Syntax Directed Translation in Compiler Design

Syntax Directed Definition

SDD = CFG + Semantic Rules

CFG → Context Free Grammar

→ A SDD is a Context Free grammar together with Semantic Rules

→ Attributes are Associated with grammar symbols and Semantic Rules are associated with productions.

→ If 'x' is a symbol and 'a' is one of its attribute then x.a denotes value at node 'x'.

→ Attributes may be numbers, strings, references, datatypes etc.

production	Semantic Rule
$E \rightarrow E + T$	$E.val = E.val + T.val$
$E \rightarrow T$	$E.val = T.val$

Scanned with CamScanner

Types of Attributes:-

1. **Synthesized Attribute :-** If a node takes value from its children then its Synthesized Attribute.

Ex :- $A \rightarrow BCD$, A be a parent node
B, C, D are children nodes.

$A.S = B.S$
$A.S = C.S$
$A.S = D.S$

} parent node A taking value from its children B, C, D

2. **Inherited Attribute :-** If a node takes value from its parent or Siblings

Ex:- $A \rightarrow BCD$

$C.i = A.i \rightarrow$ parent node
$C.i = B.i \rightarrow$ sibling node
$C.i = D.i \rightarrow$ sibling node

Types of SDD:-

1. S-Attributed SDD (or) S-Attributed Definitions (or) S-Attributed grammar
2. L-Attributed SDD (or) L-Attributed Definitions (or) L-Attributed grammar

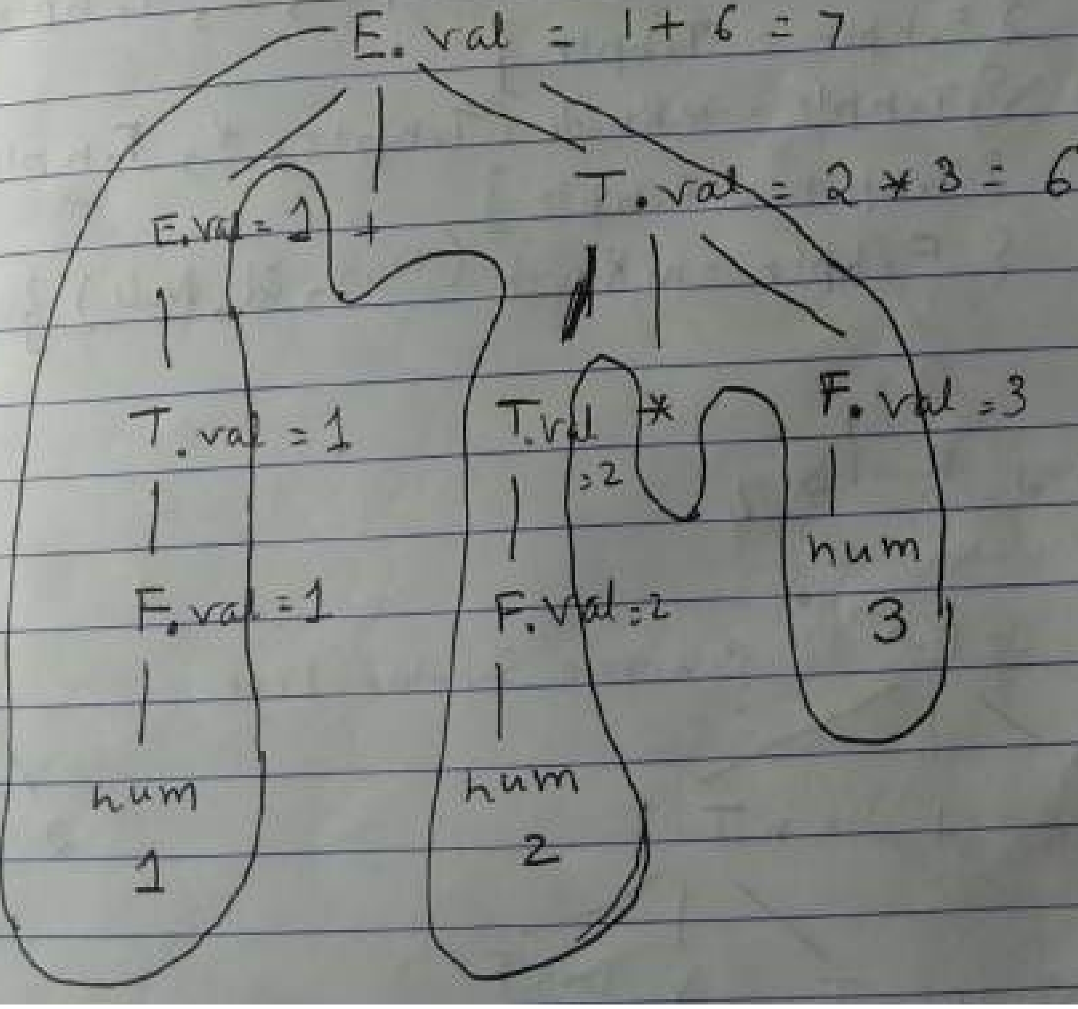
S-Attributed SDD	L-Attributed SDD
1. A SDD that uses only synthesized attributes is called as S Attributed SDD. Ex:- $A \rightarrow BCD$ $A.S = B.S$ $A.S = C.S$ $A.S = D.S$	1. A SDD that uses both Syntesized & inherited Antributes is called L-Attributes SDD but each inherited Attribute is restricted to inherits from parent or left sibling only. Ex:- $A \rightarrow xyz\{Y.S = A.S,\ Y.S = X.S,\ Y.S = Z.S\}$
2. Semantic Actions are always placed at right end of the production. It is also called as "pos-tfix SDD".	2. Semantic Actions are placed anywhere on RHS.
3. Attributes are Evaluated with Bottom-up Parsing	3. Attributes are Evaluated by traversing parse tree depth first, left to right order

Top down right
left to right

Creating Parse Tree

Grammar	Actions {Semantic Rule}
$E \to E + T$	{E.val = E.val + T.val}
$E \to T$	{E.val = T.val
$T \to T * F$	{T.val = T.val * F.val}
$T \to F$	{T.val = F.val
$F \to num$	{F.val = num.lval}

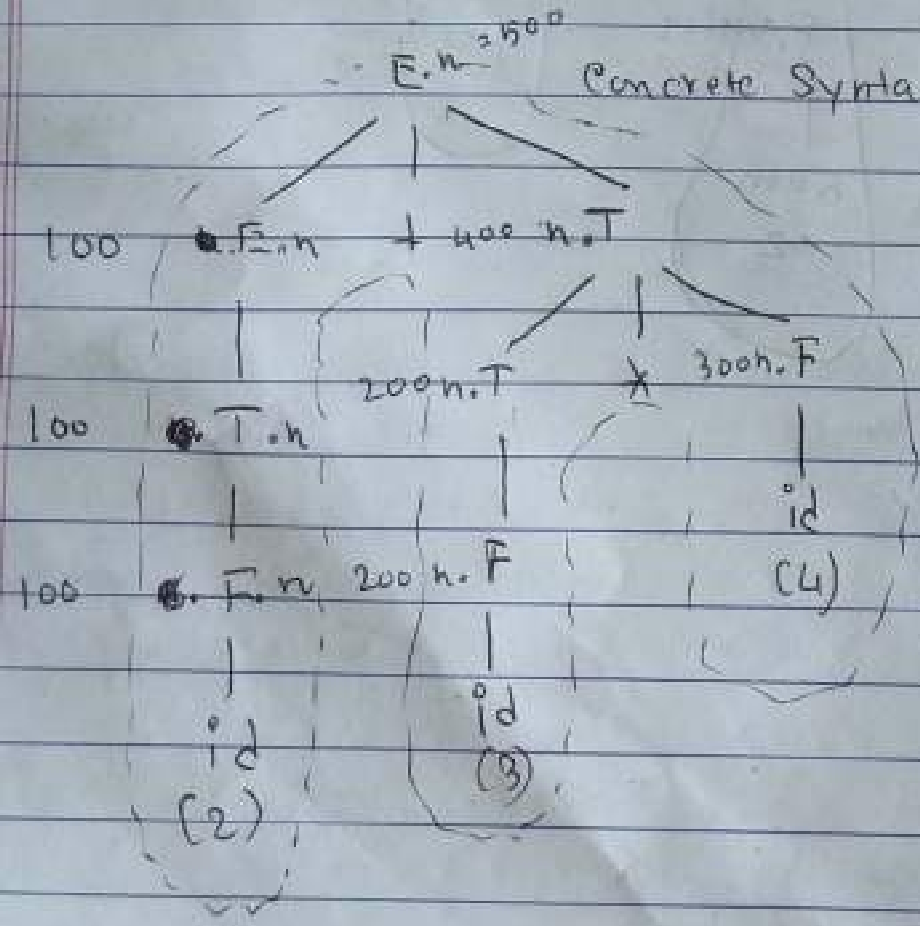
Eg: $1 + 2 * 3$



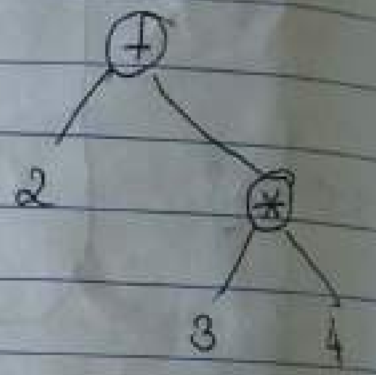
A notated parse tree

$E \rightarrow E + T$	$\{E.nptr = mknode(E.nptr, +, T.nptr)\}$
$E \rightarrow T$	$\{E.nptr = T.nptr\}$
$T \rightarrow T * F$	$\{T.nptr = mknode(T.nptr, *, F.nptr)\}$
$T \rightarrow F$	$\{T.nptr = F.nptr\}$
$F \rightarrow id$	$\{F.nptr = mknode(Null, id, Null)\}$

$2 + 3 * 4$



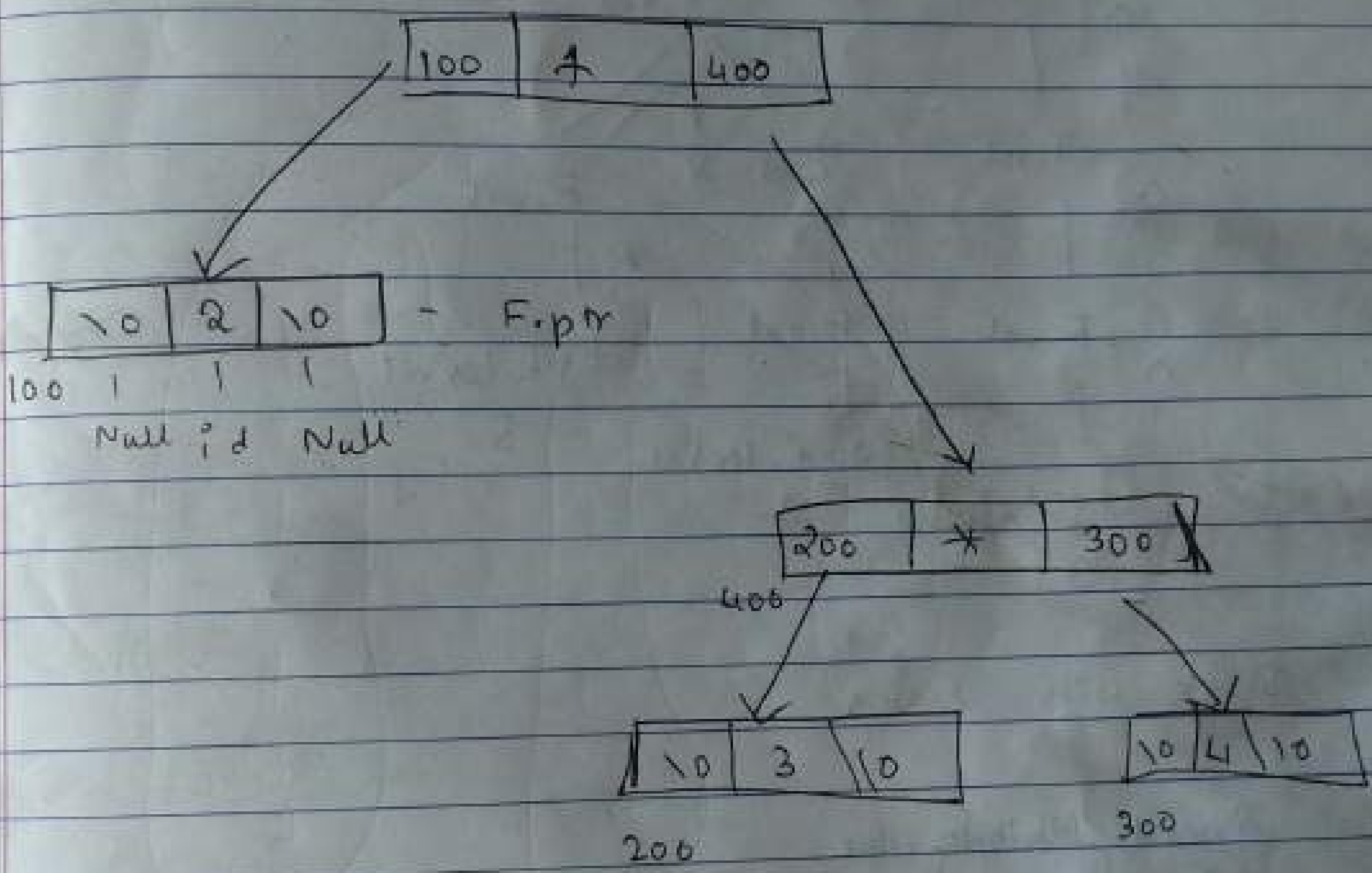
Concrete Syntax tree



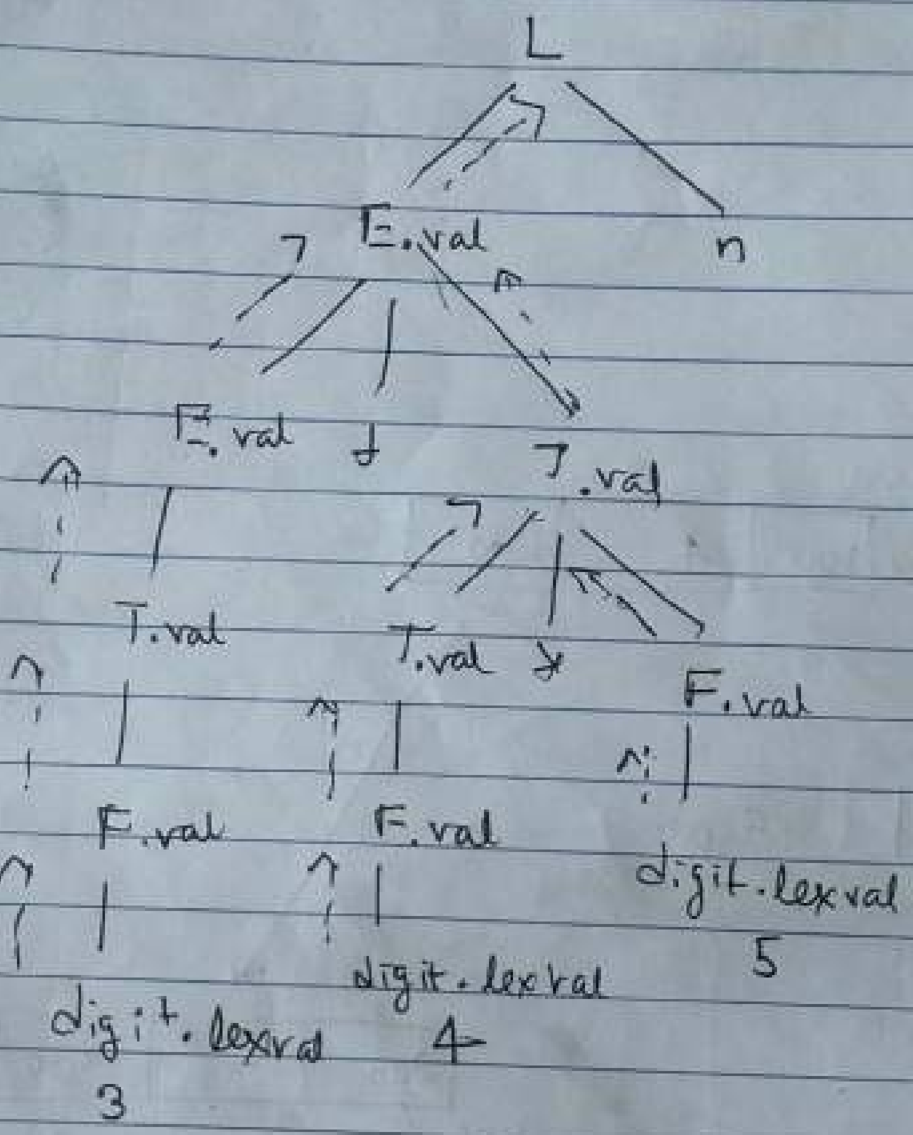
Abstract Syntax tree

T→F

F→id



100 | + | 400
10 | 2 | 10
- F.ptr
100
Null id Null
200 | * | 300
400
10 | 3 | 10
200
10 | 4 | 10
300



Stack with 2 array

Symbol	Attributes Value
E	

Input

3 + 4 * 5n

A → XYZ

RHS is reduced to LHS

& updated accordingly

L → En
E → E + T
E → T
T → T * F
T → F
F → (E)
F → digit

I/P	Stack	Reducing Production used
3+4*5n	—	—
+4*5n	3 | –	
+4*5n	F | 3	F → digit
+4*5n	T | 3	T → F
+4*5n	E | 3	E → T
4*5n	+ | – E | 3	—
*5n	4 | – + | – E | 3	—
*5n	F | 4 + | – E | 3	F → digit
*5n	T | 4 + | – E | 3	T → F
5n	* | – T | 4 + | – E | 3	—

i/p | Stack | Production Used

n

5	-
*	-
T	4
+	-
F	3

n

F	5
*	-
T	4
+	-
E	3

F → digit

n

	-
T	20
+	-
E	3

T → T*F

n

E	23

E → E+T

—

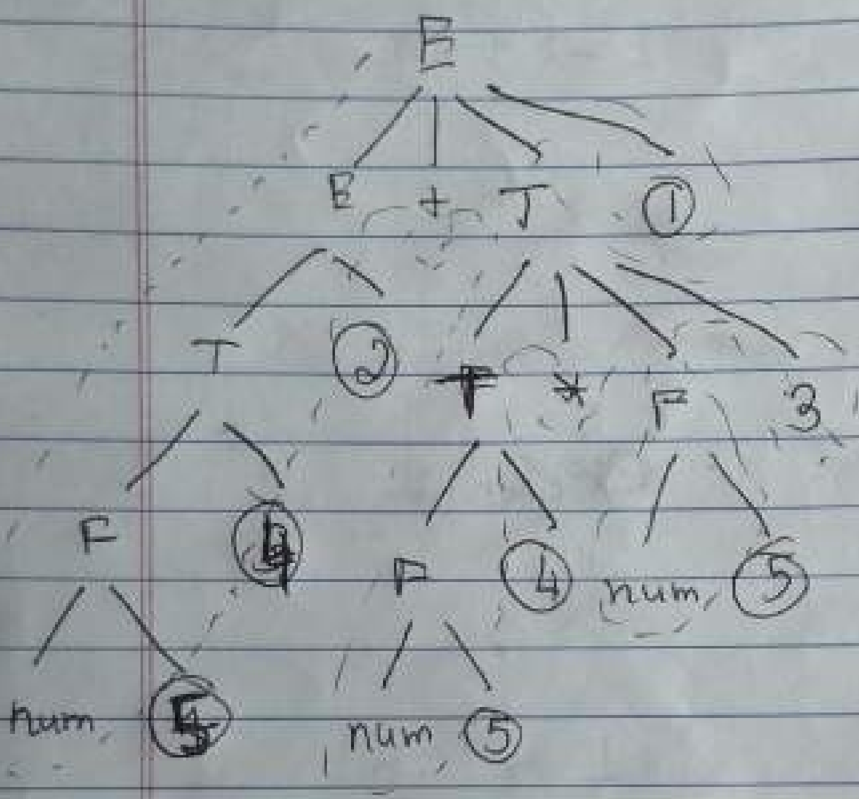
n	-
E	23

~~[illegible]~~ -

—

L	23

L → En



E → E+T {1}

E → T {1}

T → T*F {2}

T → F {2}

F → num {3}

num + num * num

i/p

op

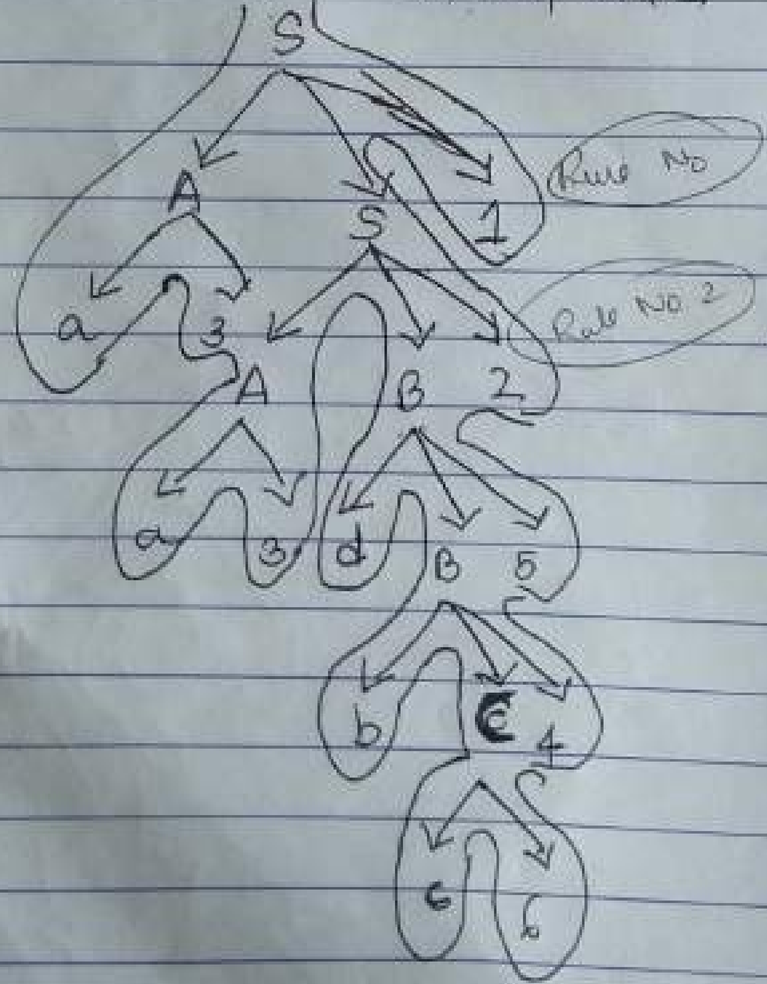
Syntax Directed Translations

Top down

$S \rightarrow AS$	{ printf ("1"); }
$S \rightarrow AB$	{ printf ("2"); }
$A \rightarrow a$	{ printf ("3"); }
$B \rightarrow b\epsilon$	{ printf ("4"); }
$B \rightarrow dB$	{ printf ("5"); }
$C \rightarrow c$	{ printf ("6"); }

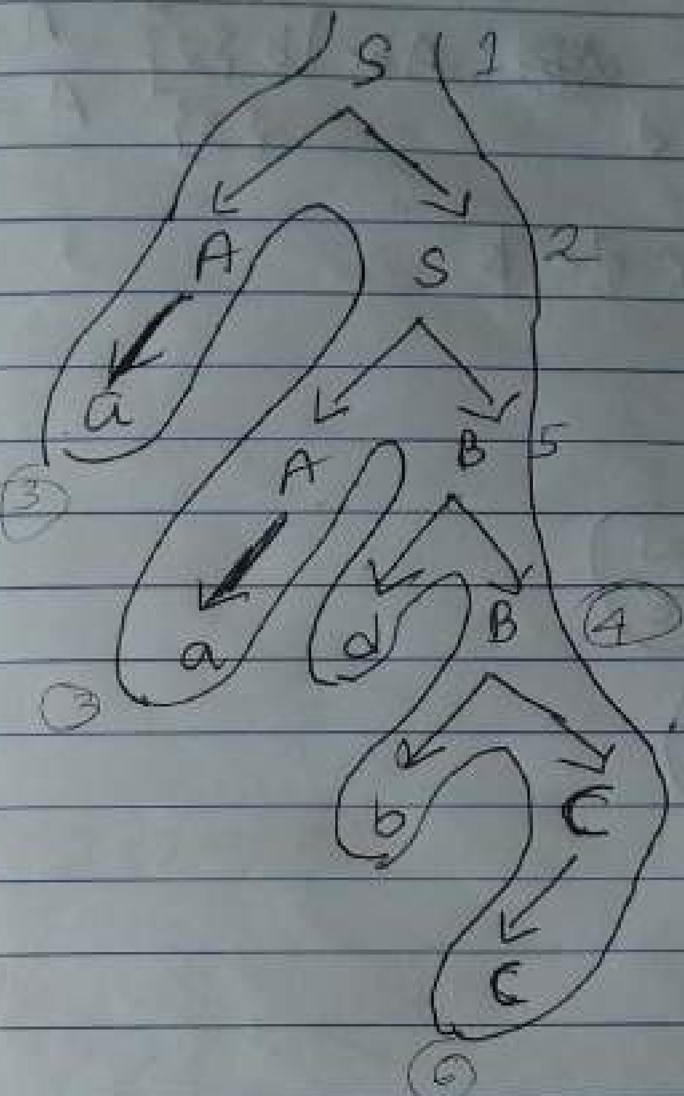
I/p = aadbc

O/p 3364521



Top to down

Left to down



Reduce

reduction

~~[illegible]~~ ~~[illegible]~~

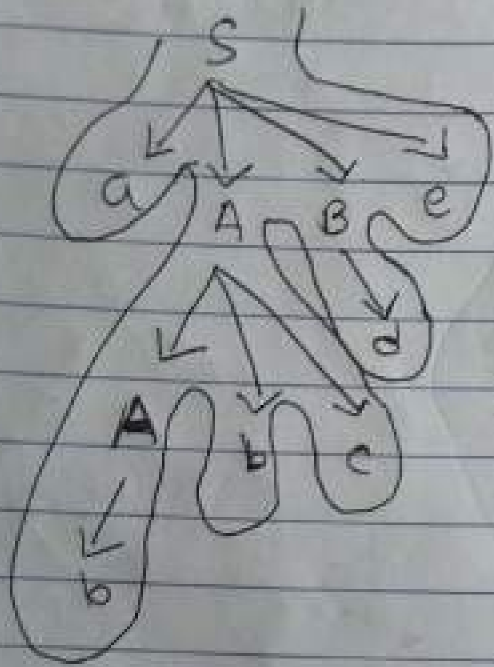
3 3 6 4 5 2 1

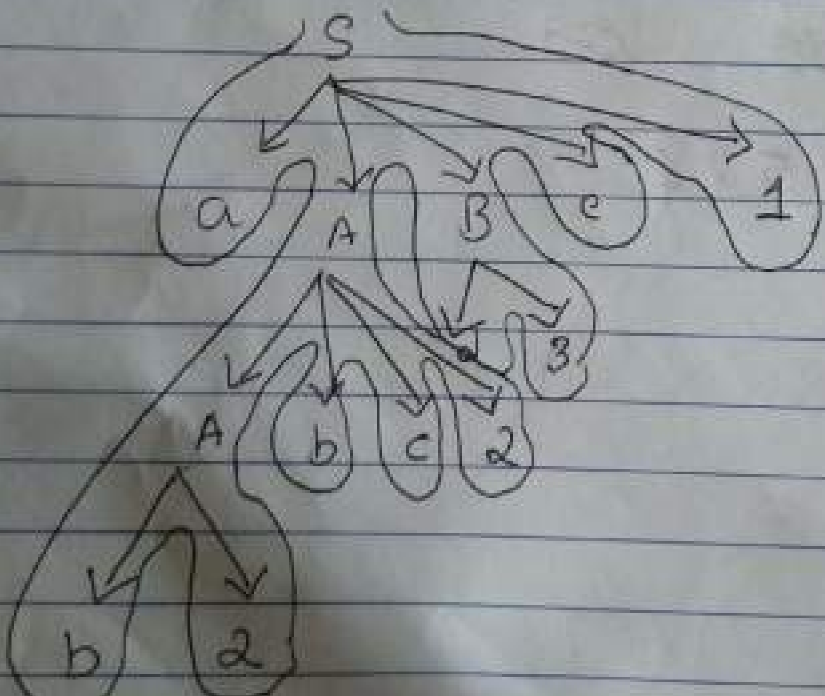
S → aABe {1}

A → Abc | b {2}   A → Abc

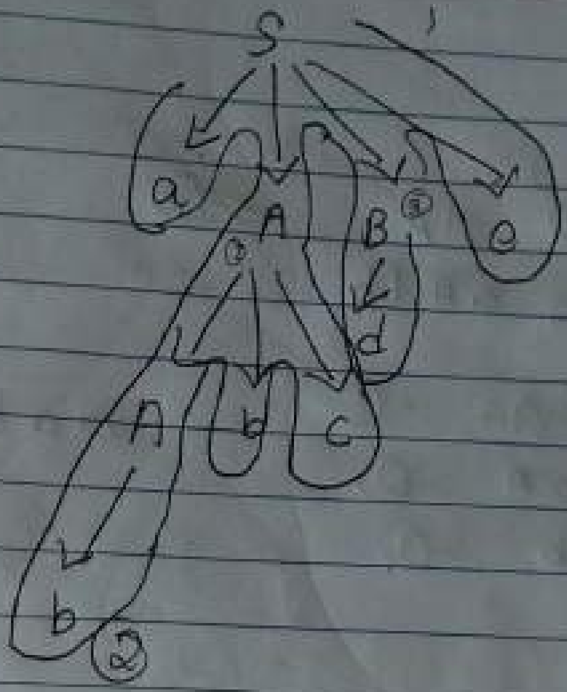
B → d {3}   A → b

a bb c de $

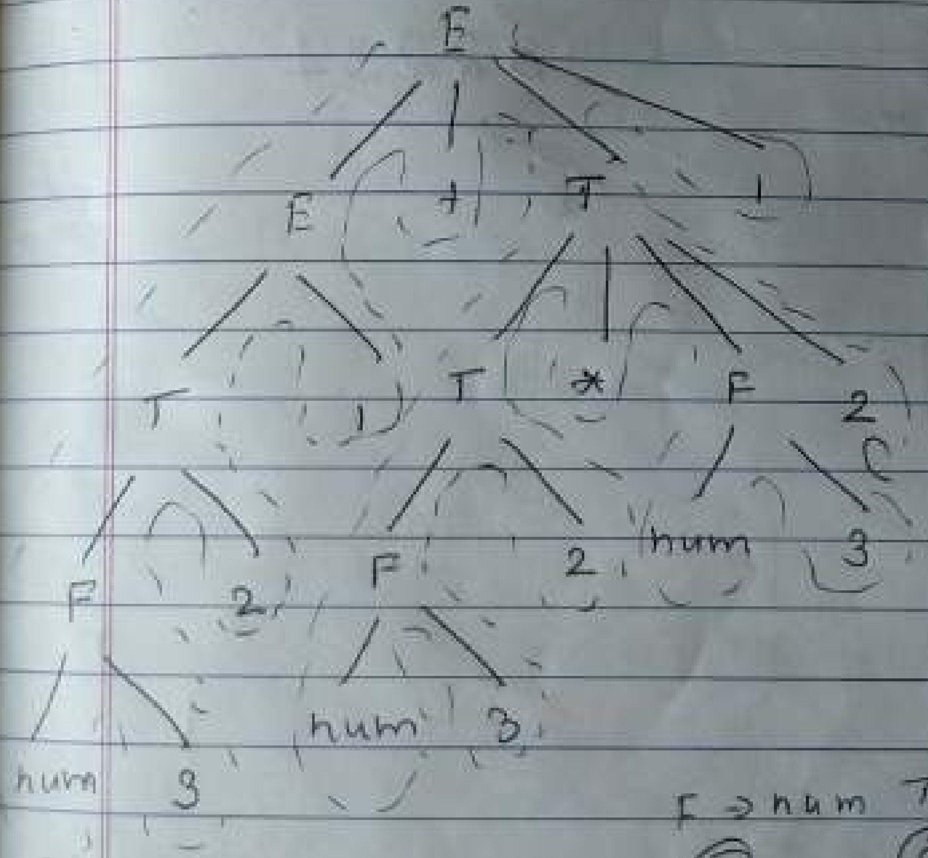




2 2 3 1



2231



i/p: num+num*num

o/p: 32132321

F → num   T → F   E → T   F → num

3   2   1   3

T → F   F → num   T → T*F   E → E+T

2   3   2   1

$S \rightarrow AA$ — 1

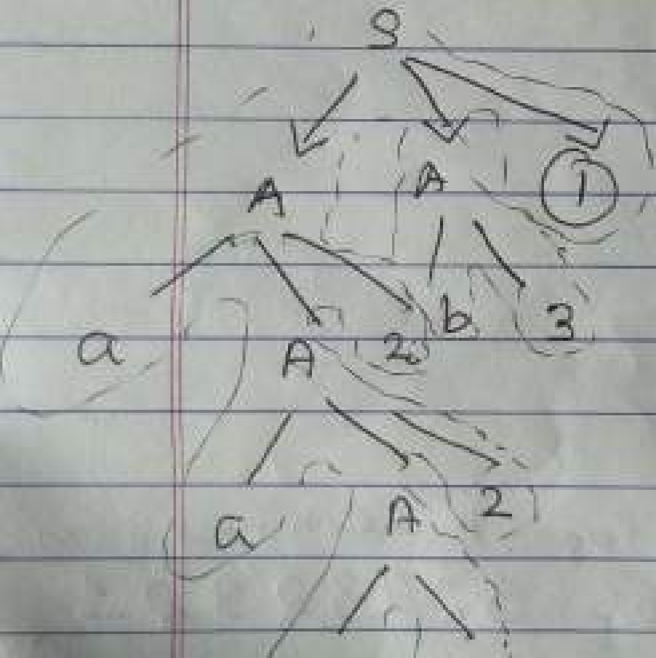
$A \rightarrow aA \mid b$

$S \rightarrow AA$ — 1

$A \rightarrow aA$ — 2

$A \rightarrow b$ — 3

aabb



3 2 2 3 1

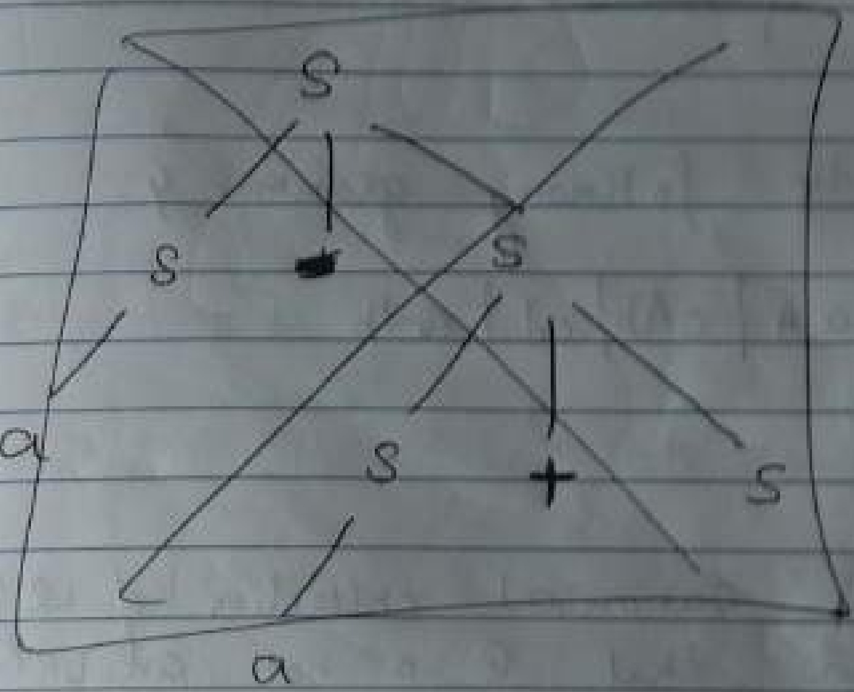
b

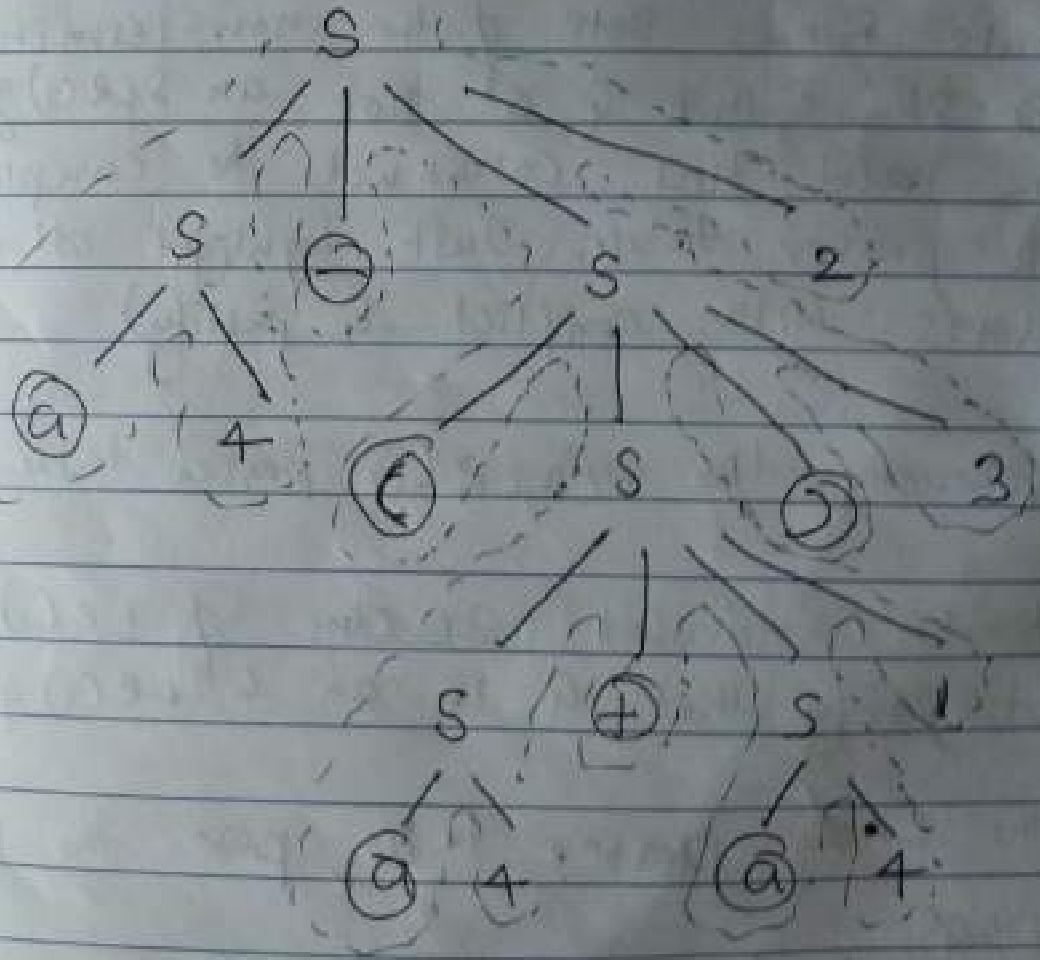
$S \rightarrow S + S$ — 1

$S \rightarrow S - S$ — 2   i/p $a_1 - (a_2 + a_3)$

$S \rightarrow (S)$ — 3

$S \rightarrow a$ — 4   o/p 444132





Consider the following grammar G

$S \rightarrow aA \mid cAb \mid cd \mid adb$
$A \rightarrow d$

(i) Construct canonical collection of LR(0) items and show that G is not an LR(0) grammar

(ii) Compute follow sets of the non-terminals & show that G is not an SLR(1) grammar (You need not construct the complete SLR parser table. Just highlight all the states(s with conflict & justify)

(iii) Construct the LALR(1) parser table for G.

(iv) Construct Canonical collection of LR(1) items and justify that G is an LALR(1) grammar

(v) Use the LALR parser table, parse the following strings

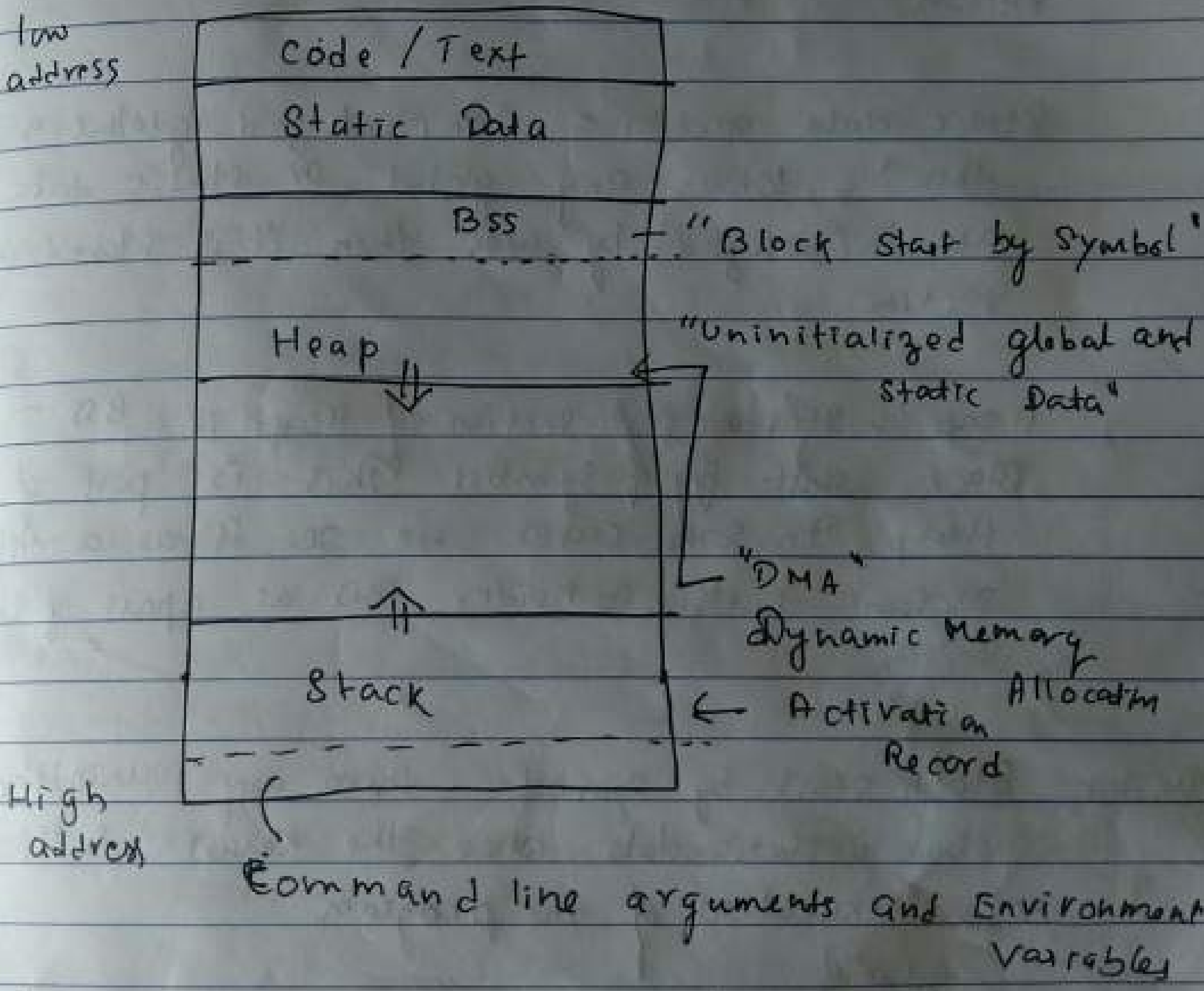
(a) cdb

(b) ad

Run time Environment of a C-Program

When a program is under a execution then we say it as a process.

Consider the process structure



A run time process structure has code/text section, static data section, Heap and Stack

Code/Text - This is a read only section where you find a executable code

It is read only because the instruction do not get modified at any point of time

Thus we say this as code section or Text data section

Static data section - "Initialized global or static data", When any global or static data that are initialized by user then it is stored in this section

Heap - Starting of section of Heap is BSS - Block start by Symbol. This is part of Heap. In some cases we see it as a different section. Let's consider this as apart of the section.

Under Block start by symbol there are uninitialized global & static data where the kernel initializes before execution of the program.

Thus, value of extern and static value by default is zero.

Because when this is utilized, the kernel would have already initialized extern & static value from 0 to 1.

The next section in Heap is DMA which is "Dynamic Memory Allocation".

Under this we have malloc, calloc etc.

Next is Stack, the bottom most of stack has command line arguments and environment variables

In command line argument we have

main (int argc, char *argv)

and other envt variables.

Next, all the variables in the function except static will be there in the stack.

We tell the stack an Activation Record

Run time Environment

If we take any program c Program, Java program then the program is stored in the hard disk During compilation also ~~a~~ the program resides in harddisk.

But ~~during~~ ~~~~ a CPU or processor executes a program only when the program resides in main memory

If the program resides in main memory then only the processor executes can execute the program

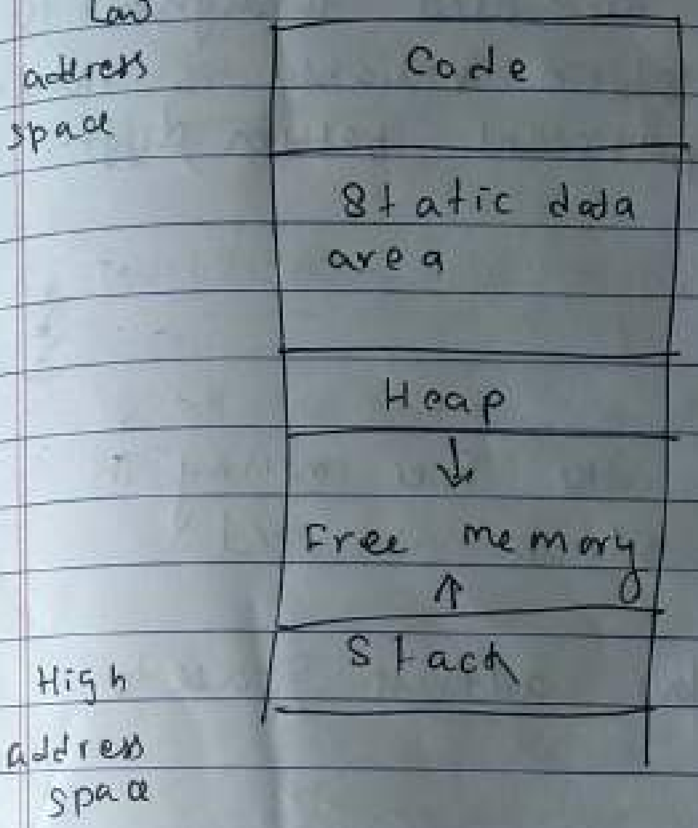
Block of program is compiled by the compiler Now after compilation is over the compiler takes a block of memory from the operating System in order to store it in the main memory

So, OS allocates a free block of memory to ~~of~~ the corresponding program.

Compiler uses the free block of memory in order to store the compiled program

So this is called Run time Storage Management

Here we can divide the main memory into 4 parts



During the compilation the size of the program will be decided.

This code area mainly stores the executable code.

After compilation there is linking, linking means it links object code of several library files into a single file.

The code portion mainly contain .exe (that is nothing but the executable code)

Static data area - it is used to store the static variables & global variables.

[The global variables are used throughout the program whereas static variables is value of the variable is persistent between diff function calls]

Heap & Stack

In order to utilize the free memory in effective Heap & stack are used

Stack grows from low address space to high address space

whereas heap grows from high address space to low address space.

~~Advantages~~

Stack is LIFO, where inorder to store activation record. stack is req

So whenever a function or procedure call occurs then activation record ~~[illegible]~~ gets created

Model of Activation Record

or

Fields of Activation Record

Actual Parameters
Returned values
Control or dynamic link
Access or static link
Saved Machine Status
Local variables
Temporary variables

(1) **Actual Parameters** :- It holds the actual parameters of the calling function

actual parameters the parameters declared inside the calling function

Eg: func1 (12, 23) here 12 & 23 are actual parameter

(2) **Returned values** :- It is used to store the result of function call.

Eg:- a = sum(); - function call

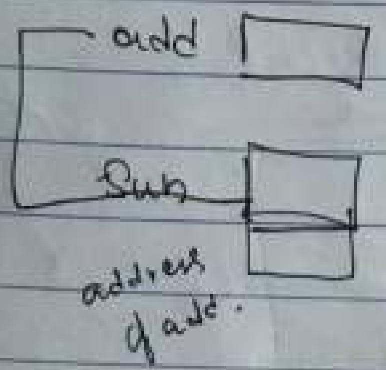
int sum() - function defn

{

}

Control link :- It points to the Activation record of calling function

```
add()          - calling function
{
  sub();       - called function
}
```



function activation record

function activation record

Access link :- It refers to the local data of called function but found in another Activation record.

```
int g = 12;

void A()
{
  printf("%d", g);
}

void main()
{
  A();
}
```

Saved Machine Status :-

Stores address of next instruction to be executed

Local variable There variables are local to a function

Temporary variable :- Needed during expression evaluation

Heap - In clang,

malloc
calloc
realloc
& free function by using these 4 function we can allocate & deallocate the memory at run time.

If there are too many functions stack is used
If there are too many dynamic or pointer variables then heap is used.

Stack is ~~..~~ stored at high addresses
Heap is stored at low addresses.

Storage organization
or
Storage Allocation Strategies

There are 3 types of ~~storage~~ of Storage allocation available:

1. Static allocation - Static memory allocation
2. Stack allocation } Dynamic
3. Heap allocation } Allocation

1. Static Allocation or Static Memory Allocation

The allocation of memory during compilation time is known as Static memory allocation. for ordinary variables or for array.

Drawbacks of static Memory Allocation

1. We must know in advance the size of the array.

Let us suppose the array size is a[5] for 6 elements and perform the operations of 10 elements then the size of array is a[10]

2. If more memory is allocated then the memory will be wasted.

Let us suppose the memory is allocated for 100 elements. , whereas we use it for 5 elements

ie., a[100]
↓ only
a[5] is used

That means a[95] - 95 elements is wasted.

③ If less memory is allocated than required then it causes the problems because it is not possible to perform the corresponding operations.

Let us assume that memory is allocated for 5 elements whereas it performs operations on 10 elements

a [5]
memory

a [10]
operations performed

Here we can perform operations for 1st 5 elements only not for the next 5 elements (6 - 10]

④ Insertion & deletion operations are very very expensive.

Consider elements :

10 20 30 40 50

I want to add element 100

Then for that I need to move all the element one position to the right

10 20 30 40 50

100 shift right shift right

100 10 20 30 40 50

to accomodate this

Let us assume the Array contains 1 lakh elements then 1 lakh shifting is required.

Like that deletion of the element is very very expensive.

Let us suppose,

10 20 30 40 50

and we want to delete first element 10

10 20 30 40 50

20 30 40 50

So we will have to move each elements to the left.

Same in this case if there are 1 lakh elements then 99,999 shiftings are needed.

So this is very very expensive

Thus we use Dynamic memory Allocation, which is nothing but memory allocation done during runtime the memory will be allocated for the corresponding variables

Dynamic memory Allocation

① Stack allocation

Here stack is a Data Structure which works in the form of LIFO (Last In First Out) i.e, last inserted can be deleted first.

Whenever a function or procedure call occurs then an activation record will be created for the corresponding function.

One activation record will be pushed up to the top of the stack.

Here we are using stack mainly for storing procedure or function information. That information resides in the activation record which is pushed on to the top of the stack.

If there are 5 functions in the ~~program~~ program then 5 activation records will be created for the corresponding 5 functions

[What is activation record? Already explained in last class.]

Drawbacks of Stack Allocation

1. It supports dynamic memory allocation but slower than static allocation.

2. The stack allocation supports recursion but references to non local variables after activation records can't be retained.

In order to overcome this problem we used Heap allocation

Heap Allocation

So we mainly use Heap in order to ~~[illegible]~~ increment Dynamic Memory Allocation

If we use C lang than we can use dynamic memory allocation functions like: malloc, calloc, realloc, free functions

By using those functions we can allocate the memory as well as we can change the size of the already allocated memory block and also release the memory block

In Java, we have new operator & delete operator so by using those 2 functions we can allocate the memory as well as we can deallocate the memory.