[Note:
↳ We need cannonical collection of LR(1) item]

CLR (1)

LALR(1)

* cannonical collection of LR (1) item = Cannonical Collection of LR(0) + lookahead

Ex:

$S \rightarrow AA$

$A \rightarrow aA \mid b$

Augmenting the grammar (& closure)

$S' \rightarrow .S, \$$ ← look-ahead (only in CCLR(1) item)

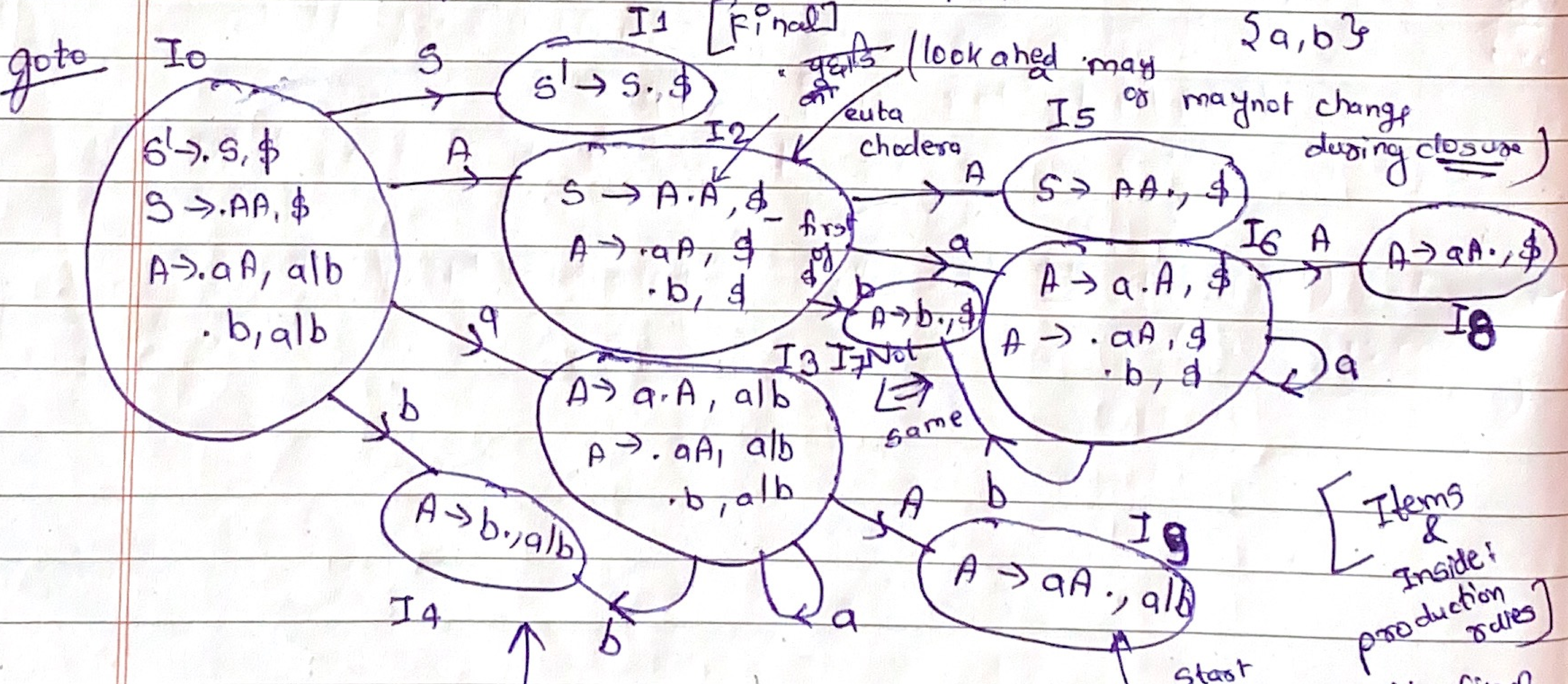
$I_0 \Rightarrow S \rightarrow .AA, \$$ → first of \$

$A \rightarrow .aA, a|b$ → first of A\$ ↓ {a,b}

$.b, a|b$

example: (\$) or (a|b)

[lookahead remains same in goto operation)



or for follow: || not necessarily follow we start reducing the final items

a|b   but look-ahead

[LR(0) → r_3 for all
SLR(1) → r_3 for FOLLOW
CLR(1) → r_3 for Lookahead]

Date
Page

S → AA (r_1)

A → aA | b
(r_2) (r_3)

Parsing Table → Action Part
→ Goto Part

[CLR(1)]

	Action			Goto	
	a	b	$	S	A
0	S_{36}	S_4		1	2
1			Accept		
2	S_6 ; S_{36}	S_7			5
3	S_{36}	S_4			9
4	[r_3]	[r_3]			
5			r_1		
6	S_6 ; S_{36}	S_7			8
7			r_3 // As A → b.(r_3) on $		
8			r_2		
9	r_2	r_2			

A → b. (r_3) for follow - a/b

No difference

↳ Those states Same [other than lookahead]

move S_3 or S_6 to S_{36} [New state]

I_3 I_6 ⇒ I_{36}

I_4 I_7 ⇒ I_{47}

I_8 I_9 ⇒ I_{89}

⇓ (merging) // Union Operation on Items ⇒ [LALR(1)]

[Lookahead ma मात्र than] Asked in # Final Exam.

New LALR TABLE:

	Action			Goto	
	a	b	$	S	A
0	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
~~36~~	~~S_{36}~~	S_{47}			89
~~47~~	~~γ_3~~	~~γ_3~~	~~γ_2 (Added)~~		
5			γ_1		
~~36~~	~~S_{36}~~	~~S_{47}~~			~~89~~
~~47~~			~~γ_3~~		
~~89~~			~~γ_2~~		
89	γ_2	γ_2	(γ_2 Added)		

If there is conflict on Parsing Table ; like γ_2 / γ_3

Then it is not LALR(1)

No. of states on CLR(1) $\geq$ LALR(1)

(Union so; can be less)

Algorithm for Constructing SLR Parsing Table

Date
Page

1) Given the grammar G, construct an <u>augmented</u> grammar G' by introducing a production of the form:
   $S' \rightarrow S$, where S is the start symbol of G.

2) Let I_0 = closure ($\{[S' \rightarrow S]\}$). Starting from I_0, construct SLR <u>DFA</u> using <u>closure</u> and <u>goto</u>.

3) <u>State 'i'</u> is constructed from <u>I_i</u>. The parsing actions for state i are defined as follows:

(*) ⟸ a) If goto $[I_i, A] = I_j$, set action $[i, a] = shift\ j$,
   here 'a' must be a terminal. (can be terminals or non-terminals)

   b) If I_i has the production of the form $A \rightarrow \alpha$, then for all symbols a in FOLLOW(α), set action $[i, a]$ = reduce $\underbrace{A \rightarrow \alpha}_{\text{reduction no. 1/2/3 from Prod. Rule.}}$. Here, A should not be S'.

   c) If $S' \rightarrow S.$ is in I_i, then set action $[i, \$]$ = accept.

4) For all blank entries, set entries to <u>"error"</u>.

5) The start state S_0 corresponding to I_0 = closure($\{[S' \rightarrow S]\}$)

Note :- The set of all items $\{I_0, I_1, I_2, \ldots, I_n\}$ computed for the grammar is called set of LR(0) items.
// or, cannonical collection of LR(0) items.

(*) ⇒ If goto $[I_i, A] = I_j$, where 'A' is a non-terminal, then goto $[i, A] = j$

ex: $[0, A] \rightarrow$ 1 state मा जायो।

Constructing Canonical LR(1) Parsing Table
[CLR(1)]

1) Given the grammar G, construct an augmented grammar G' by introducing a production of the form $S' \to S$, where S is the start symbol of G.

2) Construct the set $C = \{ I_0, I_1, \ldots, I_n \}$ of LR(1) items.

3) If $[A \to \alpha . a\beta, \underline{b}]$ ∈ I_i and goto $(I_i, a) = I_j$ then set action $[i, a] = shift_j$.
(b ← lookahead)

4) If $[A \to \alpha ., b] \in I_i$ then set action $[i, b]$ = reduce $A \to \alpha$ (apply only if $A \neq S'$)

5) If $[S' \to S., \$]$ is in I_i then set action $[i, \$]$ = accept.
↳ look-ahead for 'S' is $.

6) If goto $(I_i, A) = I_j$ then set go to $[i, A] = j$ (ex: 2 on A goto → 3)
= Only number just a no. for goto.

7) Repeat 3 to 6 until no more entries added.

8) The initial state I is the I_i holding item $[S' \to S_0, \$]$

SLR(1)
SLA → // Lookahead symbol

Union Operation on Item // [for LALR(1)]

Given an item of the form $[A \to \alpha . B\beta, a]$, the first part (before comma) is called the "core" of the item.

Given two states of the form $I_i^\circ = \{[A \to \alpha ., a]\}$ and $I_j^\circ = \{[A \to \alpha ., b]\}$ the union of the states $I_{ij}^\circ = I_i^\circ \cup I_j^\circ = \{[A \to \alpha ., a/b]\}$.

(when "core" is same)

(State I_{gg}) g g

↳ If there is conflict on Parsing Table; then it is not LALR(1).

Chapter

4 Syntax Directed Translation

Date
Page

[Semantic Analyzer]
(Meaning - ko kaam garxa ta)
ex: type checking

C.F.G + Semantic Action = S.D.T
(Context-Free Grammar) (Syntax Directed Translation)

⇒ We associate information with a programming language construct by attaching attributes to the grammar symbol. The values for attributes are computed by Semantic Rules associated with the grammar production. The way of associating attributes to each of the symbol in the grammar that provides us the order of translation is called S.D.T.

Evaluation of these semantic rules may:

1) Generate intermediate code

2) Put information into the symbol table

3) Perform type-checking

4) Issue Error Message (can be in every phase of compiler)

Symbol Table : information about keywords & token..

cond... A syntax directed definition bind a set of semantic rules to productions. They are the high-level specifications for translation schemes. It is the generalization

of parse-tree where each terminal and non-terminal(s) has attributes associated to it whose values are determined by semantic rules. In other words, a Syntax Directed Definition (S.D.D) is a generalization of a Context-Free Gramar in which:

→ Each grammar symbol is associated with a set of attributes
→ This set of attributes for a grammar symbol has 2 subsets called synthesized and inherited attributes.
→ Each production rules is associated with a set of semantic rules.

Semantic Rules set up dependencies between attributes, which can be represented by a Dependency Graph. This Dependency Graph determines the Evaluation Order of these Semantic Rules.

If the value of attribute only depends upon its children, then it is called synthesized attribute. The syntax-directed definition with only synthesized attribute is called S-attributed definition or S-attributed Grammar.

If the value of attribute depends upon its parent or sibling, then it is called inherited attribute. (फलानोंको होवे फलानोंका भाई)

A parse tree showing the value of attributes at each node is called annotated parse-tree. The process of computing the attribute values at the nodes is

called annotating (or decorating) of the parse-tree.

Suppose a grammar:

$E \to E + T \quad \{ E.val = E.val + T.val \}$ // Grammar + Semantic Action

Semantic-action associated with this production Rule.

$= g, D, T$

$| T \quad \{ E.val = T.val \}$

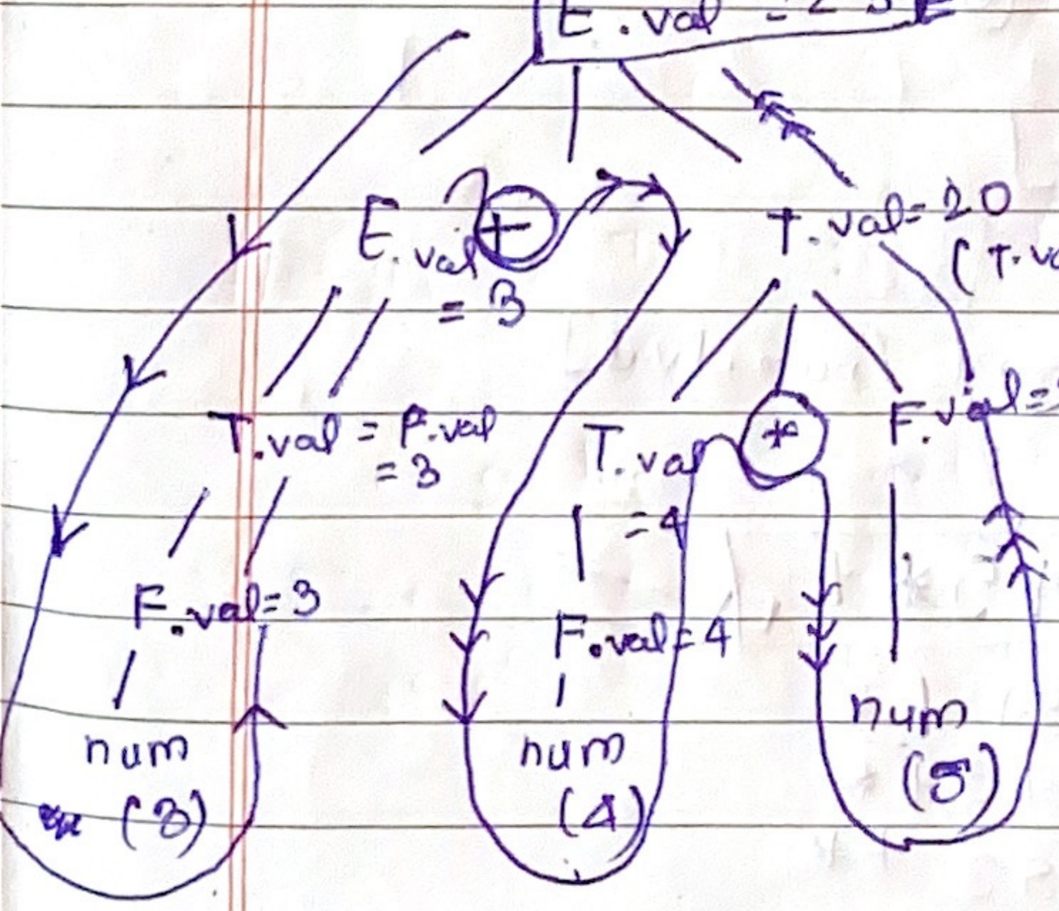
$T \to T * F \quad \{ T.val = T.val * F.val \}$

$| F \quad \{ T.val = F.val \}$

$F \to num \quad \{ F.val = num.lexval \}$

3 + 4 * 5

num + num * num



Top - to - Bottom

Left - to - Right

annotating (computing values at the nodes) [just like a graph]

Date
Page

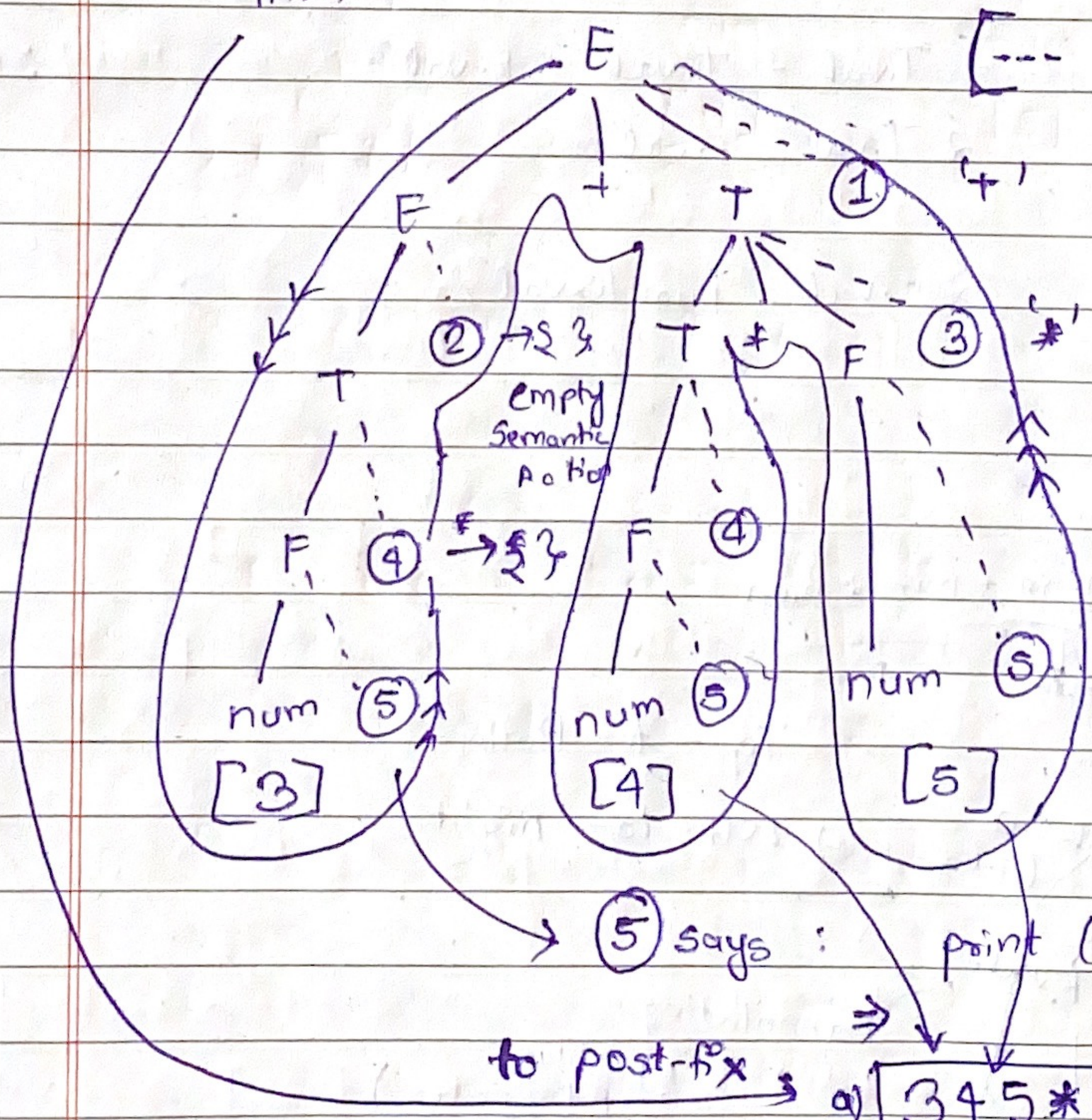
Suppose a grammar:

	Semantic Action:	
$E \rightarrow E + T$	{ printf("+"); }	(1)
$\mid T$	{ }	(2)
$T \rightarrow T * F$	{ printf("*"); }	(3)
$\mid F$	{ }	(4)
$F \rightarrow$ num	{ printf(num.lval); }	(5)

[lexical value]

3 + 4 * 5

Infix ⇒



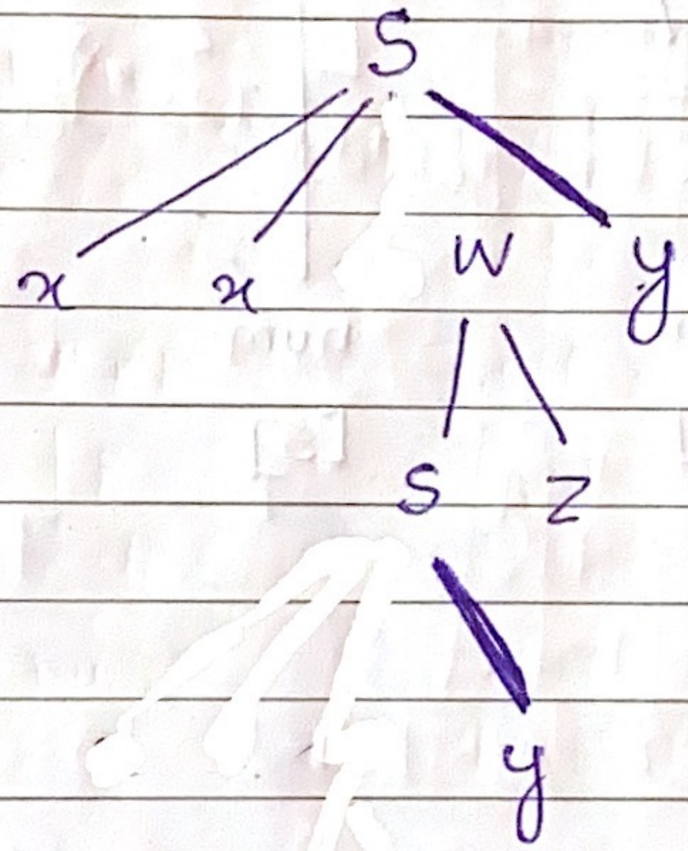
[--- indicates Semantic Action numbers]

$S \rightarrow xx\,W$ ① { printf('1'); }

$\quad\quad | \; y$ ② { printf('2'); }

$W \rightarrow Sz$ ③ { printf('3'); }

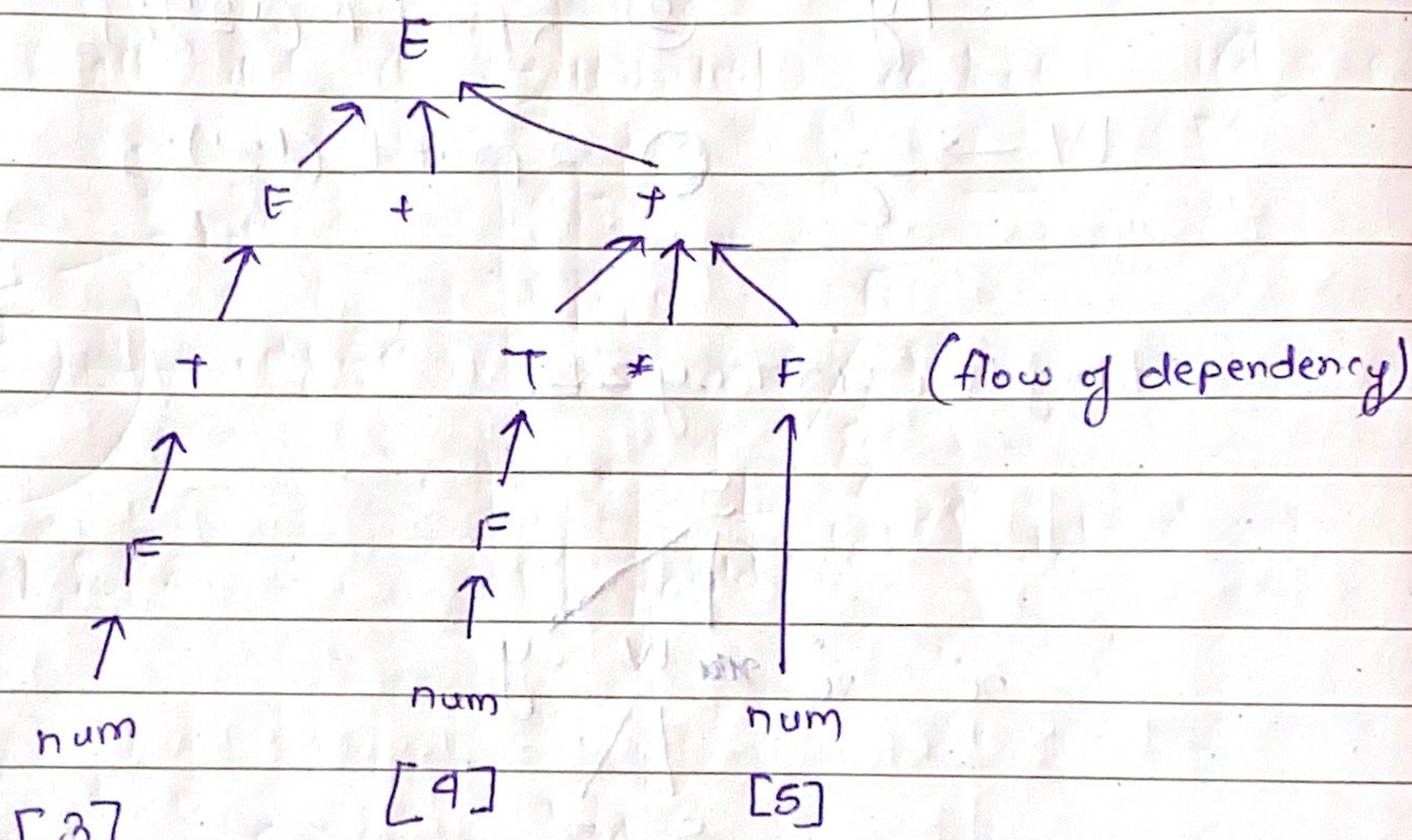
String is xxxxyzz



Dependency Graph :

↳ In order to correctly evaluate the attribute of syntax-tree node, a dependency graph is useful. A dependency graph is a directed graph that consists

attributes as node and dependencies across attributes as edge.



Difference Between :

Date
Page

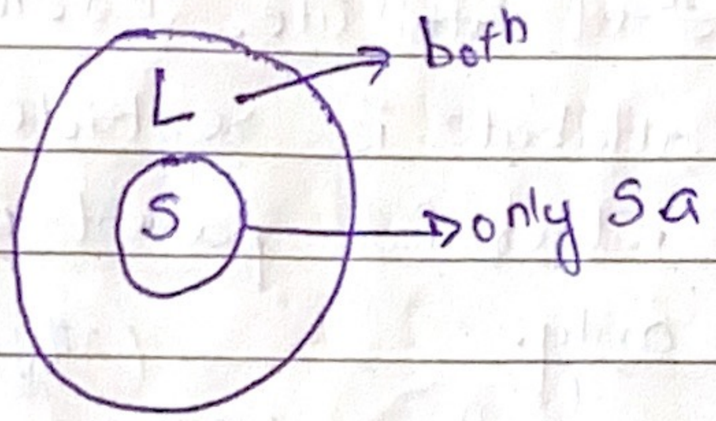
S-attributed SDT	L-attributed SDT
1) Uses only synthesized attribute.	1) Uses both inherited and synthesized attribute. Each inherited attribute is restricted to inherit either from parent or left-sibling only. (attributes can be derived from L.H.S; not R.H.S) e.g.: $A \longrightarrow BCD$ $\{C.S = A.S, C.S = B.S, C.S = D.S\}$ ✓ ✓ ✗ (not possible) C ko attribute vaneko B ko attribute hunxa
2) Semantic action are placed at right-end of production $A \rightarrow BcD\{\}$	2) Semantic actions are placed anywhere on the R.H.S. e.g. $A \rightarrow \{\}BcD$ $B\{\}cD$ $Bc\{\}D$ $BcD\{\}$ ← curly bracket माझ लेख्या जती (Grammar + Rule भिजो लेखेको)
3) Attributes are evaluated during bottom-up parsing	3) Attributes are evaluated by traversing parse-tree depth-first; left-right.

Date
Page

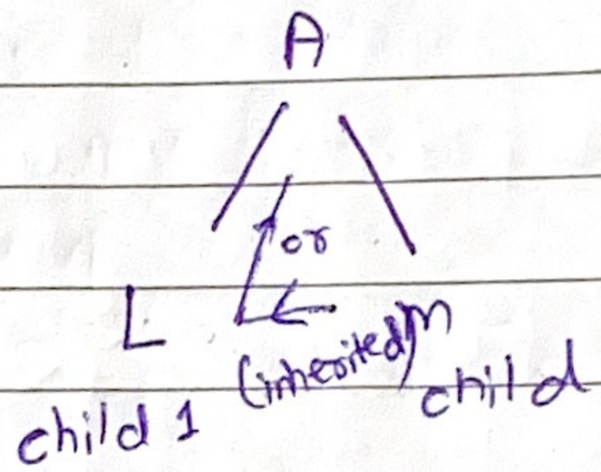
It is Inherited (to the left)
not synthesized

1> $A \rightarrow L\,M$ { $L.i = f(A.i)$; $M.i = f(L.s)$; $A.s = f(m.s)$; }

(a) S-attributed (b) L-attributed (c) both (d) none



Every L is S



2> $A \rightarrow BC$ { $B.s = A.s$ }

(a) S-attributed (b) L-attributed (c) both (d) none

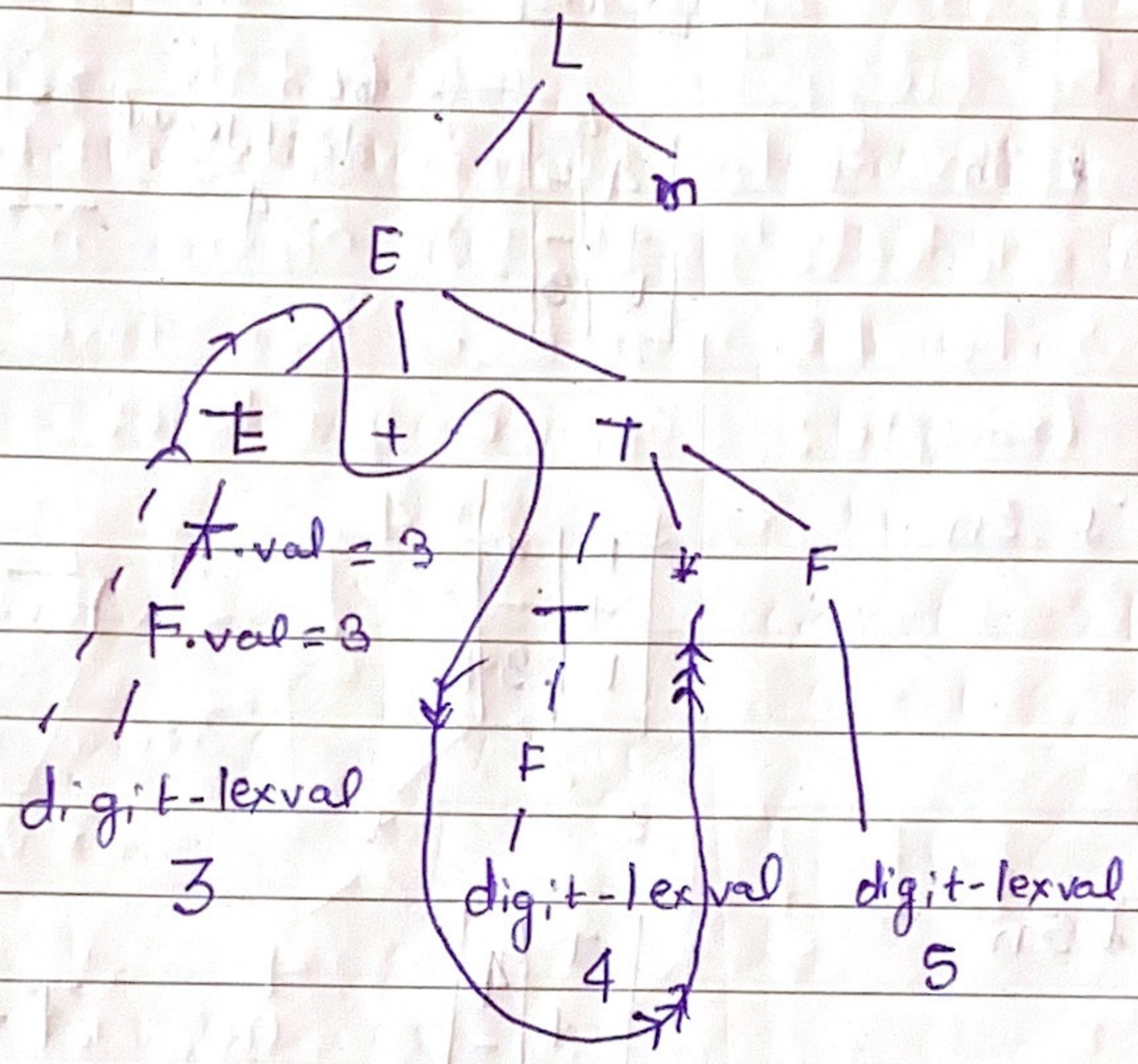
3> $A \rightarrow QR$ { $R.i = f(A.i)$; $Q.i = f(R.i)$; $A.s = f(Q.s)$; }

right sibling
(none)

(a) S-attributed (b) L-attributed (c) both (d) none

Imp#

Bottom-up evaluation of S-attributed definition (stack implementation)



$L \rightarrow En$

$E \rightarrow E + T$

T

$T \rightarrow T * F$

F

$F \rightarrow digit$

Input	Stack	Production
3 + 4 * 5 n	—	—
+ 4 * 5 n	3 | —	—
+ 4 * 5 n	F | 3	F → digit
+ 4 * 5 n	T | 3	T → F
+ 4 * 5 n	E | 3	E → T

Date
Page

4 * 5n

+	-
E	3

—

* 5n

4	-
+	-
E	3

from digit. leaval = 4 —

* 5n

F	4
+	-
E	3

F → digit

* 5n

T	4
+	-
E	3

T → F

5n

*	-
T	4
+	-
E	3

—

5 n

5	-
*	-
T	4
+	-
E	3

Date
Page

L n

F	5
*	-
T	4
+	-
E	3

F → digit

n

T	2
*	-
*	*
+	-
E	3

T → T. digit * F. digit
(T * F)

n

E	23

E → F + T

—

n	-
E	23

—

L	23 n

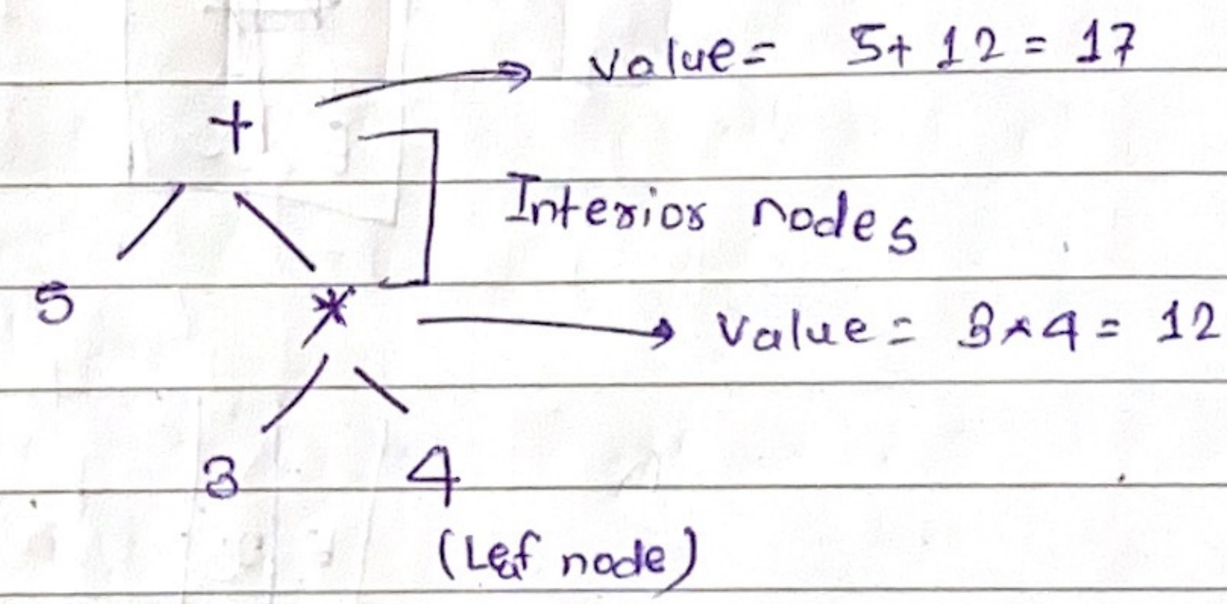
L → E x n

Construction of Syntax Tree

The syntax - directed definition can be used to specify the construction of syntax-tree and other graphical representation. The syntax tree is an intermediate representation of an expression. In syntax tree, operators and keywords do not appear as leaves and they appear in the interior nodes that will be the parent of those leaves in the parse-tree.

Example: Suppose you have:

5 + 3 * 4



The syntax directed definition to create a parse tree for the expression must execute the implementation of the following function:

1) make node (op, left, right) :- to create an operator node with label op and two field containing pointer left and right.

2) make leaf (id, entry) :- create an identifier node with label id and a field containing entry; a pointer to the symbol table entry for that id.

3) make leaf (num, val) : create a node with

label num and a field containing val; the value of the number.

Consider the following grammar with associated semantic rules.

Production Rules:	Attributes:
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow E_1 - T$	$E.val = E_1.val - T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow (E)$	$T.val = E.val$
$T \rightarrow id$	$T.val = id.entry$
$T \rightarrow num$	$T.val = num.val$

The construction of syntax-tree using these rules will be as follows:

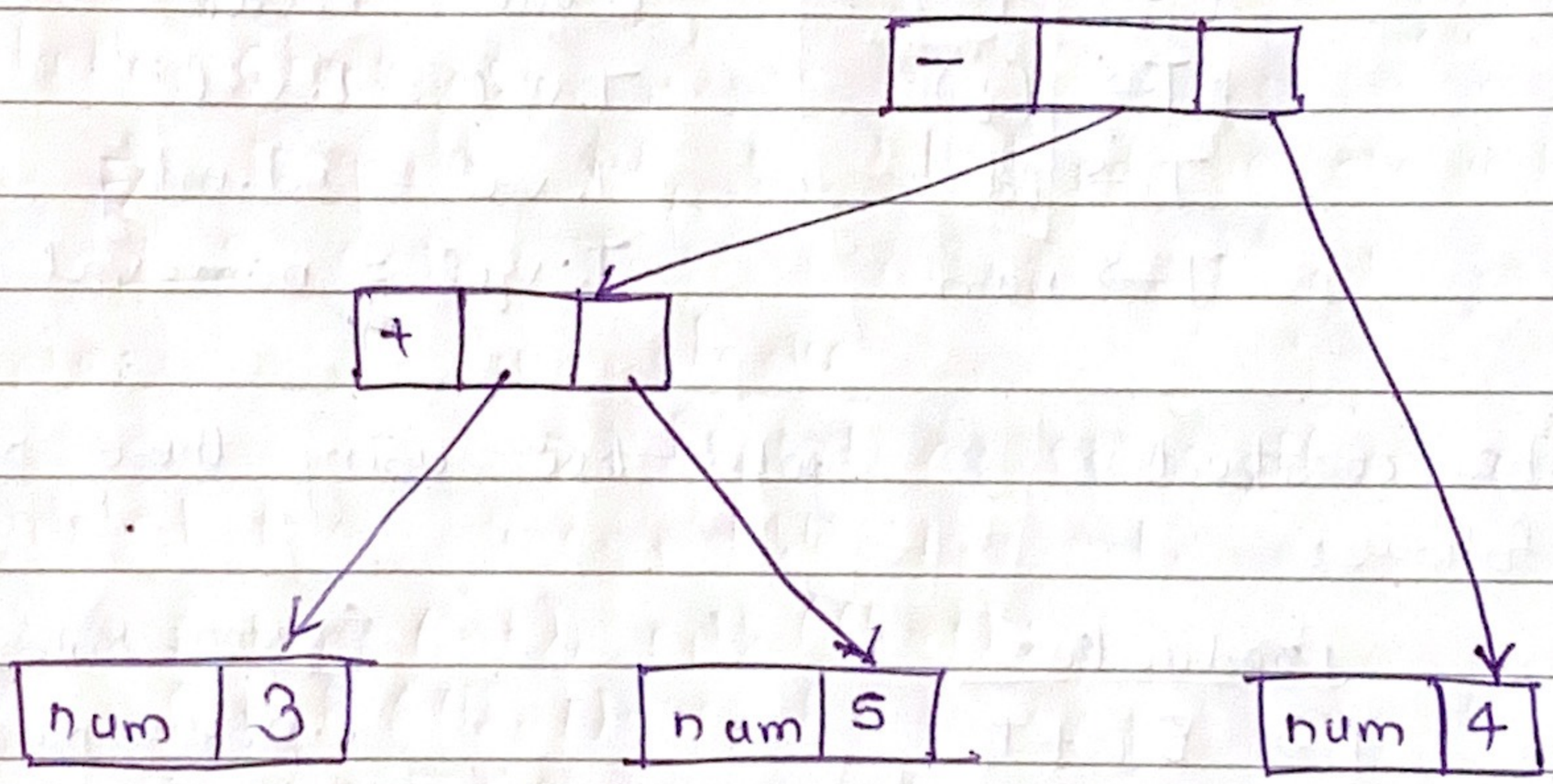
Production	Semantic Rules
$E \rightarrow E_1 + T$	$E.ptr = makenode('+', E_1.ptr, T.ptr)$ (from 1) makenode (op, left, right))
$E \rightarrow E_1 - T$	$E.ptr = makenode('-', E_1.ptr, T.ptr)$
$E \rightarrow T$	$E.ptr = T.ptr$
$T \rightarrow (E)$	$T.ptr = E.ptr$
$T \rightarrow id$	$T.ptr = makeleaf(id, entry)$
$T \rightarrow num$	$T.ptr = makeleaf(num, num.val)$

So, for the expression: 3+5-4 sequence of execution for the creation of syntax tree is:

Date
Page

3 + 5 - 4

1) p_1 = makeleaf (num, 3)
2) p_2 = makeleaf (num, 5)
3) p_3 = makeleaf (num, 4)
4) p_4 = makenode ('+', p_1, p_2)
5) p_5 = makenode ('-', p_4, p_3)



Directed Acyclic Graph (D.A.G)

⇒ A directed acyclic graph (DAG) for an expression identifies the common sub-expression in the expression.

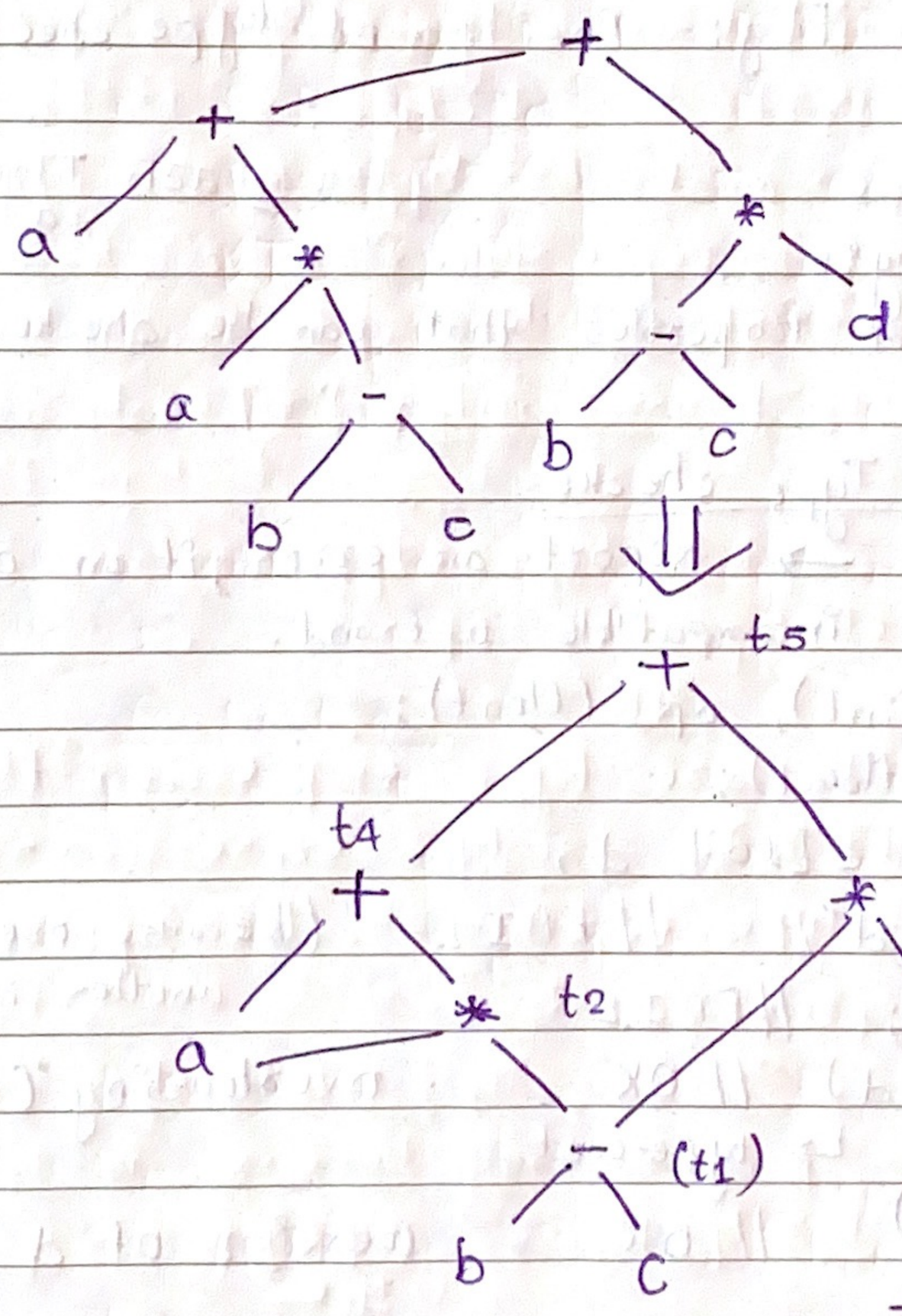
The only difference between syntax tree and DAG is that a node representing common sub expression has more than one parent in the syntax-tree.

Date
Page

Example:

Construct Syntax-tree and D.A.G for the following expression.

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



[D.A.G]

(t1 xa vaney tei use garera)

Three-Address Code

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

$t_1 = b - c$ $\quad t_3 = t_1 * d$ $\quad t_5 = t_4 + t_3$

$t_2 = a * t_1$ $\quad t_4 = a + t_2$

b t1 c (यसरी हेरेर बनानु पर्यो आपी को D.A.G जस्तो)

Type - Checking :-

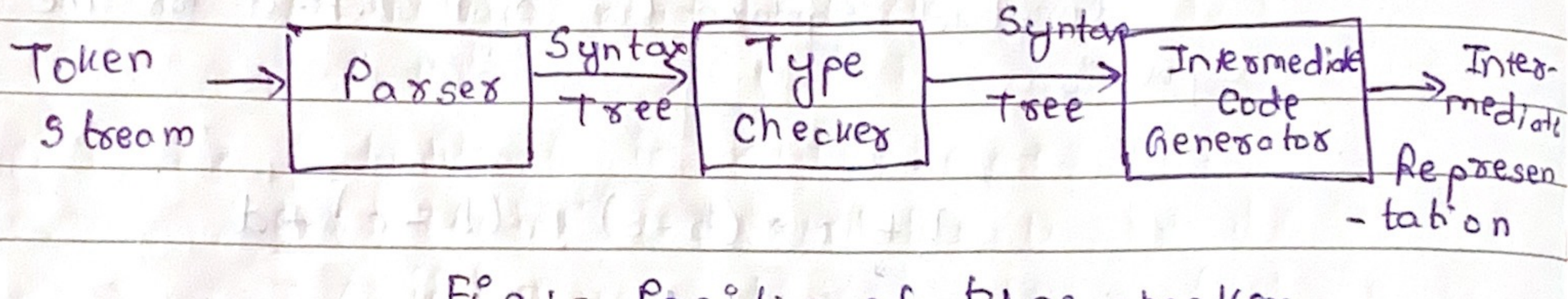


Fig:- Position of type checker.

2 act Time
- Compile-Time
- Run-Time

Static Checking:-

⇒ Program Properties that can be checked at compile-time.

Example: 1> Type checks

→ Report an error; if an operator is applied to an incompatible operand.

```
int op(int), op (float);
int f (float);
int a, c[10], d;
d = c+d;   // FAIL (because one is array-c[10] another is integer-d)
* d = a;   // FAIL
a = op(d)  // OK   overloading (c++)
        ↳ type-cast
a = f(d)   // OK   coersion of d to float
```

a=+a

2> Flow of control checks

=>

```
myfunc (int a)
{
    cout << a;
    break;  // Error (tettikai break gareko vayera)
}
```

Date _____
Page _____

```
myfunc()
{ ...
  switch (a) ~
  {
    case 0:
      ...
      break; //ok
    case 1:
      ...
      ...
  }
```

```
myfunc()
{
  ...
  if (i>10)
    break; //ok
  ...
  ...
}
```

3> Uniqueness Checks

=>
```
myfunc()
{
  int i, j, i; //ERROR
  ...
}
```

4> Name-related checks

```
my func()
{
  int i, j;
  ...
  ...
}
myfunct1()  // error
```

basic type → primitive type

Date
Page

jasko type agadi declare vaisakyaxa.

Type Expression

⇒ 1) A basic-type is a type expression:
Ex: integer, char, real etc

2) A type name is a type expression:
Ex: If 'int' is named by a variable 'x', then x is a ~~var~~ type expression.

3) A type constructor applied to a type expression is a type expression. The constructor include: array, product, pointer, function or record
Ex: T array of type [T] or array [I, T] is
size type
a type expression with I elements of type T.

⇓

→ If T_1 and T_2 are type expression then cartesian product $T_1 \times T_2$ is type expression - product.
→ If T is a type expression, then pointer (T) is type expression.

→ Function in programming language is a mapping of domain type D to a range [R].
type
Ex. Example:
~~int~~ int x int → pointer (char) denotes a func^n that takes a pair of integers and returns a pointer to character.
→ A record is a structure type

Date
Page

* Type checking of an expression

E → id { E. type = lookup (id. entry) }
E → literal { E. type = char }
E → int literal { E. type = int }
E → real literal { E. type = real }
E → E1 + E2 { if (E1. type = int and E2. type = int)
then E. type = int
else if (E1. type = int and E2. type = real)
then E. type = real
else if (E1. type = real and E2. type = int)
then E. type = real
else if (E1. type = real and E2. type = real)
then E. type = real
else E. type = type_error }

E → E1 [E2] { if (E2. type = int and E1. type = array (s, t)
↓
Array of size s
and type t.
then E. type = t
else
E. type = type_error }

E → * E1 { if (E1. type = pointer (t) then E. type = t
else E. type = type_error }

int a[50]
↳ a(50, int)
(same)
size

A simple language type checking specification

$E \rightarrow$ true	{ E.type = boolean }
$E \rightarrow$ false	{ E.type = boolean }
$E \rightarrow$ literal	{ E.type = char }
$E \rightarrow$ num	{ E.type = integer }
$E \rightarrow$ id	{ E.type = lookup (id.entry) }
$E \rightarrow E_1 + E_2$	{ E.type := if E_1.type = integer and E_2.type = integer then integer else type-error }
$E \rightarrow E_1 + E_2$	{ E.type := if E_1.type = boolean and E_2.type = boolean then boolean else type-error } (2 वटा same huna paryo)

Eg.

$T \rightarrow$ int	{ T.type = int }
$T \rightarrow$ char	{ T.type = char }
$T \rightarrow$ real	{ T.type = real }
$T \rightarrow$ array [intnum, T_1]	{ T.type = array [intnum.val, T_1.type] }
$T \rightarrow$ pointer (T_1)	{ T.type = pointer (T_1.type) }

(Exam मा Sodexa)

V.V. Imp!!

Date
Page

Q. Type-checking expression of an array of pointer to real, where array index ranges from 1 to 100.

$T \rightarrow$ real { T.type = real }

$E \rightarrow$ array [100, T] { if T.type = real
then
E.type = array (1 ... 100, T)
else type-error ( ) }

$E \rightarrow$ pointer [E1]

{ if (E1.type = array(100, real))
then E.type = E1.type
else E.type = type-error }

Type checking expression of a statement

Assignment statement:

(cuz there's no return type here)

$S \rightarrow$ id = E { if (id.type = E.type) then S.type = void
else, S.type = type-error }

If then else statement

$S \rightarrow$ if E then S_1 { if (E.type = boolean) then
S.type = S1.type else S.type
= type-error }

Date ________
Page ________

while statement

S → while E do S_1 { if (E.type = boolean) then S.type = S_1.type }

Type Conversion and Coercion

implicit – done by compiler (Python)

⇒ Type Conversion is explicit (defined by the programmer / typecast) (C++ Ex: integer define)

⇒ Type Coercion is implicitly performed by the compiler.

⊙ Equivalence of the type expression:

(2 ठटा expression kati khera equal vanda kheri parameter eutai vayo vaney:
→ Argument ko type
add (int, float)
sum (int, float))

Symbol Table.

(Interact At every phase of Compiler)

Date
Page

Symbol Table :

↳ Compiler creates and maintains symbol table to keep detail about the following:

=> variables, names, function names, objects, classes, interfaces, etc.

1 ⇒ Symbol Table is used for analysis and synthesis.

2 ⇒ It helps to determine scope resolution.

3 => It helps to determine whether a variable is defined already or not.

4 ⇒ To add a new name to the table

5 => To access information associated with a given name

6 ⇒ To add new information with a given name

7 ⇒ To delete a name or group of names from the table

8 ⇒ Type-checking for semantic correctness determination.

Data Structures used fo

Date
Page

Data Structure used for implementation of Symbol Table:

1. Binary Search Tree
2. Hash Table
3. Linear List

Examples:

```
int var1;
int procA()
{
    int var2, var3;
    ...
    ...
    {
        int var4, var5;
        ...
        ...
    }
    int var6;
    ...
    ...
    {
        int var7, var8;
        ...
        ...
    }
}
int procB()
{
    int var9, var10;
    ...
    ...
```

Date
Page

```
{
    int var 11, var 12;
    ...
}
int var 13;
}
```

Symbol table (Global)

name	type	
var 1	var	int
procA	proc	int
proc B	proc	int

Symbol Table:
procA

var 2	var	int
var 3	var	int
var 6	var	int

Symbol Table: procB

var 9	var	int
var 10	var	int
var 13	var	int

var 4	var	int
var 5	var	int

Inner Scope-1
Symbol Table

var 7	var	int
var 8	var	int

Inner Scope-2
Symbol Table

var 11	var	int
var 12	var	int

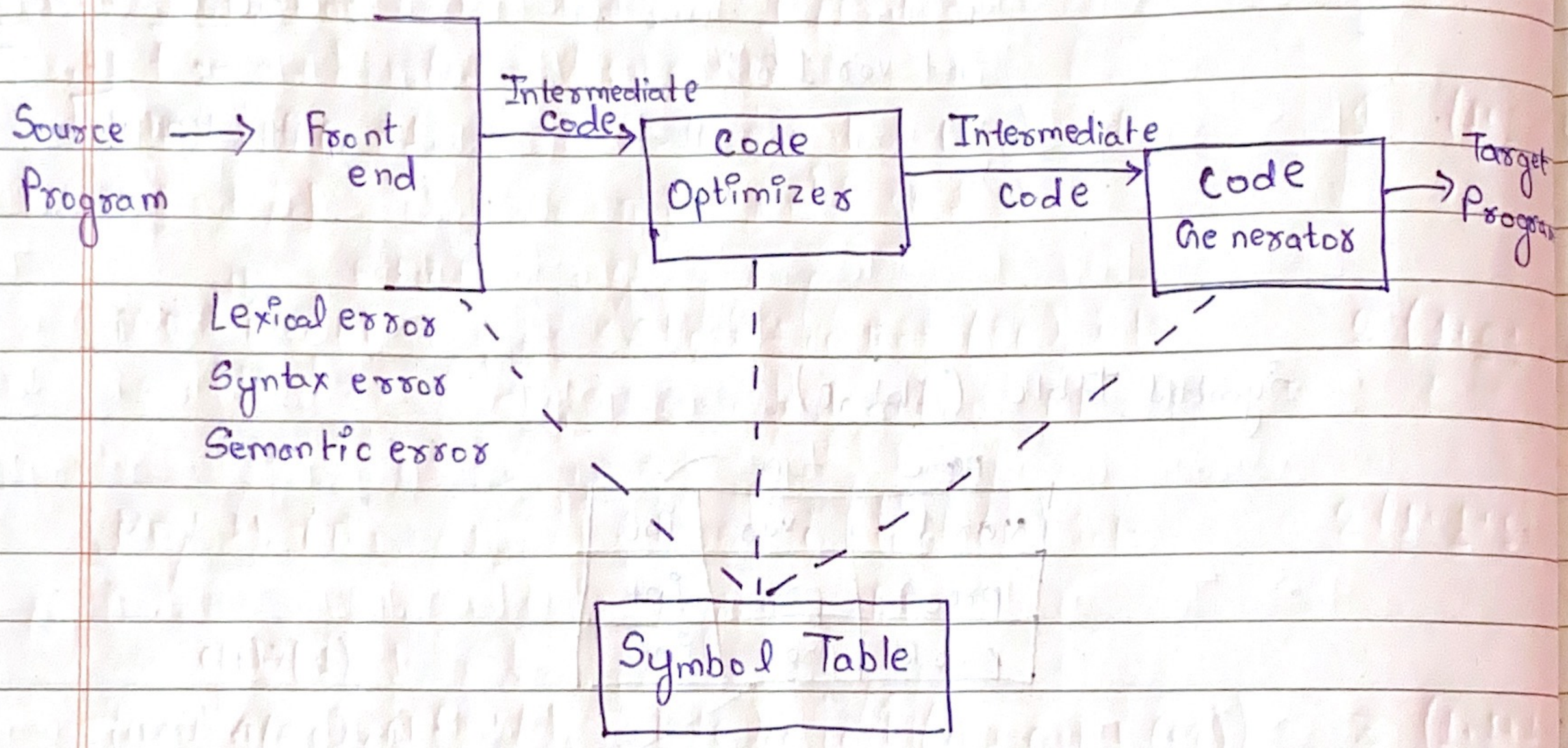
Inner Scope-3
Symbol Table

Date
Page

Chapter - Last

Code Generation and Optimization

(milne samma Chalaune)



How the target codes are generated optimally from an intermediate form of programming language?

Code Generator Design Issue (Intermediate Code lai Target code ma change garne kaam)

1. Input to the code generator

↳ The input to the code generator is intermediate representation together with the information in the symbol table.

What type of input? → postfix, three-address code, DAG or tree.

Date
Page

2. Target Program (Machine Dependent)
↳ which one is the output of code generator ?

3. Target Machine

4. Instruction Selection (Low cost logne instruction select gorna paryo)

5. Register Allocation

6. Choice of Evaluation Order

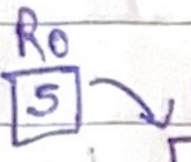
The Target Machine :

Addressing Mode :

Mode	From	(To) 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	N/A	1

Date
Page

Cost of instruction = 1 + cost (source-mode) + Cost (destination-mode)



Instruction	Operation	Cost
MOV R0, R1	Store content (R0) into register R1	1 (1+0+0)
MOV R0, M	Store content (R0) into memory location M	2 (1+0+1)
MOV M, R0	Store content (M) into register R0	2 (1+1+0)
C(R)→(1) MOV 4(R0), M	Store content (4 + Content (R0)) into M	3 (1+1+1)
*C(R)→(1) MOV *4 (R0), M	Store contents (contents (4+ Contents (R0))) into M	3 (1+1+1) (M)
#C→(1) MOV #1, R0	Store 1 into R0	2 (1+1+0) (#C)
ADD 4(R0), *12(R1)	Add contents (4+contents (R0)) to value location contents (12 + contents (R1))	3 (1+1+1) (C(R), *C(R))

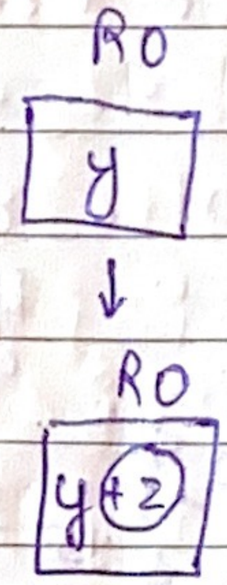
Date
Page

Instruction Selection:

↳ Instruction Selection is important to obtain efficient code.

Suppose, we translate 3 address code : $x = y + z$ to:

```
Mov y, R0
ADD z, R0
Mov R0, x
```



$a = a + 1$

```
mov a, R0   ⇒ 2
ADD #1, R0  ⇒ 2
mov R0, a   ⇒ 2
```

Cost : ⟶ 6

```
ADD #1, a        a → (1+a)      INC a        (a+1)
Cost = 3                        Cost = 2
```

Register Allocation and Assignment

↳ Register Allocation denotes the selection of which variables will go into register.

↳ Register Assignment is a determination of exactly which register to place a given variable.

x, y, z (variables)

R. Alloc → y → R. Assig → R0, R1, R2 (registers)

⊙ Example:

t = a * b (also t := a * b)

t = t + a

t = t / d

⇓ R1 = t

a = 3, b = 5, d = 6

MOV a, R1 ⇒ 3

MUL b, R1 ⇒ 15

ADD a, R1 ⇒ 18

DIV d, R1 ⇒ 3 (R1/d → then save in: R1)

mov R1, t

R1: 3 × 5 → ↓ → R1: 15 + 3 (15 + a)

R1: 18 / 6

Choice of Evaluation Order:

t1 ... t2

a + b - (c + d) * e

⇓ ... t3

t1 = a + b

t2 = c + d

t3 = e * t2

t4 = t1 - t3

re-order ↗ ④

MOV a, R0

ADD b, R0

mov R0, t1

mov c, R1

ADD d, R1

mov e, R0/R3

Date
Page

```
MUL   R1, R0
MOV   t1, R1
SUB   R0, R1
MOV   R1, t4
```

} 10

(*) $t_2 = c + d$

$t_3 = e * t_2$

$t_1 = e + b$

$t_4 = t_1 - t_3$

⇒ 8

```
MOV   c, R0
ADD   d, R0
MOV   e, R1
MUL   R0, R1
MOV   a, R0
ADD   b, R0
SUB   R1, R0      (R0 - R1 → store in R0)
MOV   R0, t4
```

Basic Block Construction Algorithm

Input:- A sequence of three-address statements

Output:- A list of basic blocks with each three-address statement in exactly one block.

1) Indentify leader first
2) First line of code is leader
3) Address of conditional, unconditional goto are leade
4) Immediate next line of goto are leader
5) Make basic block from leader to line before next leader

V.V.Imp

Address of (goto) ... → Leader

1) $i = 1$ — **B1**

2) $j = 1$ — **B2**

3) $t_1 = 10 * i$
4) $t_2 = t_1 + j$
5) $t_3 = 8 * t_3$
6) $t_4 = t_3 - 88$
7) $a[t_4] = 0.0$ (float value)
8) $j = j + 1$
9) if $j <= 10$ goto (3)

— **B3**

(immediate line next to goto) →

10) $i = i + 1$
11) if $i <= 10$ goto (2)

— **B4**

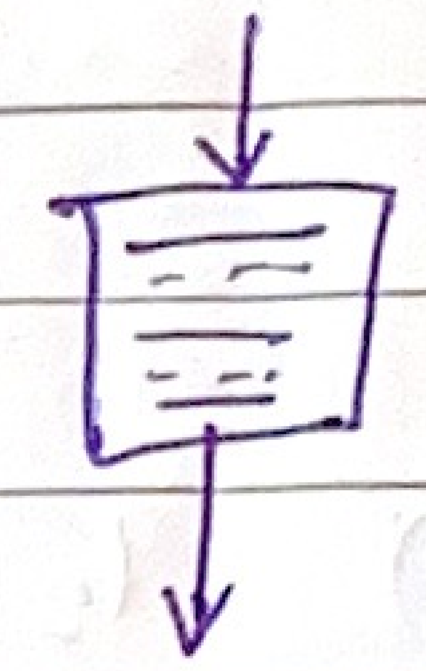
12) $i = 1$ — B5//

Address/ (with goto are leaders) →

13) $t_5 = i - 1$
14) $t_6 = 88 * t_5$
15) $a[t_6] = 1.0$
16) $i = i + 1$
17) if $i <= 10$ goto (13)

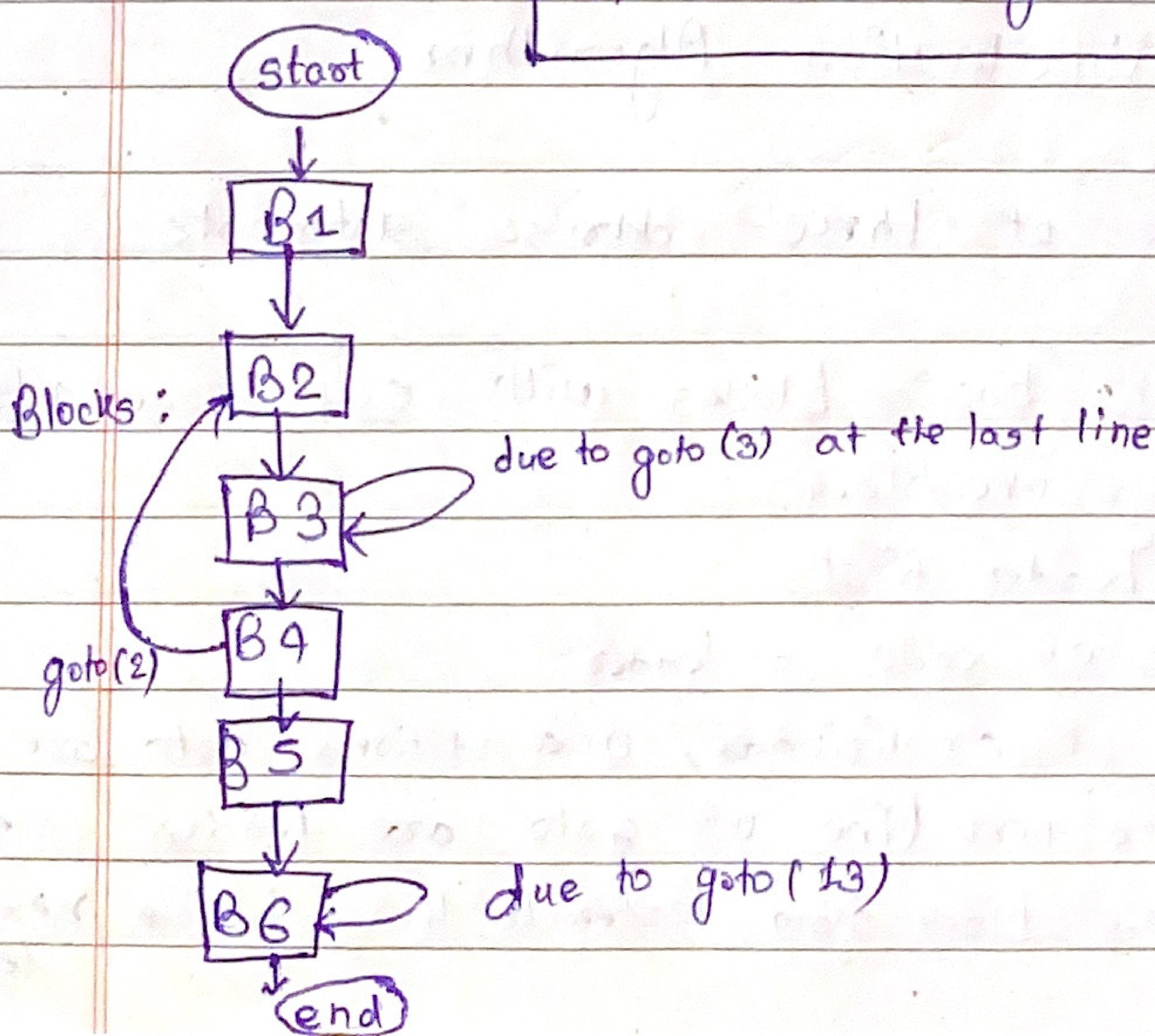
— B6//

Note:
Block consists of one entry point and one exit point.



From leader to the next leader

Blocks :



V. V. Imp

Date
Page

Assignment:

~~*~~ Transformation of Basic Blocks (Blocks lai minimize gorna milxa ki)

→ Common - subExpression elimination
→ Dead Code elimination
→ Renaming Temporary Variables
→ Interchange of statements
→ Algebraic Transformation (ex: $x = x - x$ is 0 (no need)
$x = x + 1$ is x)

V. V. Imp

Peephole Optimization