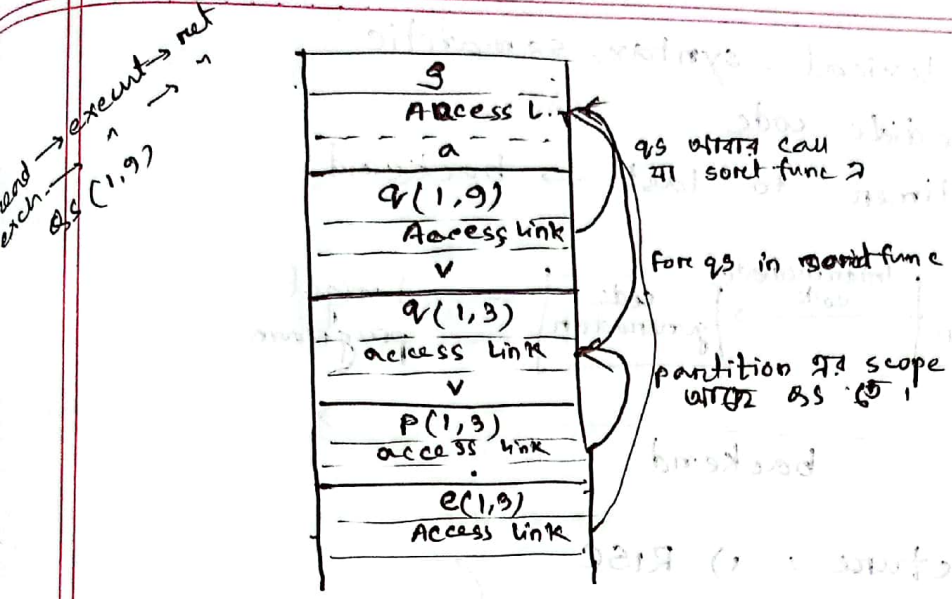
QS using nested f :

```
* fun sort( ) =
    let
        val a = array(11,0);
        fun readArray (input) =
            -- a --;
        fun exchange ( ) =
            -- a --;

        fun QS (m, n) =
            let
            val v = --;
            fun partition (y, z) =
                -- a -- v -- exchange --
            in
                -- a -- v -- partition -- QS
            end
    in
        -- a -- readArray -- QS --
```

sort এর body তে

QS এর body তে



qs, exchange → in sort
partition → in qs

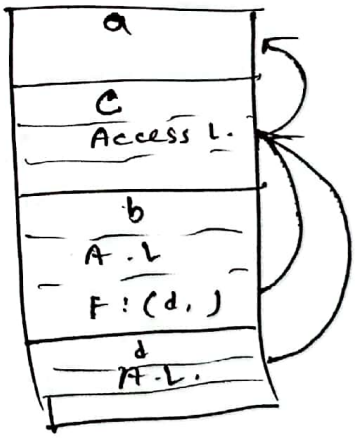
Example :

b(d)
↑
root function

→ nested function allow
→ func. as parameter.

c → inside a

let → declarations
in → body of func.



7.3 → 7.3.1 , 7.3.2 , 7.3.4, 7.3.5 , 7.3.6 , 7.3.7

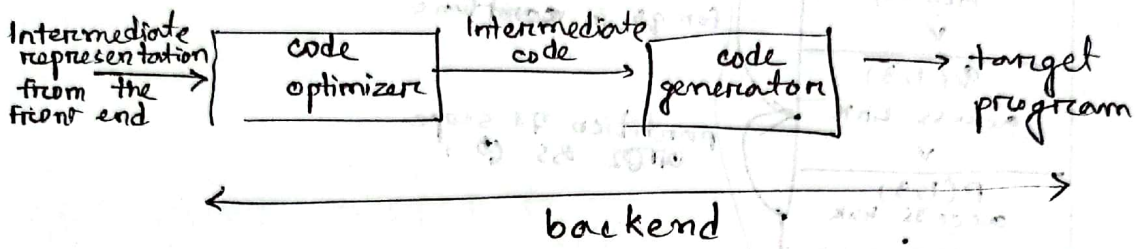
display

Chapter 8
CODE GENERATION

Front end → lexical, syntax, semantic

IR → Intermediate code

from code optimer to last → backend.



Inst. set architecture : 1) RISC
2) CISC
3) stack based.

25.9.24

① $x = y + z$

req → load.

```
LD   R0, y
LD   R1, z
ADD  R0, R0, R1
ST   x, R0
```

var → store

② $a = b + c$

$d = a + e$

```
LD   R0, b
LD   R1, c
ADD  R0, R0, R1
~~ST a~~, ST a, R0
          LD R0, a
LD   R2, e
ADD  R0, R0, R2
ST   d, R0
```

inefficient / redundant

naive translation → correct but unacceptably inefficient.

Scanned with CamScanner

A simple Target machine model :

* load, store, computation op, jump, conditional, label.
* R_0-$R(n-1)$ registers
* integers.

LOAD

LD dest, addrc.

dest → register, addrc → variable

LD r, x (mem to reg.)

LD r_1, r_2 (reg to reg.)

STORE

ST x, r → reg. to mem location

Operation

OP dest, src 1, src 2

OP → ADD or SUB

Uncond

BR L

BR → branch, L → label

cond

Bcond r, L → label

r → reg या fix value check करना fulfil होने पर L पर jump

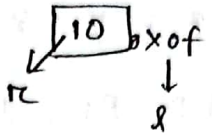
BLTZ r, L

r में value less than zero होने पर L पर jump

addressing modes :

l - value → location

r - " → real value.



* x → var

* a(r) → base + offset.
  - a → var
  - r → reg.

LD R1, a(R2) ← let R2 reg ला घेतले

(a म्हणजे location + R2 मध्ये location)

= (loc.) ← real value of that location

R1 ला घातले.

LD R1, 100(R2)

real - value (100 + R2 मध्ये mem location).

indirect add mode :

* * r → content of reg. r.

* 100(R2)

→ value (100 + (content)(R2))

constant add :

LD R1, # 100

100 असा R1 ला store

~~LD R1, 100~~ → location

Ex 8.2

x = y - z

```
LD  R1, y
LD  R2, z
SUB R1, R1, R2
ST  x , R1
```

b = a[i] ⟶ array start from 0
& byte value

```
LD  R1, i
MUL R1, R1, 8
LD  R2, a(R1)
ST  b , R2
```

a[j] = c

```
LD  R1, c
LD  R2, j
MUL R2, R2, 8
ST  a(R2), R1
```

x = *p

```
LD  R1, p
LD  R2, 0(R1)
ST  x , R2
```

Scanned with CamScanner

* p = y

```
LD  R1, p
LD  R2, y
ST  0(R1), R2
```

if x < y goto L

```
LD   R1, x
LD   R2, y
SUB  R1, R1, R2
BLTZ R1, M
```

↳ memory address where L label is written

Costs :

```
LD R0, R1
```

→ 1 वी inst. → Total cost = 1

→ reg वर cost = 0

```
LD R0, M
```

→ Total cost = 2

```
LD R1, 100(R2)
```

→ " " = 3

Scanned with CamScanner

Addresses in target code :

Static → global constant → compile time

dynamic → heap → cannot be determined in compile time

" → stack → " " " " " "

↓

for proc calls and return

Static allocation :

call callee

return

halt

action

call callee → ST. callee, static Area, #here + 20

↑ const

BR callee, code Area

↑ const

↓ two inst

5 words → 20 bytes.

return callee → BR * callee, static area

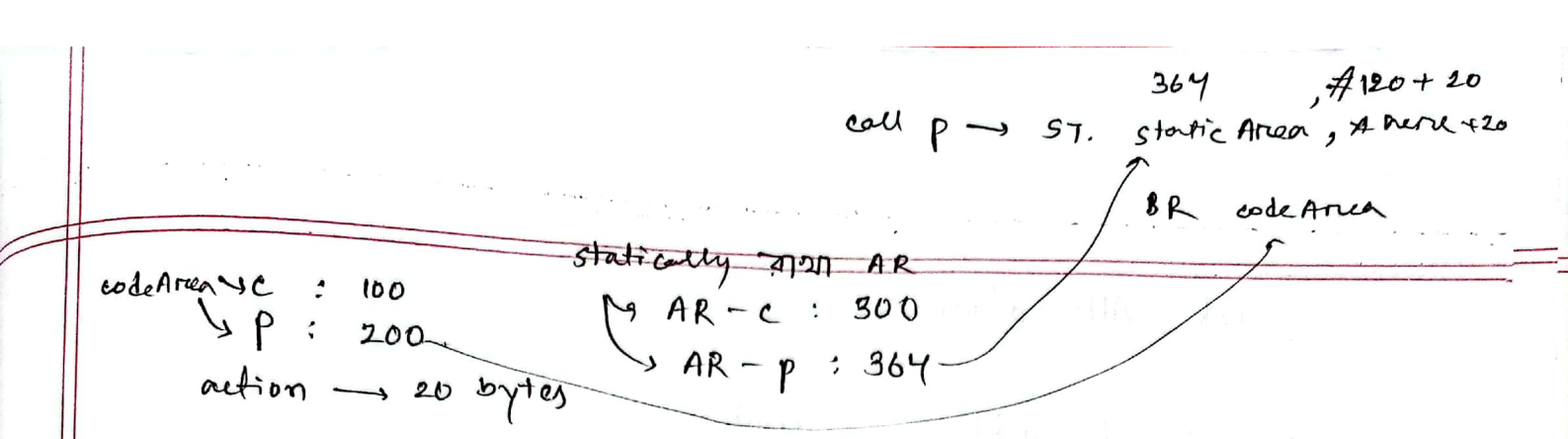
Ex 8.3 :

proc c — start → 100

" p — " → 200

each inst takes 20 bytes.

activation records start from 360 and 304 respectively



// code c

action 1 (20 bytes জায়গা নিবে)

call p

action 2

halt

100 : Action 1

120 : ST 364 , #140

3 × 4 = 12 byte

132 : BR 200

140 : ACTION 2

160 : Halt

code p

action 3

return

200 : Action 3

220 : BR * 364

return callee

↳ BR * callee. static area

100 : // c

200 : // p

300 : AR-C

364 : 140

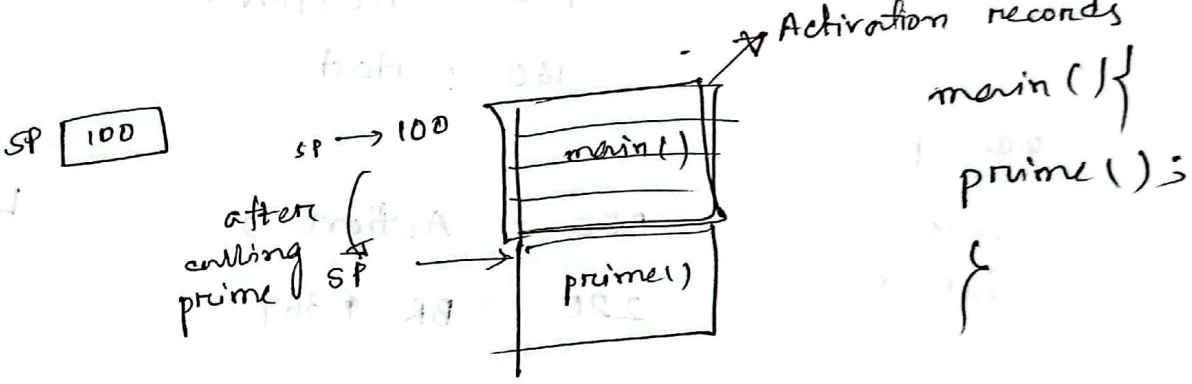
↑ memory

Scanned with CamScanner

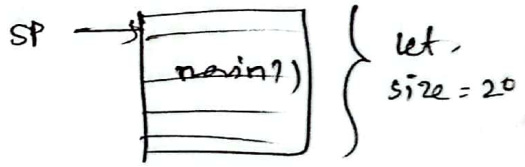
2.10.2024

Stack Allocation :

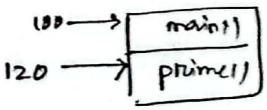
* indexed address mode
  → a(r) → (base add of a + r)
  100(r) → (100 + r)
* position using SP (stack pointer)
* proc calls → inc. SP → transfer to called proc.



after returning, dec. SP.



Step 1
• LD SP, # stack start
// code
HALT



Step 2
call callee → ADD SP, SP, # caller. record size
ST 0(SP), #here + 16 ← return address storee
BR callee. code Area // jump to callee

4 words: INCT + reg / addr / ind reg

step 3

return ⟶ BR * 0(SP)

SP | 120 |

0 + 120 = 120

ঐ location এ
যে value আছে
Jump.

step 4

SUB SP, SP, # caller. recordsize

```
main ( ) {
    QS ( )
}
QS ( ) {                    m = 20
                            P = 40
                            q = 60
    partition ( )
      QS ( )
      QS ( )
}
```

→ Activation record size :

msize = 20
psize = 40
qsize = 60

codeArea :

m 100
P 200
q 300

stack starts → 600

action x ⟶ ACTION ⟶ 20 bytes (let)

Example 8.4 :

// main. code Area

As stack address start 600 से ।

100 LD SP, #600

// partition. code Area

200

300

600

SP | 600 |

2word = 8 byte

Inst.	reg	

100 : LD SP, #600

20 byte ← 108 : ACTION 1

128 : ADD SP, SP, # 20 SP | 620 |

136 : ST 0(SP), #152 (# 136+16 = 152) → 620 : 152

144 : BR 300 (call q) → q का code area = 300)

152 :

// quick Sort code Area

300 : ACTION 4

call p. 320 : ADD SP, SP, # 60 ← q का size SP | 680 |

328 : ST 0(SP), # 344 (328 + 16 = 344) ⟶ 680 : 344

call p. 336 : BR 200

// partition code Area

200 : ACTION 3

return : 220 : BR *0(SP)

SP | 680 |

680 : 344

344 : SUB SP, SP, #60 ← q का size. (q caller) | 620 |

352 : ACTION 5.

Scanned with CamScanner

q3 AT size

372 : ADD SP, SP, #60. SP 680

380 : ST 0(SP), #396 680 : 396

call q : 388 BR 300

action 6 : 396 ~~ACTION6~~ SUB, SP, SP, #60 SP 620

call q 404 : ACTION 6

424 : ADD SP, SP, #60 SP 680

432 : ST 0(SP), #448. 680 : 448

448 : SUB, SP, SP, #60 SP 620

456 : BR *0(SP) 620 : 152.

// main

152 : SUB SP, SP, #20 SP 600

action 2 160 : ACTION 2

180 : HALT.

Basic blocks & Flow graphs :

First instruction first Basic block ला enter करतो.

8.5

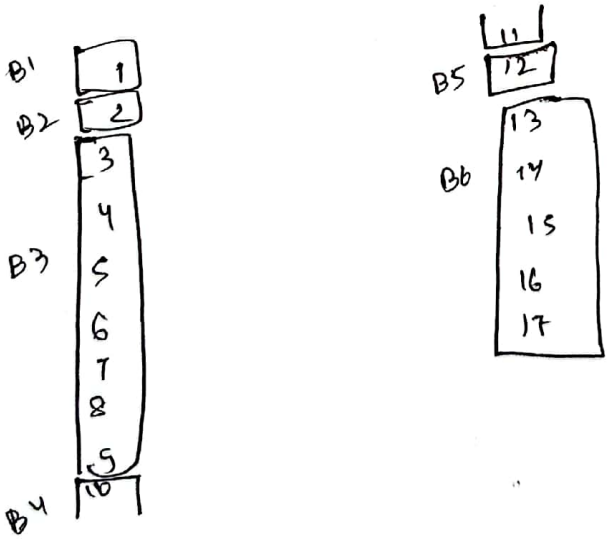
↳ INPUT : seq of 3-add instructions

OUTPUT :
each inst 1 ला basic block मध्ये part होतो.

Method :
leader → basic block चा first inst.

FIRST inst → leader
target of cond / uncond jump → leader
immediately followed by uncond / cond. jump → leader.

leaders → 1, 3 (target of cond. jump (9)), 2 (target (11)), 13 (target (17)), 10, 12, 8 (9, 11, 17 चे immediate inst).

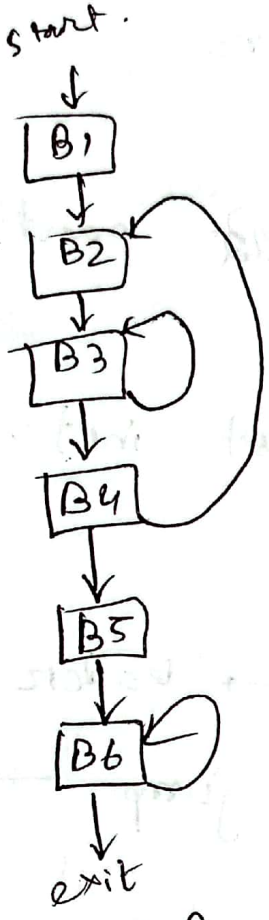


6 basic blocks.

Scanned with CamScanner

Flow Graph :

nodes ⟶ Basic blocks.



Basic block optimization