9/02/23

CD UNIT-III

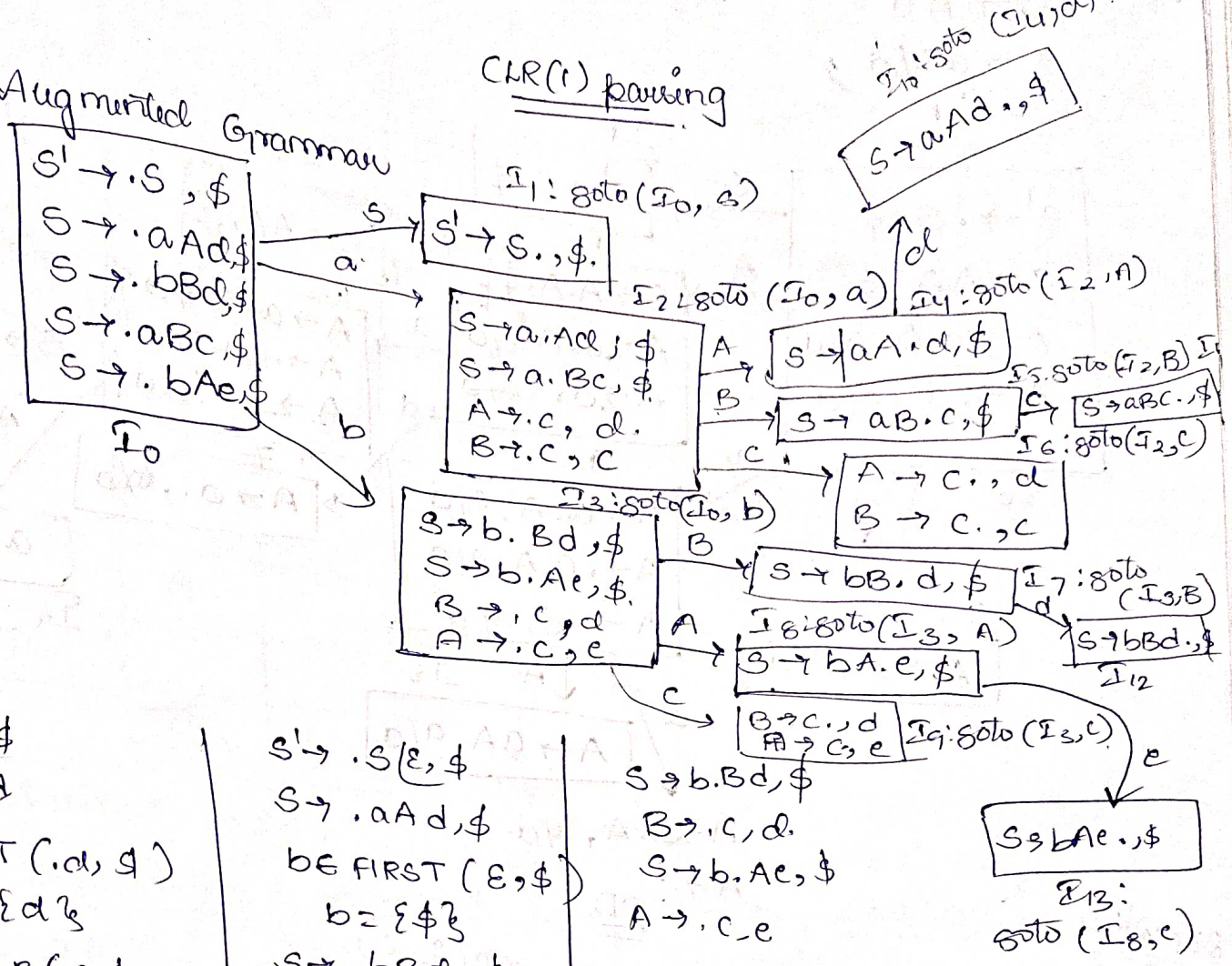
$G = S \to aAd \mid bBd \mid aBc \mid bAe$

$A \to c$

$B \to c.$

CLR(1) parsing

Step 1: Augmented Grammar



$S \to a.A, d/\$$

$A \to .c, d$

$b \in FIRST(.d, \$)$

$b = \{d\}$

$S \to a.Bc, \$$

$B \to .c$

$b \in FIRST(c, \$)$

$b = c.$

$S' \to .S, \varepsilon/\$$

$S \to .aAd, \$$

$b \in FIRST(\varepsilon, \$)$

$b = \{\$\}$

$S \to .bBd, \$$

$S \to .aBc$

$S \to .bAe.$

$S \to b.Bd, \$$

$B \to .c, d.$

$S \to b.Ae, \$$

$A \to .c, e$

Construct

Q) LR(1) items for the grammar $G = S \to AA$

$A \to aA/d$

CLR(1) parsing:

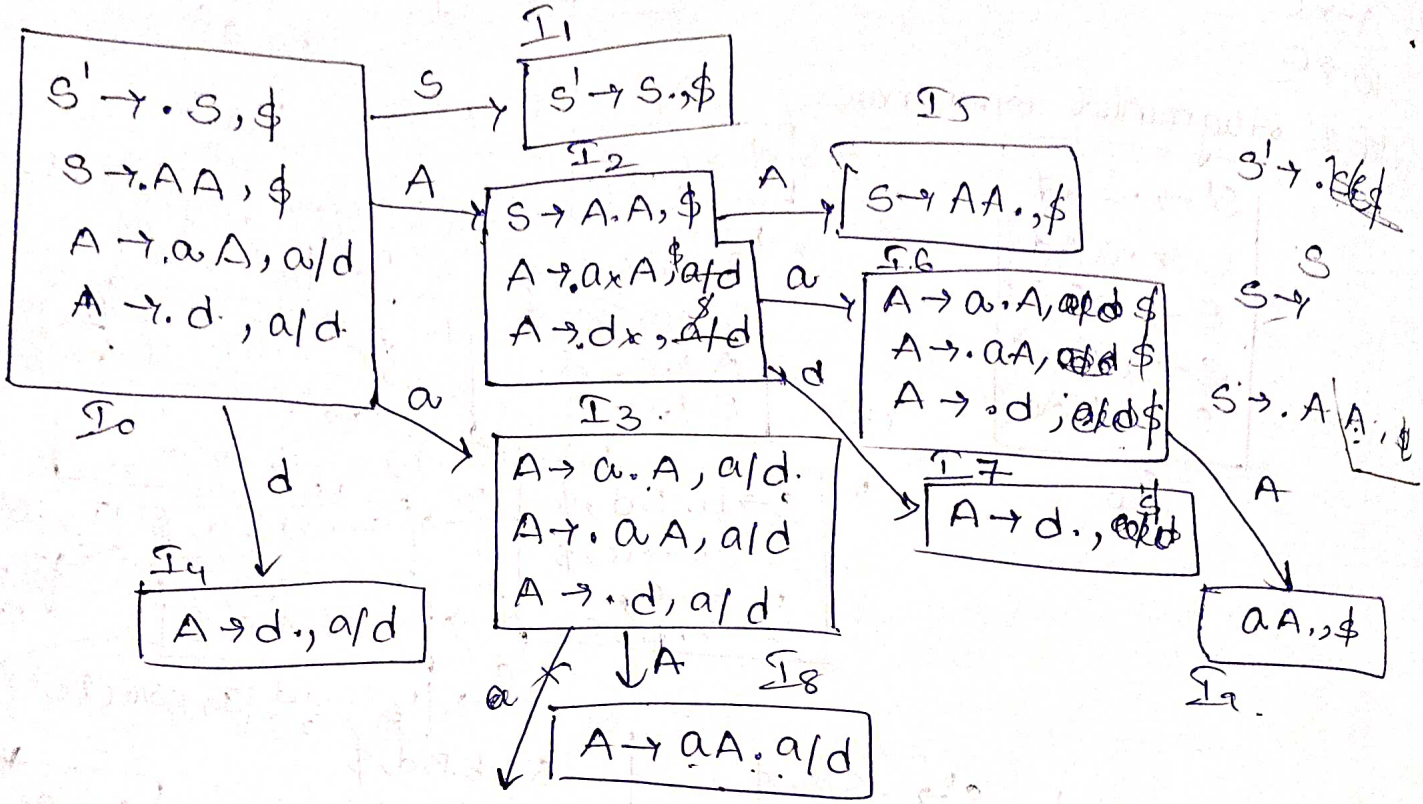
LR(0)
SLR(1) } LR(0) items

CLR(1)
LALR(1) } LR(1) items.

*

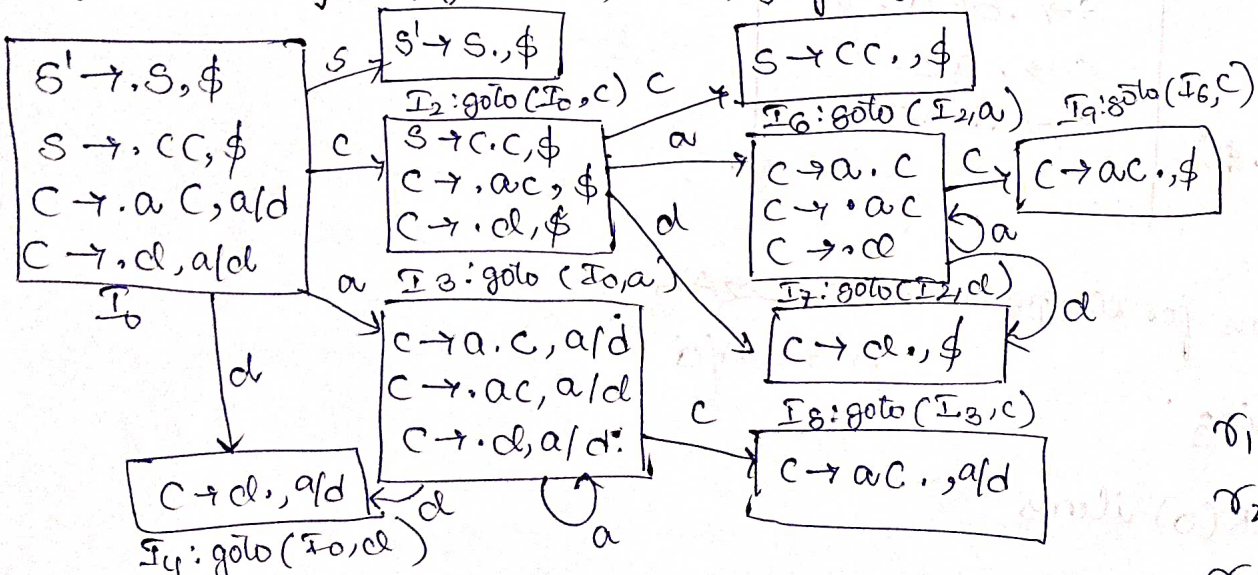
S → AA

A → aA/d



23/02/23

* CLR Parsing :- I1 : goto(I0, S)



$r_1 \rightarrow S \rightarrow .CC$

$r_2 \rightarrow C \rightarrow .aC$

$r_3 \rightarrow C \rightarrow .d$

Step 2: Parsing table :

	Action			Goto	
	a	d	$	S	C
0	S_3	S_4		1	2
1			accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
			r_2		

Step 3: Input string - aadd.

Stack	Input string	Action	Goto	Parsing action
$0	aadd$	[0,a] = S_3		shift
$0a3	add$	[3,a] = S_3		shift
$0a3a3	dd$	[3,d] = S_4		shift
$0a3a3d4	d$	[4,d] = r_3 C→d		reduce
$0a3a3C	d$		[3,C] = 8	shift
$0a3a3C8	d$	[8,d] = r_2 C→aC		reduce
$0a3C	d$		[3,C] = 8	shift
$0a3C8	d$	[8,d] = r_2 C→aC		reduce
$0C	d$	[0,C] = 2		shift
$0C2	d$	[2,d] = S_7		shift
$0C2d7	$	[7,$] = r_3 C→d		reduce

$0C2C	$	[2,C] = 5		shift.
$0C2C5	$	[5,$] = r_1	S → CC.	reduce
$0S	$	[0,S] = 1		shift.
$0S1	$	accept.		

2) Construct the parsing table for the grammar

S → aAd / bBC / bAe

A → c

B → c

	a	b	c	d	e	$	S	A	B
	S_2	S_3					S_1		
1									
2			S_6					4	5
3			S_9					8	7
4				S_{10}					
5			S_{11}						
6			r_6	r_5					
7				S_{12}					
8					S_{13}				
9				r_6	r_5				
10						r_1			
11						r_3			
12						r_2			
13						r_4			

$r_1 \Rightarrow S \to aAd$

$r_2 \Rightarrow S \to bBd$

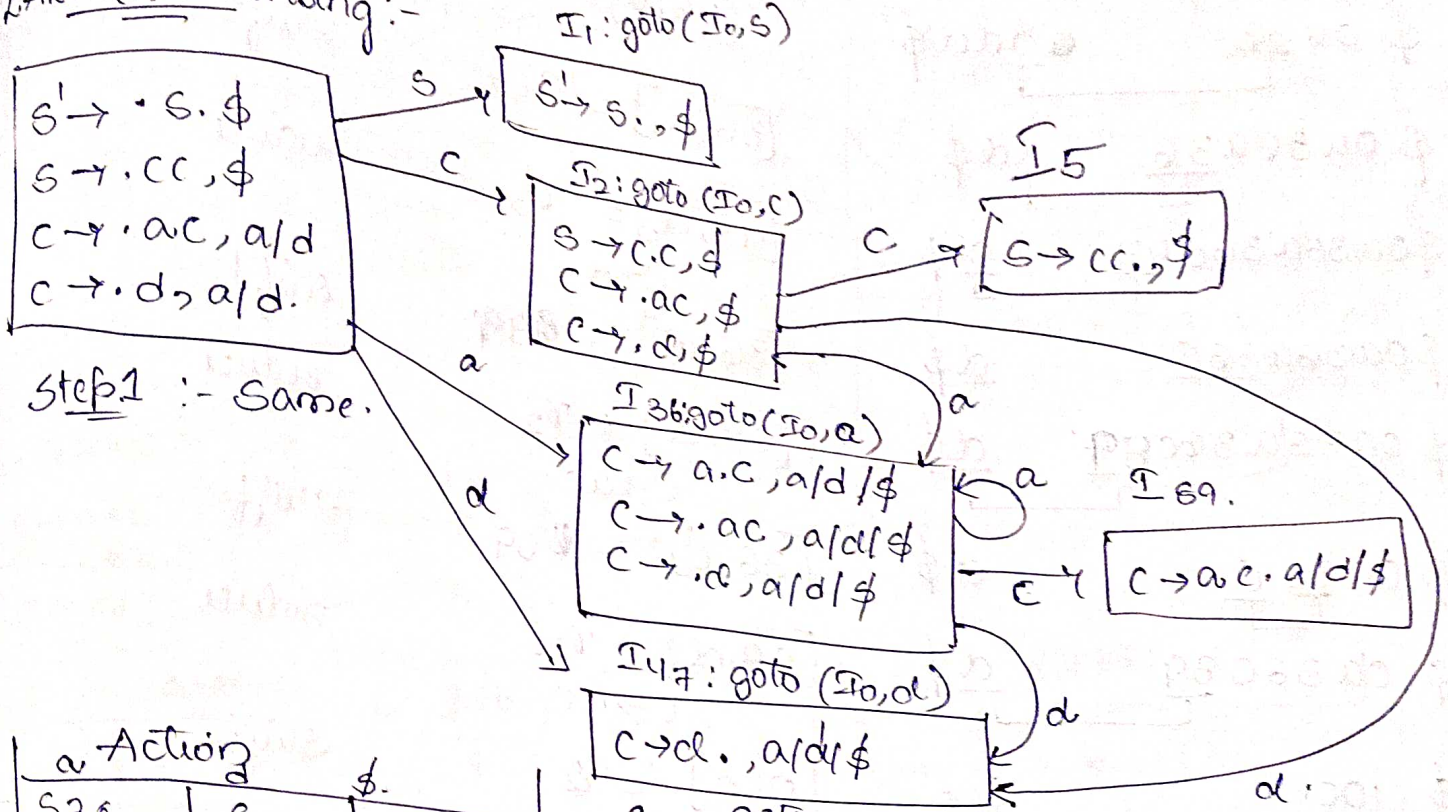
$r_3 \Rightarrow S \to aBe$

$r_4 \Rightarrow S \to bAe$

$r_5 \Rightarrow A \to c$

$r_6 \Rightarrow B \to c$

LALR (1) Parsing :-



Step 1 :- Same.

	a	Action $		S	goto C
0	S36	S47		1	2
1			accept		
2	S36	S47			5
36	S36	S47			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

$S \to cc - r_1$

$C \to ac - r_2$

$C \to d - r_3$

aadcd

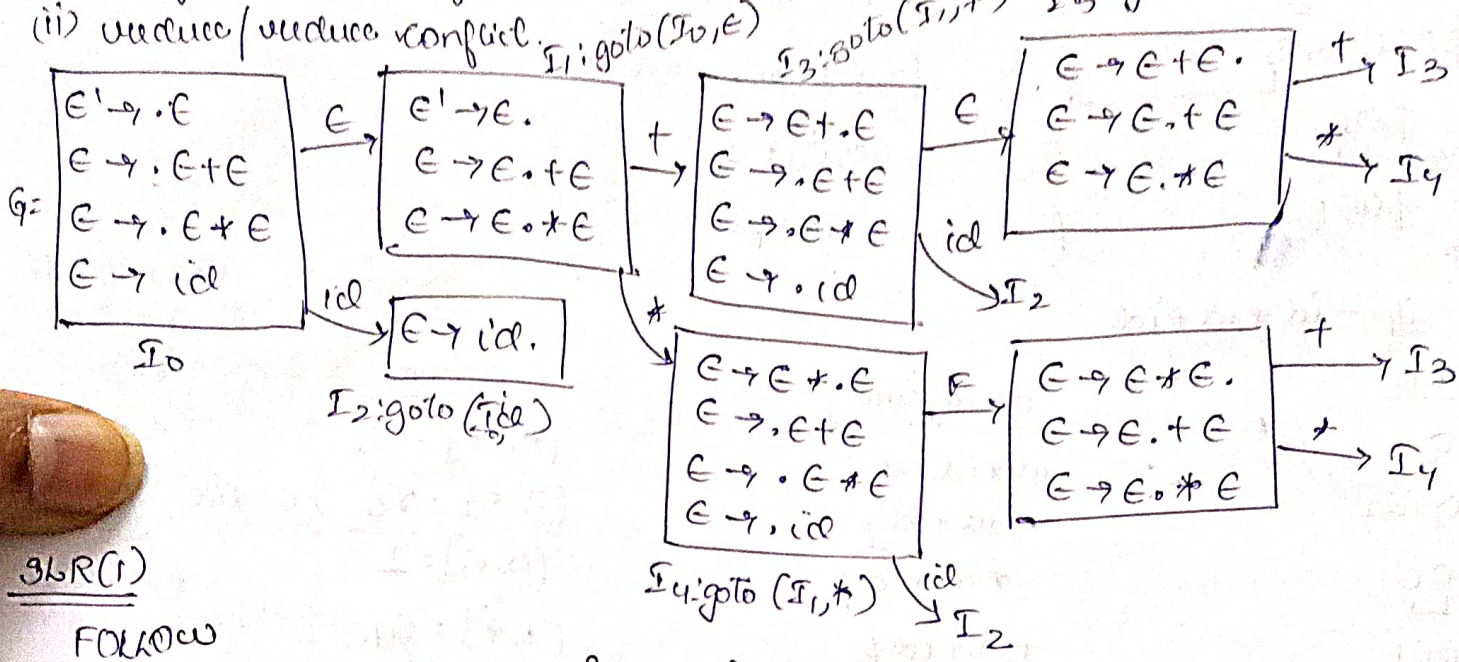
Stack	Input string	Action	parsing action
$ 0	aadd$	[0, a] = S_{36}.	shift.
$ 0a36	add$	[36, a] = S_{36}.	shift
$ 0a36a36	dd$	[36, d] = S_{47}.	shift
$ 0a36a36d47	d$	[47, d] = r_3. C → d.	reduce
$ 0a36a36C	d$	[36, C] = 89	shift
$ 0a36a36C89	d$	[89, d] = r_2 C → ac.	reduce
$ 0a36C	d$	[36, C] = 89	shift
$ 0a36C89	d$	[89, d] = r_2 C → ac.	reduce
$ 0C	d$	[0, C] = 2	shift
$ 0C2	d$	[2, d] = S_{47}	shift
$ 0C2d47	$	[47, $] = r_3 C → d	reduce
$ 0C2C	$	[2, C] = 5	shift
$ 0C2C5	$	[5, $] = r_1 S → cc	reduce
$ 0S	$	[0, S] = 1	shift
$ 0S1	$	[1, $] = accept.	

28/02/23

* Handling ambiguous grammar :-

(i) shift / reduce conflict

(ii) reduce / reduce conflict.



SLR(1)

FOLLOW

$E \to E+E$

$E \to E*E$

$E \to id$

FOLLOW(E) = {$, +, *}

	action				goto
	+	*	id	$	E
0	.		s_2		1
1	s_3	s_4		accept	
2	r_3	r_3		r_3	
3			s_2		5
4			s_2		6
5	s_3/r_1	s_4/r_1		r_1	
6	s_3/r_2	s_4/r_2		r_1	

5, $ = r_1 | 5, + = r_1 | 5, * = r_1

6, $ = r_2 | 6, + = r_2 | 6, * = r_2

2, $ = r_3 | 2, + = r_3 | 2, * = r_3

Stack	input string	Action
$0	id+id*id$	[0, id] = s_2
$0 id 2	+id*id$	[2, +] = r_3 $E \to id$
$0 E	+id*id$	[0, E] = 1
$0 E 1	+id*id$	[1, +] = s_3
$0 E 1 + 3	id*id$	[3, id] = s_2
$0 E 1 + 3 id 2	*id$	[2, *] = $r_3, E \to id$
$0 E 1 + 3 E	*id$	[3, E] = 5
$0 E 1 + 3 E 5	*id$	[5, *] = s_4
$0 E 1 + 3 E 5 * 4	id$	[4, id] = s_2
$0 E 1 + 3 E 5 * 4 id 2	$	[2, $] = $r_3, E \to id$
$0 E 1 + 3 E 5 * 4 E	$	[4, E] = 6

$0E1+3E5*4E6	$	[4,E] = 6
$0E1+3E5*4E6	$	[6,$] = r_2 $E \to E*E$
$0E1+3E	$	[3,E] = 5
$0E1+3E5	$	[5,$] = r_1, $E \to E+E$
$0E	$	[0,E] = 1
$0E1	$	accept

01/3/23

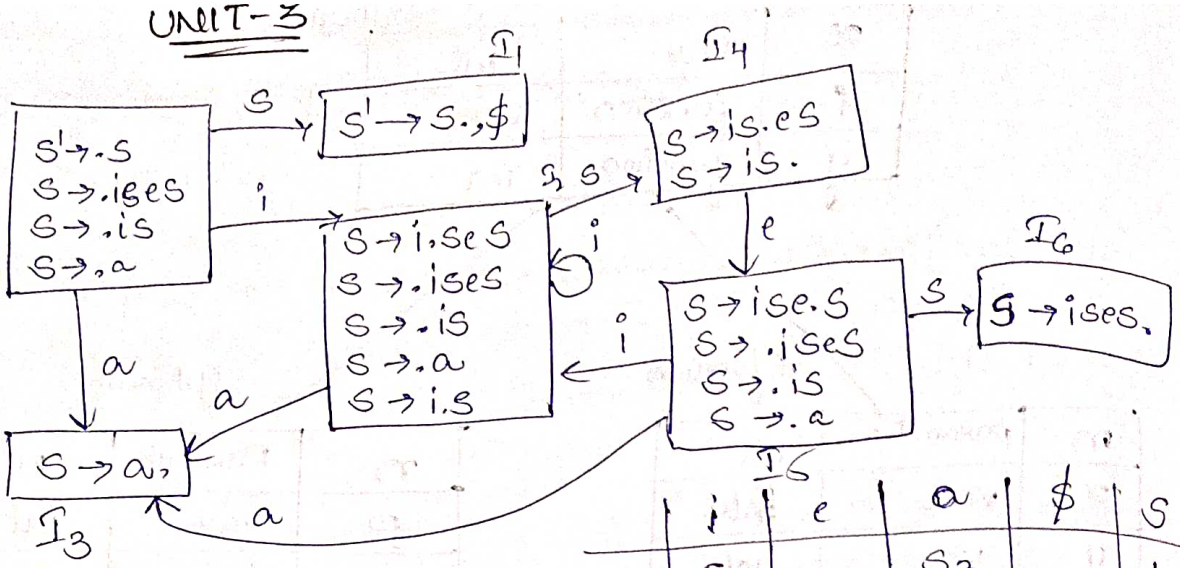
i/p - id * id + id

Stack	input string	action
$0	id * id + id $	[0, id] = S_2
$0 id2	* id + id $	[2, *] = r_3, $E \to id$
$0E	* id + id $	[0, E] = 1
$0E1	* id + id $	[1, *] = S_4
$0E1*4	id + id $	[4, id] = S_2
$0E1*4id2	+ id $	[2, +] = r_3 $E \to id$
$0E1*4E	+ id $	[4, E] = 6
$0E1*4E6	+ id $	[6, +] = r_2, $E \to E*E$
$0E	+ id $	[0, E] = 1
$0E1	+ id $	[1, +] = S_3
$0E1+3	id $	[3, id] = S_2
$0E1+3id2	$	[2, $] = r_3, $E \to id$
$0E1+3E	$	[3, E] = 5
$0E1+3E5	$	[5, $] = r_1, $E \to E+E$
$0E	$	[0, E] = 1
$0E1	$	[1, $] = accept.

01/3/23

UNIT-3

* S → iSeS
  S → iS
  S → a



FOLLOW(S) = {$, e}

S → iSeS S → iS S → a

6,$ = r1 4,$ = r2 3,$ = r3

6,e = r1 4,e = r2 3,e = r3

	i	e	a	$	S
0	S2		S3		1
1				accept	
2	S2		S3		4
3		r3		r3	
4		S5/r2		r2	
5	S2		S3		6
6		r1		r1	

Parse i/p string : iiaea

Stack	i/p string
$0	iaea$
$0i2	iaea$
$0i2i2	aea$
$0i2i2a3	ea$
$0i2i2S	ea$
$0i2i2S4	ea$
$0i2i2S4e5	a$
$0i2i2S4e5a3	$
$0i2i2S4e5S	$
$0i2i2S4e5S6	$
$0i2S	$
$0i2S4	$
$0S	$
$0S1	$

action

[0,i] = S2

[2,i] = S2

[2,a] = S3

[3,e] = r3 (S → a)

[2,S] = 4

[4,e] = S5

[5,a] = S3

[3,$] = r3 (S → a)

(5,S) = 6

[6,$] = r1 (S → iSeS)

[2,S] = 4

[4,$] = r2 (S → iS)

[0,S] = 1

accept

Comparision of Top down and Bottom-up parsers.

Top-Down	Bottom up
- it starts with root	- it starts with leaf
- the process starts with Start symbol	- the process ends with starts symbol
- it follows left most derivation	- it follows RMD
- it will not accept ambiguous grammar	- it accepts ambiguous grammar.
- it is less powerful compared to bottom up parsers.	- it is more powerful compared to top-down parser.
- It is easy to produce parsers	- it is difficult to produce parsers.
- it follows LL(1) grammar to perform parsing.	- it follows LR(0) and LR(1) grammar to perform parsing.
- error detection is weak.	- error detection is strong.

13/02/20

UNIT-IV

Syntax Directed Translation

* SDD (syntax directed difinition) :-

⇒ $E \rightarrow E+T$ | { E.val := E.val '+' T.val. }

E.val
T.val
+.addition

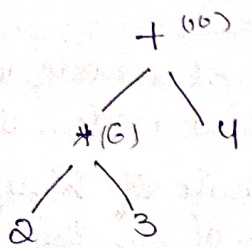
E.val → E.val '+' T.val.

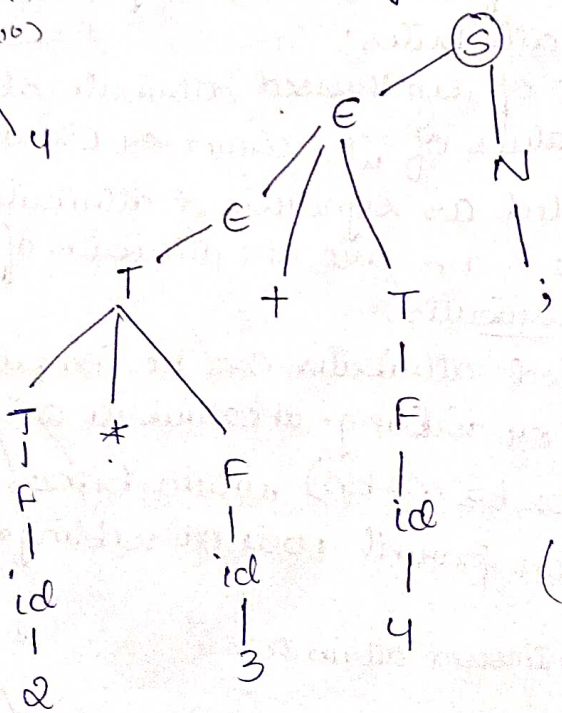
Symmantic rule/action

* $S \rightarrow EN$ { S.val := E.val } (or) { print (E.val) }
  $E \rightarrow E+T$ { E.val := E.val '+' T.val }
  $E \rightarrow E-T$ { E.val := E.val '-' T.val }
  $E \rightarrow T$ { E.val := T.val }
  $T \rightarrow T*F$ { T.val := T.val '*' F.val }
  $T \rightarrow T/F$ { T.val := T.val / F.val }
  $T \rightarrow F$ { T.val := F.val }
  $F \rightarrow (E)$ { F.val := E.val }
  $F \rightarrow id$ { F.val := id.lexval }
  $N \rightarrow ;$ { terminal Symbol }.

* Input String 2*3+4; Is it correct symmantically not syntatically?

Syntax tree



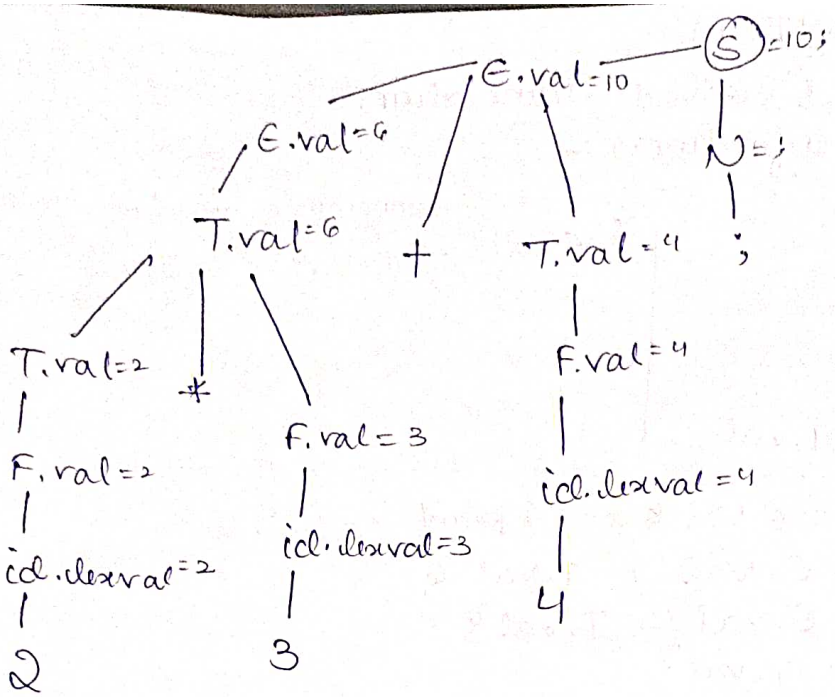


(Parse tree)

Annotated parse tree

↓

(T*F ← T)
So directly T rather than T → T.



Annotated Parse Tree

* SDD:-

→ It is a context free grammar associated with attributes and Symmantic rules. the attribute can be the string, datatype, memory allocation, lexical value, etc

→ There are 2 types of attributes, they are

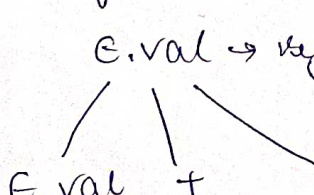
1) Synthesised attributes:

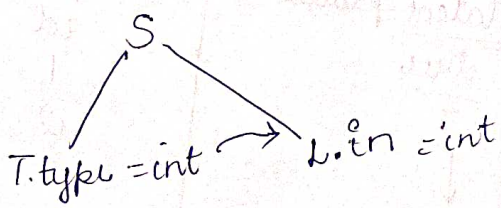
the value of synthesised attribute at a node is computed from the values of its children or child nodes. the attribute 'a' is called as synthesised attribute of X, if $a:=f(b_1, b_2, \ldots b_n)$ where $b_1, b_2, \ldots b_n$ are the attributes of its child nodes.

2) Inherited attributes:-

the Inherited attributes can be computed from the values of its parent or sibling. the attribute a is called Inherited attribute if $a=f(b_1, b_2, b_3, \ldots b_n)$ where $b_1, b_2, \ldots b_n$ are the attributes of either parent node or sibling node.

Eg:



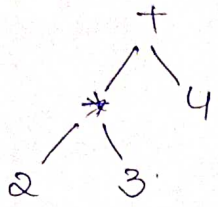


Inherited attributes

Eg: Construct syntax tree Parse tree and annotated Parse tree for 2*3+4; from the context free grammar.

S → EN
E → E+T
E → E-T
E → T
T → T*F
T → T/F
T → F
F → (E)
F → id
N → ;

Syntax tree



Parse tree

↓

before drawn in previous page.

for constructing annotated PTree we need to have Syntax Directed definition.

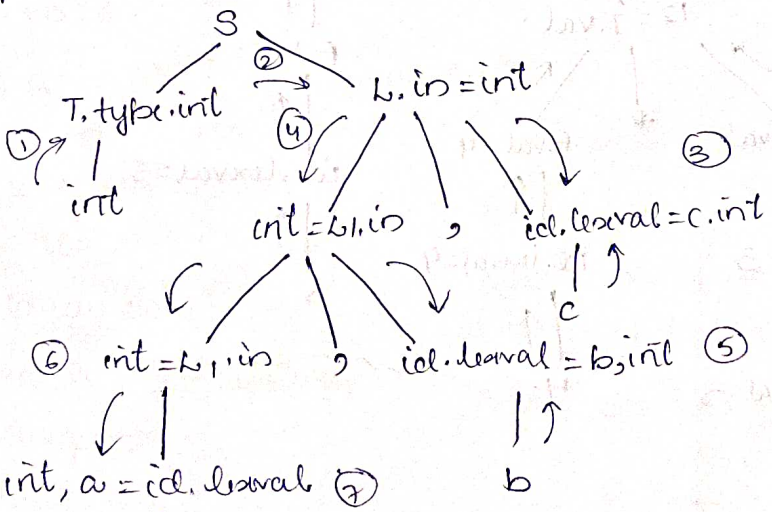
SDD for given grammar is (written in previous page).

annotated PT (drawn in previous page).

14/3/23

① S → TL { L.in := T.type }
T → int { T.type := int }
T → float { T.type := float }
T → char { T.type := char }
T → double { T.type := double }.
L → L, id { L1.in = L.in, entertype (id.lexval, L.in) }
L → id { L.in := id.lexval }

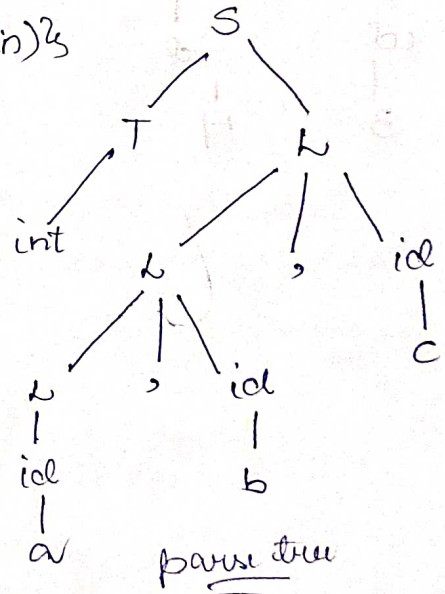
i/p string → int a, b, c



without arrow = annotation parse tree

diagram ①

with arrow = dependency.



parse tree

Inherited attribute.

Inherited attribute are evaluated either from parentnode or sibling node

for example:- Consider syntax directed definition①

Annotated parse tree can be constructed for the i/p string int a, b, c.

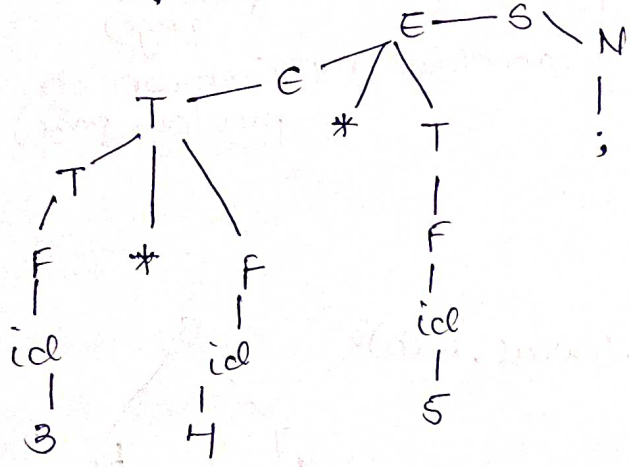
Synthesized attribute → all arrows as

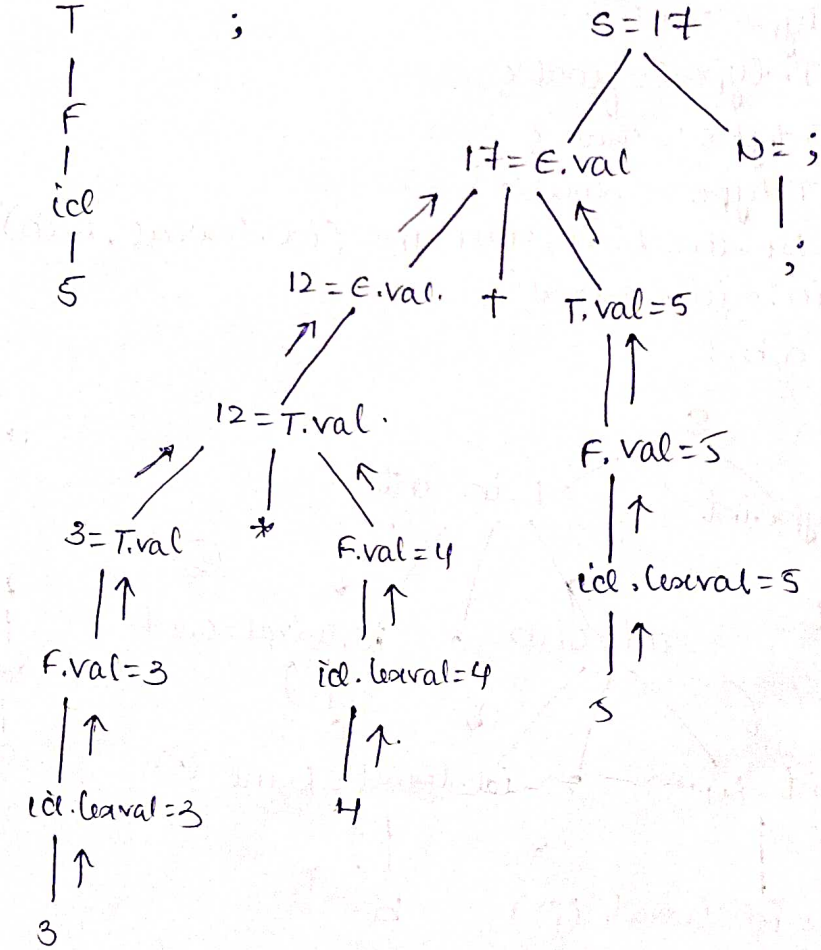
Synthesized attribute → S. attributed tree

Synthesized and inherited → L. attribute tree
attribute

S-attribute:- the SDD uses only synthesized attributes then it is called as 'S-attribute'.

=> Construct the dependency graph for the i/p string 3*4+5 and also identify the evaluation order





15/03/23

CD

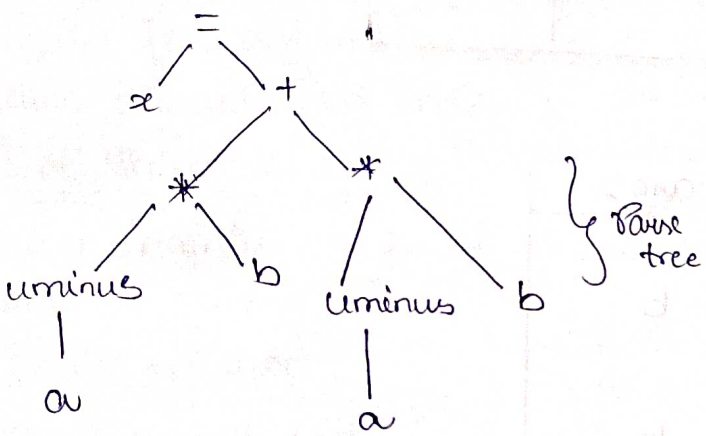
* Intermediate Code generation:-

(i) Abstract syntax tree
DAG -> direct acyclic graph

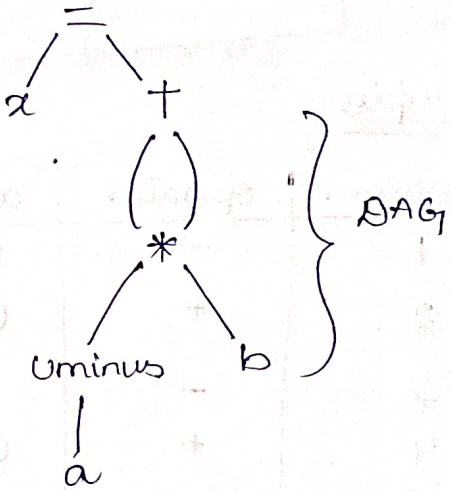
(ii) Polish notation / reverse polish notation
a+b → infix
+ab → Prefix
ab+ → postfix

(iii) Three address code.

Eg: $x = (-a*b) + (-a*b)$



Parse tree



DAG

Eg: (a+b) * (c-d)
* +ab -cd

Polish notation.

Eg: x=(a+b) * (c-d)
$t_1 := a+b$
$t_2 := c-d$
$t_3 := t_1 * t_2$
$x := t_3$

three address code

* Three address code:-

3 operands, 2 operators

(i) Quadruples
(ii) Triples
(iii) Indirect Triples

$t_1 := a+b$

Eg: $x := (-a*b) + (-a*b)$
$t_1 := \text{uminus } a$
$t_2 := t_1 * b$
$t_3 := \text{uminus } a$
$t_4 := t_3 * b$
$t_5 := t_2 + t_4$
$x := t_5$

Number	operator	arg1	arg 2	result
1	-	a		t_1
2	*	t_1	b	t_2
3	-	a		t_3
4	*	t_3	b	t_4
5	+	t_2	t_4	t_5
6		t_5		x

(ii) Triples

Number	operator	arg1	arg 2
1	-	a	
2	*	(1)	b
3	-	a	
4	*	(3)	b
5	+	(2)	(4)
6	=	(5)	x

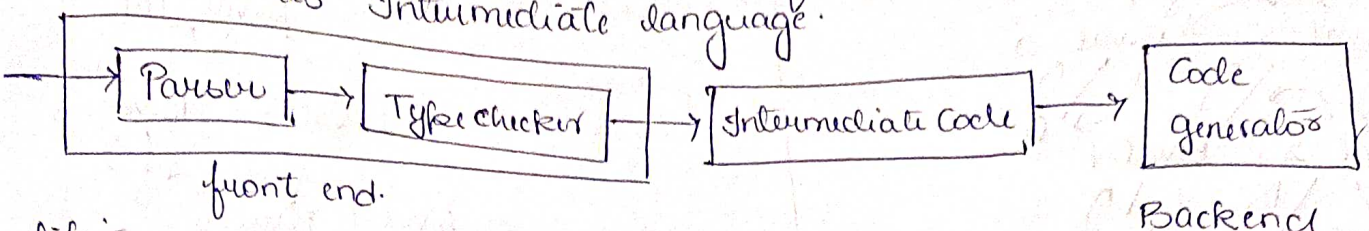
(iii) Indirect triples

Number	operator	arg1	arg 2
1	-	a	
2	*	(10)	b
3	-	a	
4	*	(12)	b
5	+	(11)	(13)
6	=	(14)	x

1	10
2	11
3	12
4	13
5	14
6	15

* Intermediate Code generation :-

→ Compilers generate an easy to represent form of source language which is called as Intermediate language.



* Benefits :-

1) A compiler for different machines can be created by attaching different back ends to existing front end.
2) A compiler for different source languages can be created by providing different front ends.

* forms of Intermediate Code :-

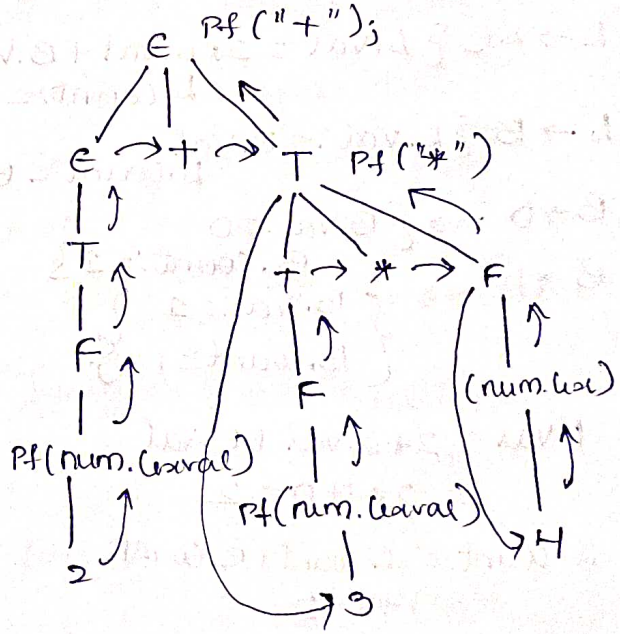
There are mainly 3 types of Intermediate code representations.

(1) abstract syntax tree
(2) polish notation
(3) three address code.

16/3/23

Syntax Directed Translation Schema (SDT) :-

E → E+T { Pf("+"); } 2+3*4
E → T { }
T → T*F { Pf("*"); }
T → F { }
F → num { Pf (num.lexical) }

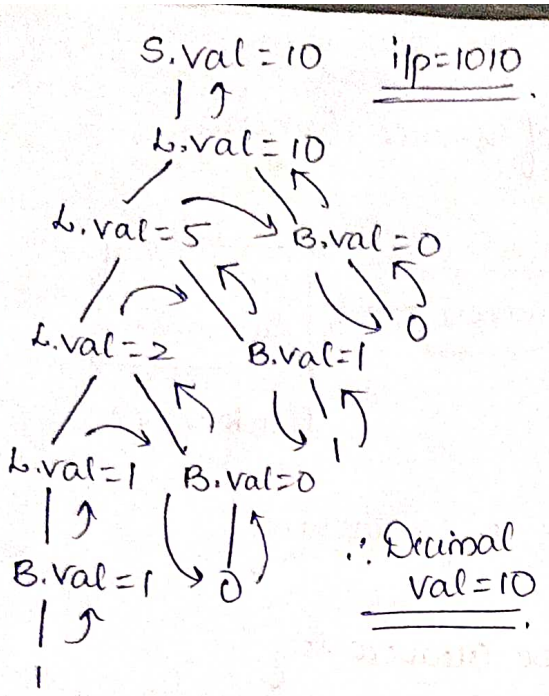


* Convert the given binary no. into decimal no. using SDT :-

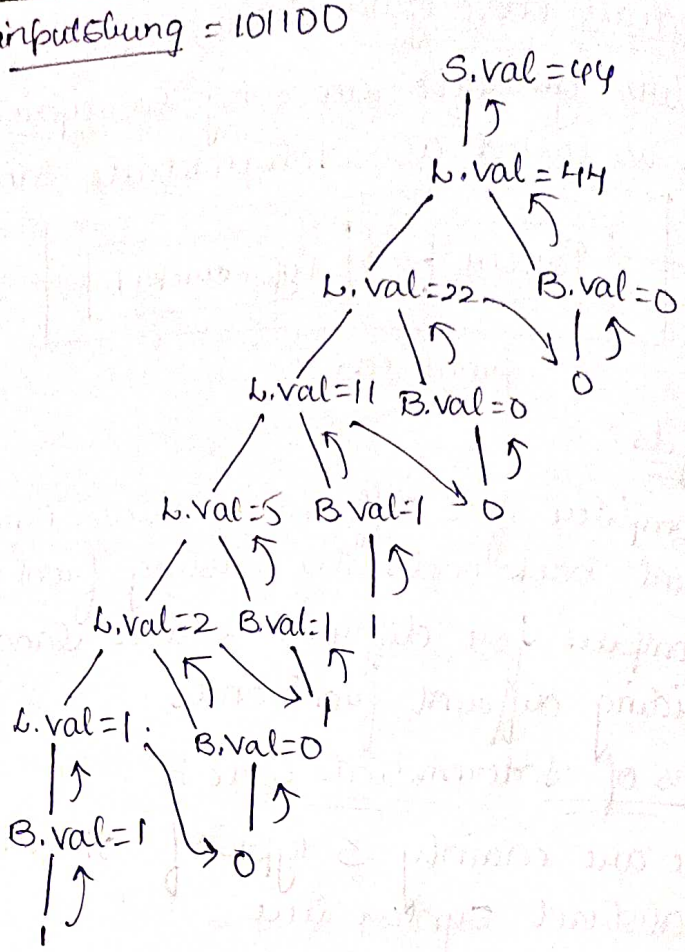
S → L { S.val := L.val }
L → LB { L.val := 2 * L.val + B.val }
L → B { L.val := B.val }
B → 0 { B.val := 0 }
B → 1 { B.val := 1 }

d.val = 2 × L.val + B.val .

input string = 1010 .



∴ Decimal val = 10



$S \to L_1 . L_2$

20/3/23

SDD translation for fraction at numbers:-

$S \to L_1 . L_2$ { S.val := L1.val + (L2.val / 2^{L2.count}) }

$L \to LB$ { L.val = 2 * L.val + B.val
L.count := L.count + B.count }

$L \to B$ { L.val := B.val
L.count = B.count }

$B \to 0 \to$ { B.val := 0
B.count := 1 }

$B \to 1 \to$ { B.val := 1
B.count := 1 }

l.val = 2 * l.val + B.val
= 2 * 1 + 0 = 2

l.count = l.count + B.count
= 1 + 1 = 2

S.val = L_1.val + $\frac{L_2.val}{2^{L_2.count}}$

= $5 + \frac{5}{2^3}$ = $5 + \frac{5}{8}$

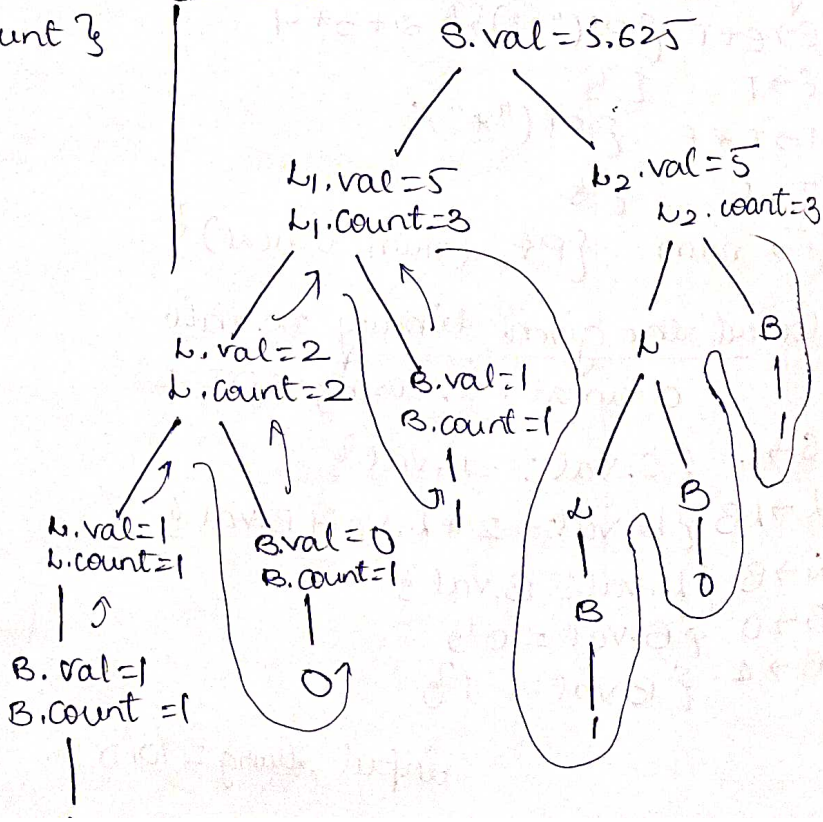
= 5 + 0.625

= 5.625

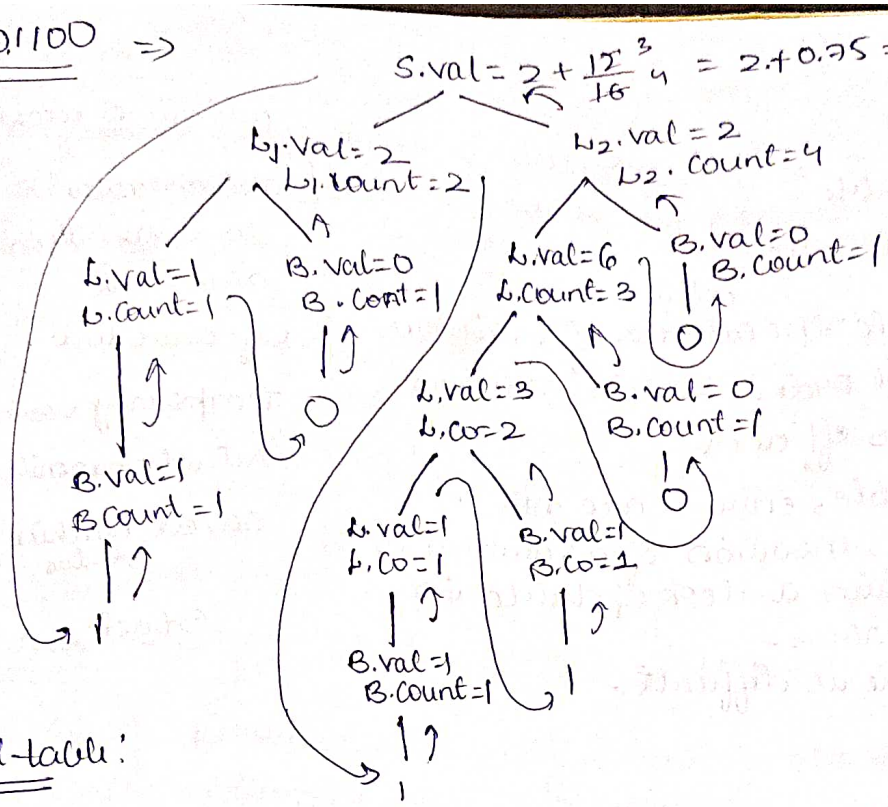
Ex: Conversion of binary to decimal with fraction:

101.101 = 5.625

5 .625



i/p = 10.1100 ⇒



* Symbol-table:

Symbol table is a datastructure used at Compilation to store the source code Constructs - the source code constructs can be

(1) Variable name: when a variable is identified it is stored in the Symbol table by its name.

(2) Constant: these are stored in the symbol table with the help of pointers.

(3) datatypes: datatypes associated with variables are stored in symbol table.

(4) temporary variables: while generating intermediate code, if any temporary variables are generated, they will be stored in Symbol-table.

(5) Function names: the names of functions will be stored in the Symbol-table.

(6) Function parameters: parameters passed to the function are stored in the Symbol-table.

(7) Scope of the variables: the range of variables can be stored in the Symbol-table.

21/3/23 CD

* Symbol table :-

(i) Ordered Symbol table
(ii) Unordered Symbol table.

In ordered symbol table, the entries of variables is made in alphabetical order. In this method searching of variables is efficient.

In unordered Symbol table, entries are not in sorted order. In this, insertion of a variable is efficient. It will maintain a look up table to store the details of variables.

Searching of a variable is difficult.

activation record

return value
local variable
access link
Dynamic link
Temporary variable
Actual parameters
Saved machine status

Stack.

* Symbol table format :-

1) fixed length name Hape :-

a fixed space is allocated for each name in symbol table. If the name is too small then ø there is wastage of space.

name					attributes
S	u	m			
a	d	d			
i	n	t			

2) Variable length name :-

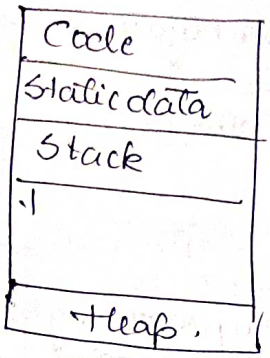
In this table, the amount of space required by the string is used to store the names.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	d	d	$	s	u	m	$	a	$	b	$	c	$

1	3	information
5	3	info.
9	1	
11	1	
13	1	

* Storage organization :-

the compiler utilizes a block of memory to execute the compiled program. this block of memory is known as runtime memory. This runtime storage is divided as



* Code area :-

→ the size of generated code. is fixed, hence the target code occupies the statically determine area of the memory.

* Static data area :-

→ the amount of memory required by the data objects is known at the compiled time. Hence, data objects can also be stored at the determine area.

* Stack area :-

→ the data structure to create the stack is dynamic. It stores all the activation records. As activation begins, the activation records are pushed into the stack and after completion the corresponding activation records is poped from the stack. It is also known as Control stack

* Heap :-

→ If the values of non local variables must be retain after the activation record then those data values will be stored in the heap. It allocated continuous block of memory. It can be deallocated when activation ends.

Saturday

* comparision between static, stack, heap.

static	stack	heap.
→ static allocation is done for all data objects at compile time.	→ It is used to manage frontend.	→ it is used to manage dynamic memory allocation
→ datastructures cannot be created dynamically because compiler can determine the storage space required by each data object.	→ ds and data objects can be create dynamically.	→ ds and dataobjects can be created dynamically.
→ the names of data objects are bound to storage at compile time.	→ the memory addressing can be done using index and registers bcz in stack the activation records are pushed according to the requirements.	→ contiguous block of memory from heap is allocated. A linked list is maintained for free blocks.
→ it is simple but recursive procedures are not supported	→ it supports recursive procedures but it can not access non-local name.	→ efficient memory management done using linked list. the deallocated memory spaces can be reused.

* Activation record:

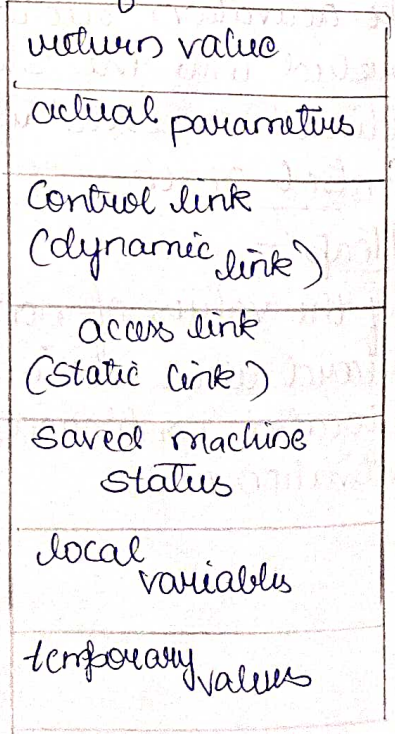
it is a block of memory used for maintaining the information for execution of a single procedure.

(i) return value:- this field is used to store the results of a function call.

(ii) actual parameters:- this field holds the information about the actual parameters passed to the called procedure.

(iii) control link:- (dynamic link):- it points to the activation record for the calling procedure.

(iv) access link:- it refers to the not local data in other activation record. it is also known as static link.



(v) saved machine status :- this field contains info regarding the state of machine just before the procedure is called. if it contains the details of the register, program counter etc.

(vi) local variable :- it contains the local data of procedure.

(vii) temporary values : it contains details of temporary values which are used during evaluation of an expression.

27/03/23

* Data Structures in symbol table :- There are 4 types of data structures to maintain the symbol table.

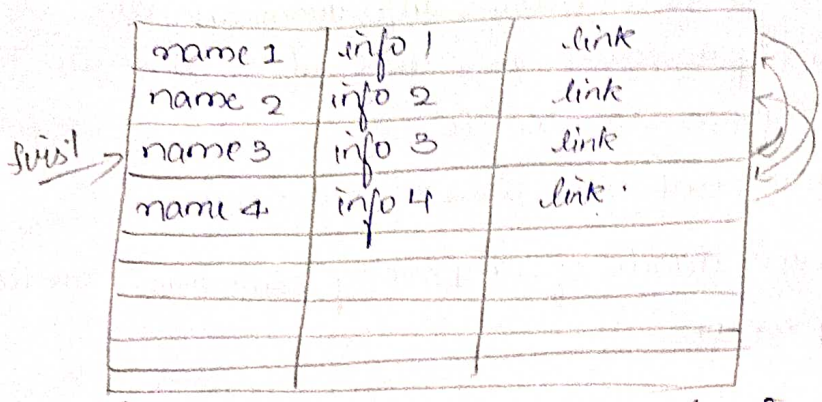
1) list data structure (linear list)
2) linked list (self-organized list)
3) Binary trees
4) Hash tables.

1) list data structure :-

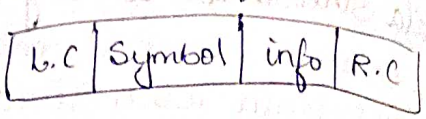
→ it is a simple mechanism to manage the symbol-table.
→ In this method an array is used to store names and associated information.
→ New names can be added in the order as they are arrived.
→ the pointer is maintained at the end of all stored records.
→ to search any element we have to search the entire table from beginning to insert to new name we should ensure that the name should not be there in the list.

name 1	info 1
name 2	info 2
name 3	info 3

2) linked list :- in this datastructure a link field is added to each record.
→ We search the record in the order pointed by the link field.
→ a pointer first is maintained to point the first record.
→ the most frequently used names will be infront of the list. hence it is easy to access frequently referred names.
→ it takes less space and insertion of new symbol is easy.



(3) Binary trees :- In Binary tree data structure each node of the symbol table contains four fields
- left child
- symbol
- information
- rightchild



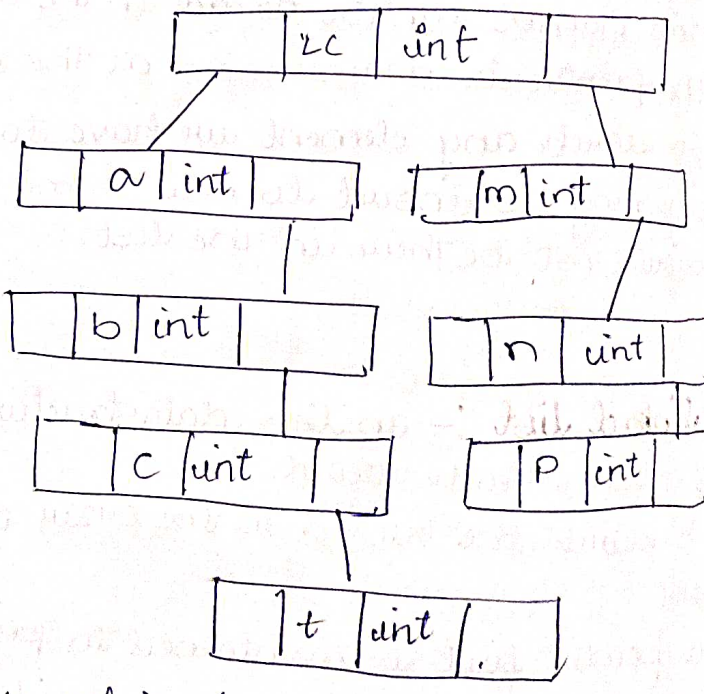
→ The leftchild stores the address of previous symbol and rightchild stores the address of next symbol.

→ the symbol field is used to store the name and information field is used to store the information about the symbol.

→ Insertion of a symbol is easy and searching can also be done efficiently

→ It takes more space to store the pointer values.

→ int m,n,p
int compute (int a, int b, int c)
{ t = a*b+c;
return(t);
}
void main()
{ int k
k = compute (10,20,30)
}



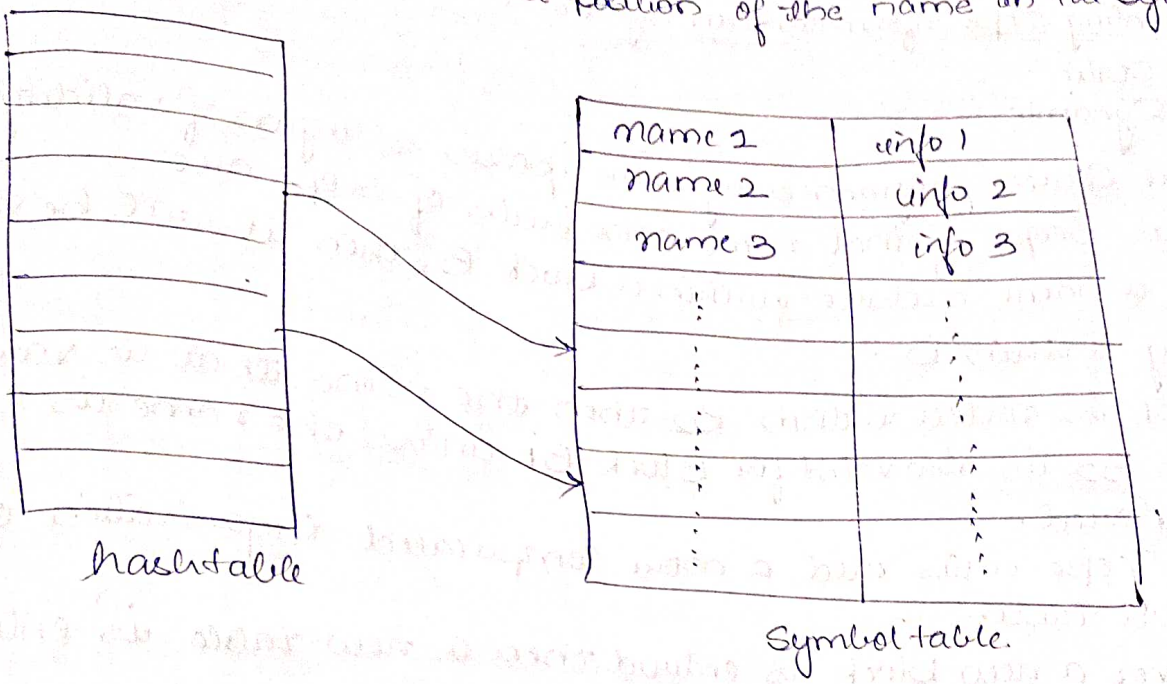
(4) Hash tables:-

In this Data Structure 2 tables are maintained.

(i) Hashtable (ii) Symbol-table.

(i) Hashtable contains the pointers to the symbol table pointing to the names of symbol table.

→ to find any name uses hash function as h(name)=position.
→ the position represents the exact position of the name in the symbol table.



28/03/23

Representing Scope Information

* Representing scope information in symbol table:
  1. Static
  2. Dynamic

=> In the source program every name possess a region of validity called as scope of that name. The rules of scope are

(1) If a name declare within a block B, then it will be valid only within B.

(2) If B1 is nested within B2 then the name that is valid for Block B2 is also valid for Block B1 unless the name is redeclared.

(3) The scope rules need a more complicated organisation of symbol-table.

(4) Whenever a new block is entered then a new table is entered on to the stack.

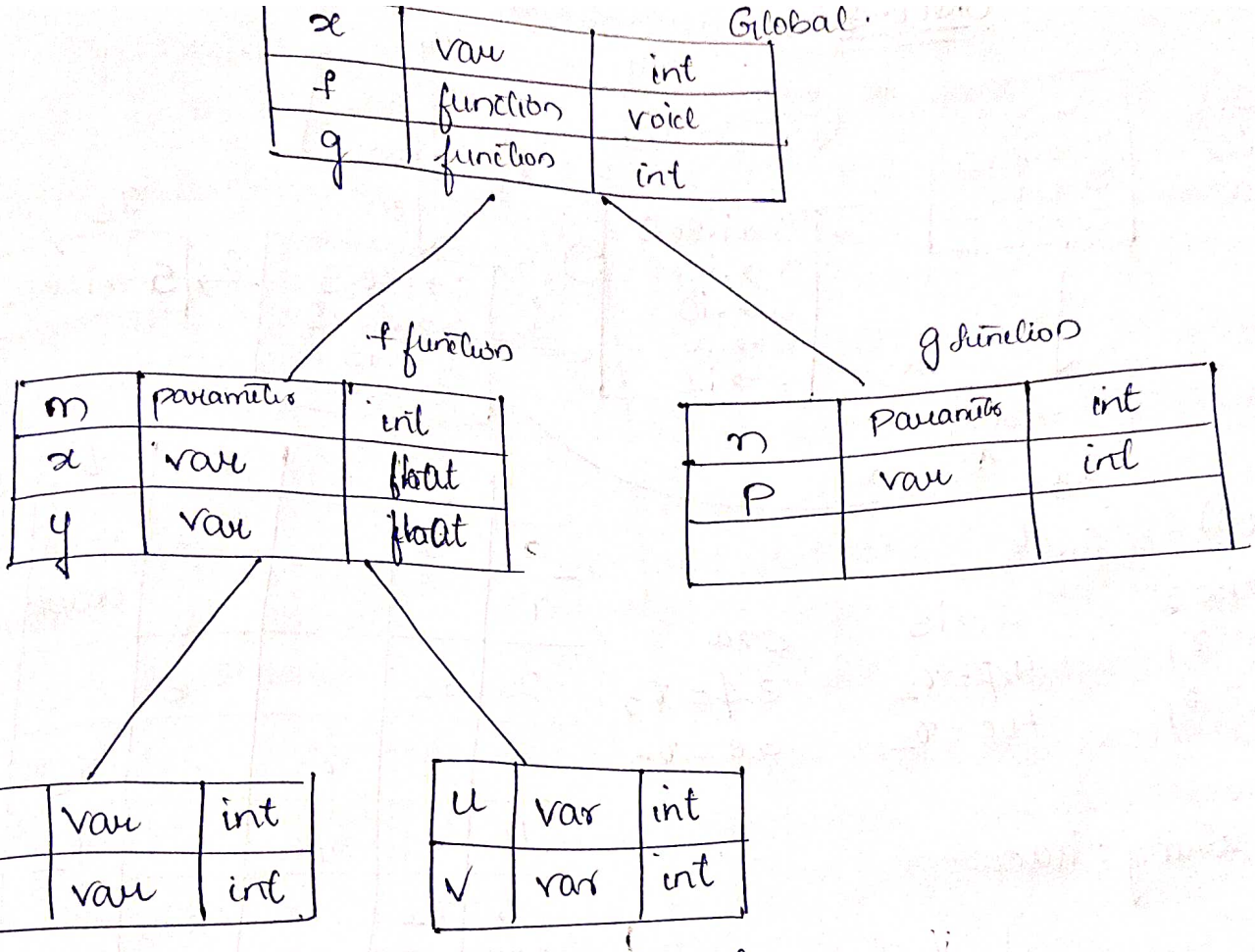
(5) The new-table contain the names declared as local to that block.

(6) These tables are organised into the stack

(7) When a name is declared the table is searched for the name, if the name is not found then the new name is inserted into the table.

Eg:

```
int x;
void f(int m)
{
  float x,y;
   {
     int i,j;
     int u,v;
   }
}
int g(int n)
    {
      int p
    }
```



10/4/23 In whatsapp, the pages are shared.

24/4/23

Runtime Storage:

1. Storage allocation
2. Stack allocation of space
3. access to non local data.