Q2. Conditions for a grammar to be LL(1)

(i) If grammar is free from ε productions, then

G is LL(1) if $\forall A \to \alpha_1 | \alpha_2 | \ldots$

First(α_1) ∩ first(α_2) ∩ first(α_3) ... = ϕ

(ii) If G has ε productions

$\forall A \to \alpha | \varepsilon$

if first(α) ∩ follows(A) = ϕ

then given grammar is LL(1)

(iii) If G has only one alternative on RHS, ∀ variables, then G is LL(1)

Eg. $A \to \alpha$

$B \to \gamma$

$C \to \beta$

→ First(X) for a grammer symbol X is the set of terminals that begins the strings derivable from X.

→ Follow(X) to be the set of terminals that can appear immediately to the right of non-terminal X in some sentential form.

We are given following production rules

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow id \mid (E)$

FIRST SET

First (E) = { (, id }
First (E') = { +, ε }
First (T) = { (, id }
First (T') = { *, ε }
First (F) = { (, id }

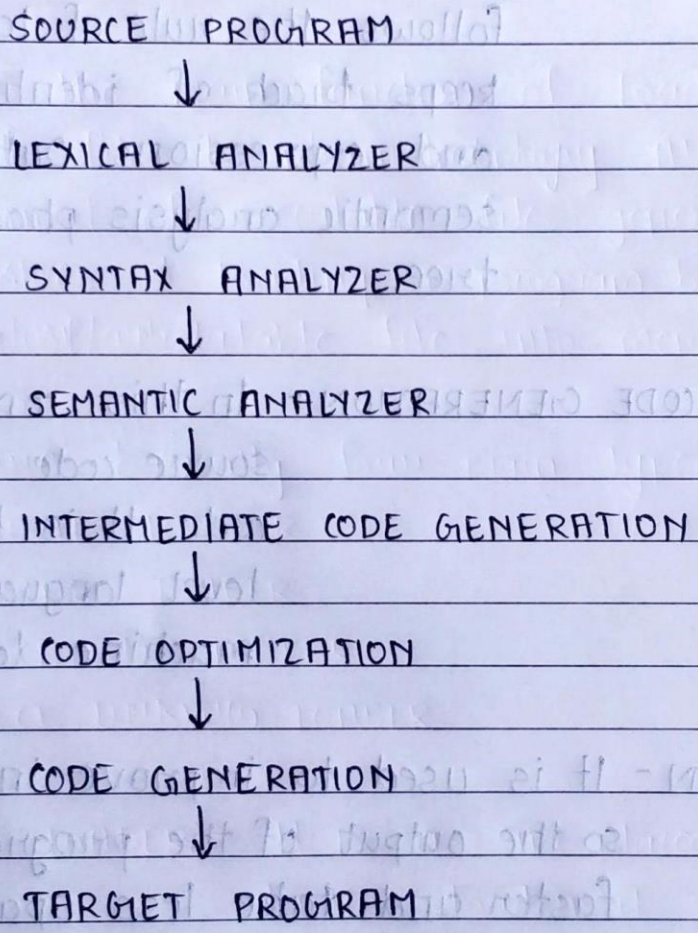
Here First (E) = First (T) = First (F)

FOLLOW SET

Follow (E) = { $,) }
Follow (E') = Follow (E) = { $,) }
Follow (T) = { First (E') − ε } ∪ Follow (E)
= { +, $,) }
Follow (T') = Follow (T) = { +, $,) }
Follow (F) = { First (T') − ε } ∪ Follow (T)
= { *, +, $,) }

Q4. Compilation process contains the sequence of various phases. Each phase take source program in one representation and ~~procedures~~ produces output in another representation.

Various phases of compiler are as follows :



(1) LEXICAL ANALYSIS - It is the first phase of compilation process. It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexems.

(II) SYNTAX ANALYSIS - It takes token as input and generates tree as output. In this phase, the parser checks that the expression made by tokens is syntactically correct or not.

(III) SEMANTIC ANALYSIS - It checks whether the parse tree follows the rule of language. It keeps track of identifiers, their and expressions. Output of semantic analysis phase is annoted tree.

(IV) INTERMEDIATE CODE GENERATION - In this, compiler converts source code into intermediat code. It is between high-level language & the machine language

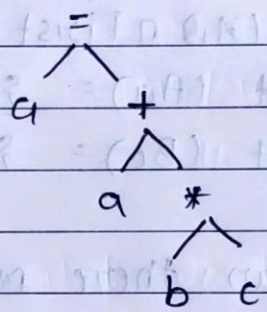
(V) CODE OPTIMIZATION - It is used to improve intermediate code so the output of the program could run faster and take less space. It removes the unnecessary lines of code & arrange the statement to speed up the process.

(VI) CODE GENERATION - This is final stage of the compilation process. It takes the optimized intermediate code as inputs & maps it to the target machine language. It translates the intermediate code to machine code.

Examples : a := a+b * c ;

→ LEXICAL ANALYSIS - a,b,c are identifiers
:= assignment operator
+ operator
* operation
; special symbol

→ SYNTAX ANALYSIS :



→ SEMANTIC ANALYSIS : It first checks whether the generated tree follows rules.

→ INTERMEDIATE CODE GENERATION : a= a+b+c

id_1 = id_2 id_3

$t_1 = id_2 * id_3$

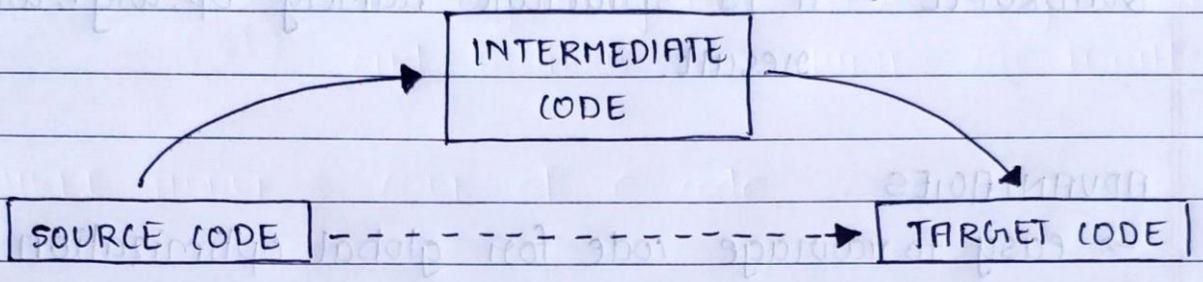
$id_1 = id_1 + t_1$

→ CODE OPTIMIZATION : a = a+b → a+=b

→ CODE GENERATON :

```
MOV  id3, R1
MOV  id2, R2
MOV  R2, R1
MOV  id1, R2
ADD  R2, R1
MOV  R1, id1
```

Q5 INTERMEDIATE CODE - It is machine independent code, but it is close to machine instructions.



Need for intermediate code generation phase is :

↳ RE-TARGETING IS FACILITATED - a compiler for different machine can be created by attacking back end for the new machine to an existing front end

↳ MACHINE INDEPENDENT - a machine independent code optimizer can be applied to intermediate code so that it can be run on any machine.

↳ SIMPLICITY - Intermediate code is simple enough to be easily converted to any target code. So it reduces overhead of target code generation.

↳ COMPLEXITY - It is complex enough to represent all the known complex structure of high-level languages.

↳ MODIFICATION - We can easily modify our code to get better performance and throughput by applying optimization technique to the intermediate code.

Two techiques to implement three address code are :-

1. QUADRUPLE - It is structure namely op, arg1, arg2 and result.

ADVANTAGES
↳ easy to rearrage code for global optimization
↳ one quickly access value of temporary variables using symbol table.

DISADVANTAGES
↳ contains lot of temporaries
↳ It increases time and space complexity

2. TRIPLES - It doesn't make use of extra temporary variable to represent single operation; instead a pointer to the triple is used. So it consist only 3 fields namely op, arg1, arg2

ADVANTAGE
↳ It requires less space

DISADVANTAGE
↳ temporaries are implicit and difficult to rearrange code.
↳ It is difficult to optimize because it involves moving intermediate code.

Q6. G → E

E → E+T

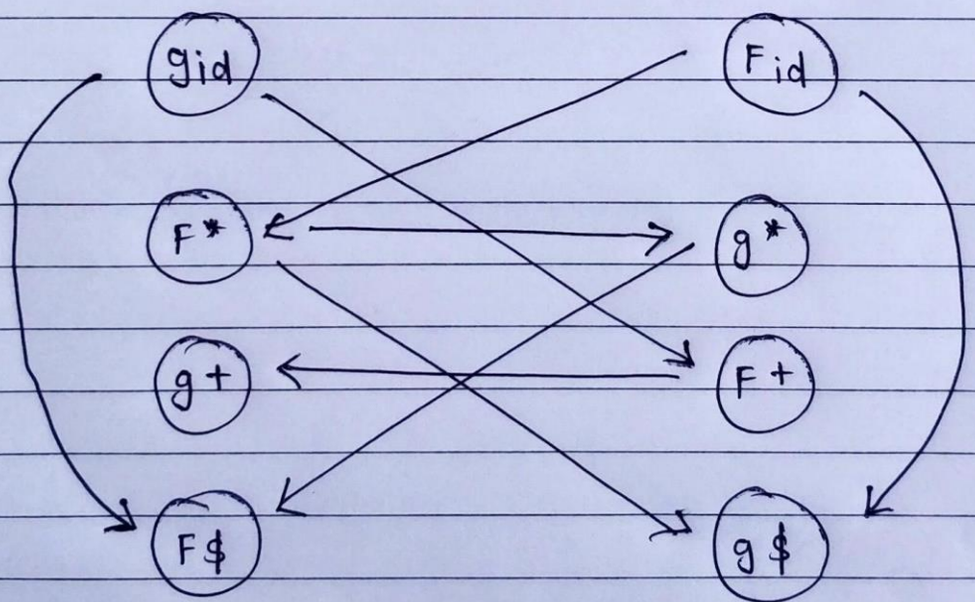
T → T*F

F → id

OPERATOR PRECEDENCE TABLE

	+	*	id	$
+	·>	<·	<·	·>
*	·>	<·	<·	·>
id	·>	·>	–	·>
$	<·	<·	<·	–

PRECEDENCE RELATIONAL GRAPH



Q8. LINKER AND LOADER

→ The main function of linker is to generate executable files. The linker takes input of object code generated by compiler / assembler. It is a tool that merges the object files produced by separate compilation or assembly and creates an executable file.

→ Whereas the main function of loader is to load executable files to main memory. Also the loader takes input of executable files generated by linker. It involves loading a program by reading the contents of executable file into memory

There are basically four main types of loader
- ABSOLUTE LOADER
- BOOTSTAP LOADER
- RELOCATING LOADER
- DIRECT LINKING LOADER

(1) ABSOLUTE LOADER - It is a sim placed type of loader. In this loader it simply accepts the machine language code produced by assembler and place it into main memory at the location specified by assembler.

(2) BOOTSRAP LOADER - When a computer is first turned on or restarted a special type of absolute loader is executed, called bootstrap loader.

It loads the operation system into the main memory and executes related program. It is added to the beginning of all object programs that are to be atleast loaded into empty ideal system.

(3) RELOCATING LOADER - to avoid possible reassembling of all sub-routines, it is changed to perform the task of allocation and linking of programmers. so relocating loader was introduced. The execution of the object programming is done using any part of available and sufficient memory.

(4) DIRECT LINKING LOADER - It is a general relocatable loader and most popular loading scan presently used. It allows programming multiple procedure segments and multiple data segments. It gives programmer complete freedom in refrencing data or instructions content in other segments.

Q9. $S \to Aa \mid bAc \mid Bc \mid bBa$
$A \to d$
$B \to d$

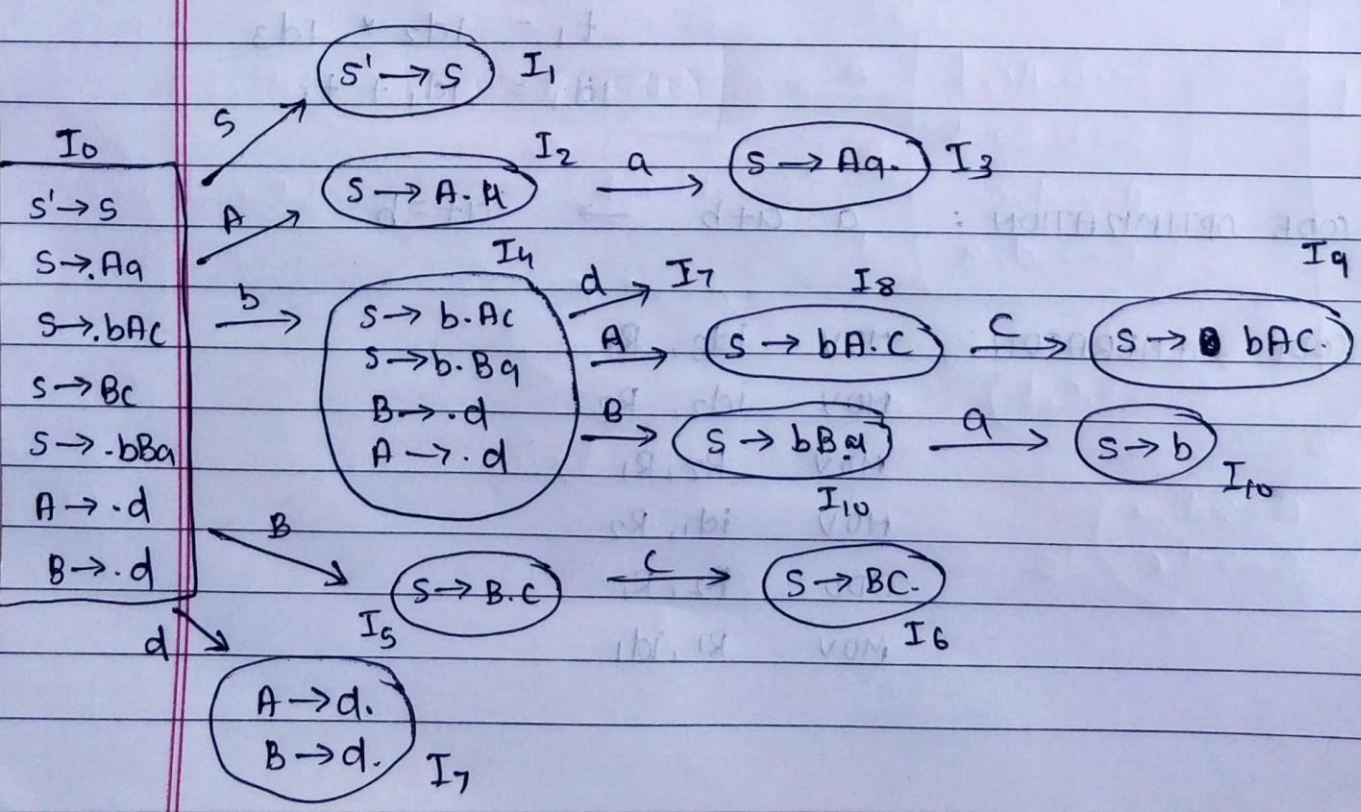
LL(1) grammar should be unambiguous + left factured and no left recursive
If G is free from ε production rules then to be a LL(1) grammar $\forall A \to \alpha_1 \mid \alpha_2 \mid \cdots$

$$First(\alpha_1) \cap First(\alpha_2) \cdots = \phi$$

Here, $First(Aa) = \{d\}$
$First(Bc) = \{d\}$

For SLR(1) grammar there must not be an SR or RR conflicts in table.

G' : $S' \to S$
$S \to Aa \mid \cdot bAc \mid \cdot Bc \mid \cdot Ba$
$A \to d$
$B \to d$

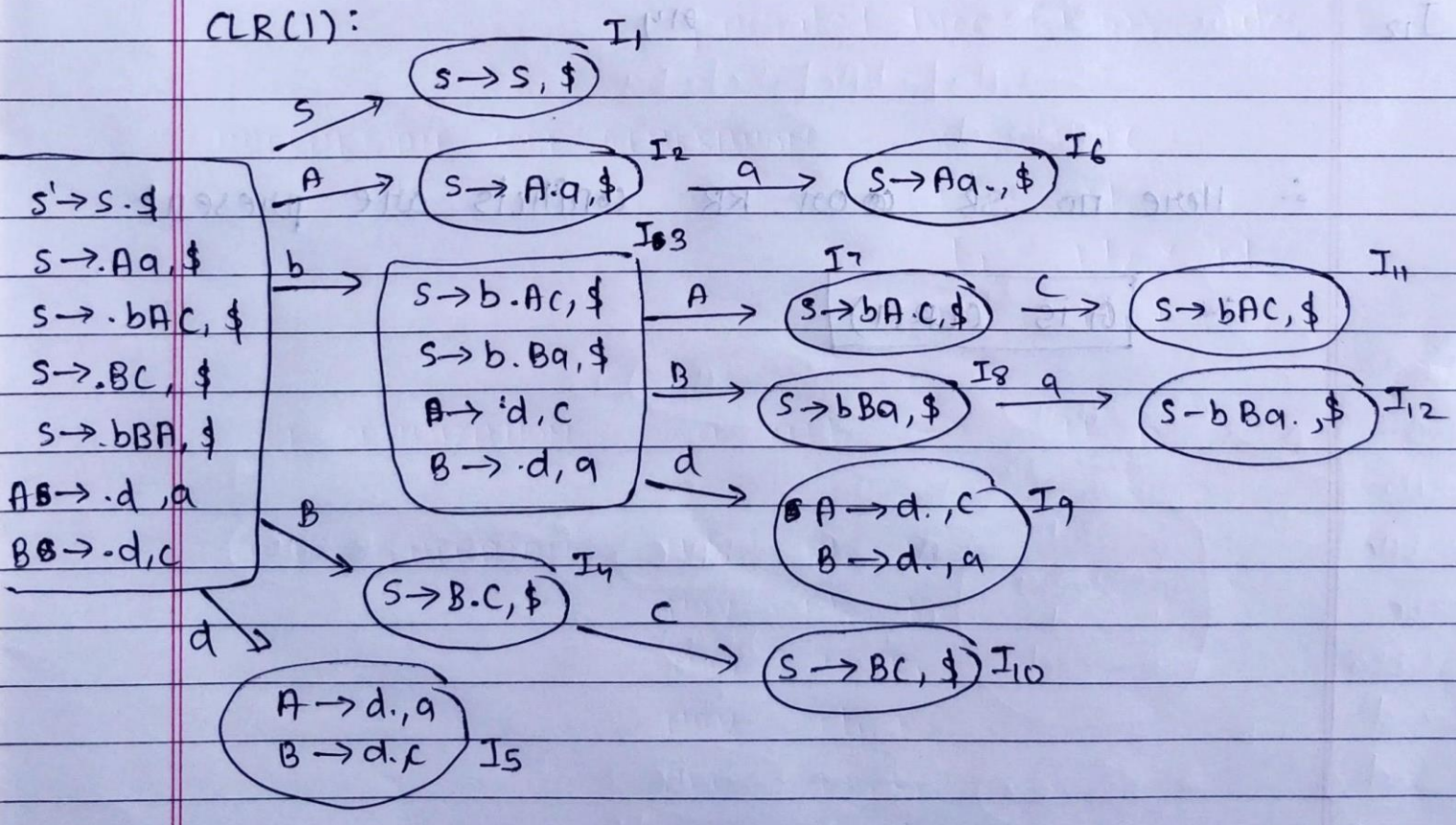


Parse table for SLR(1):

	a	b	c	d	$	S	A	B
I_0		S_4				1	2	5
I_1					Accept			
I_2	S_3							
I_3					r_1			
I_4				S_7			8	10
I_5			S_6					
I_6					r_3			
I_7					r_5/r_6			
I_8								
I_9								
I_{10}								
I_{11}								

Here, for I_7 at $ we have 'RR conflict', so G is not SLR(1)

CLR(1):



parse table for CLR(1):

	a	b	c	d	$	S	A	B
I_0		S3		S4		1	2	4
I_1					Accept			
I_2	S6							
I_3				S9			7	8
I_4			S10					
I_5	r5		r6					
I_6								
I_7			S11					
I_8	S10							
I_9	r6		r5					
I_{10}					r3			
I_{11}					r2			
I_{12}					r4			

∴ Here no SR or RR conflicts are present

∴ **G is CLR(1)**

Q10. Goals of an error handler in a parser are as follows

→ Error Detection
→ Report the presence of errors clearly and accurately
→ Recover from each error quickly enough to detect subsequent errors
→ Add minimal overhead to the processing of correct programs

Different error strategies used by a parser from a syntatic error :

1. PANIC MODE RECOVERY
→ In this method, succesive characters from the input are removed one at a time until a designated set of synchronizing tokens is found.
→ Advantage is that its easy to implement and guarantees not to go into infinite loop
→ Disadvantage is that a considerable amount of input is skipped without checking it for errors.

2. STATEMENT RECOVERY MODE
→ In this method, when a parser encounters an error, it performs the necessary correction on the remaining inputs so that rest of input statement allows parser to parse ahead
→ the correction can be deletion of extra semicolons, replacing the comma with semicolons, or inserting a missing semicolon.

→ while performing corrections, utmost care should be taken for not going in an infinite loop
→ A disadvantage is that it finds it difficult to handle situations where the actual error occurred before pointing of detection

3. ERROR PRODUCTION
→ If a user has knowledge of common errors that can be ~~generated~~ encountered then, these errors can be incorporated by augmented the grammar with error productions
→ If this is used then, during parsing appropriate error messages can be generated and parsing can be continued.
→ The disadvantage is that it's difficult to maintain.

4. GLOBAL CORRECTION
→ The parser examines the whole program and tries to find out the closest match for it which is error-free.
→ The closest match program has less number of insertions, deletions, and changes of tokens to recover from erroneous input.
→ Due to high time and space complexity, this method is not implemented practically.

Q11. BASIC BLOCK - It is a sequence of three address statements where control enters at the beginning and leaves only at the end without any jumps or haults.

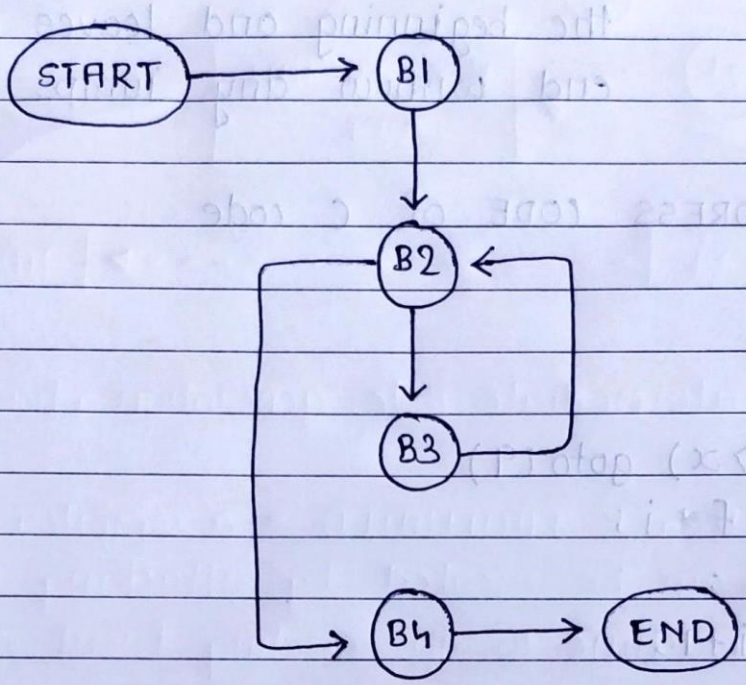
THREE ADDRESS CODE of C code

```
1. f=1;
2. i=2;
3. if (i>x) goto (9)
4. t1 = f * i;
5. f = t1;
6. t2 = i+1;
7. i = t2;
8. goto (3)
9. goto calling program
```

BASIC BLOCKS

```
f = 1 ; *                          ] B1
i = 2 ;
if (i>x) goto (9) *                ] B2
t1 = f * i; *                      ]
f = t1 ;                           ]
t2 = i+1;                          ] B3
i = t2;                            ]
goto (3)                           ]
goto calling program *             ] B4
```

CONTROL FLOW GRAPH



Q12. The runtime environment - It is the environment in which a program or application is executed.

It's the hardware and software infrastructure of a particular codebase in real time.

Memory allocation done during compilation with block structured language is as follows:

The compiler resolves at compile time where certain things will be allocated inside the process memory map

for example, consider a global array
int array [100];

→ The compiler knows at the compile time the size of array and the size of int, so it knows the entire size of array at compile time.

→ Also a global variable has static storage duration by default, it is allocated in the static memory space.

→ The compiler decides during compilation in what address of that static memory area the array will be.

LIMITATION OF STACK BASED MEMORY ALLOCATION

→ Each process is allocated a fixed amount of stack space
→ Amount of memory available through stack is far less than memory available through heap
→ Once you are out of stack based available memory your process will crash
→ Stack memory is very limited
→ Random access is not possible
→ Creating too many objects can increase the risk of stack overflow.

Q15. Symbol Table is an important data structure created and maintained by the compiler in order to keep of semantics of variable i.e. it stores information, about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.-

Information stored in symbol table are:
- variable names and constants
- procedure and function names
- literal constants and strings
- compiler generated temporaries
- labels in source languages
- associated attributes with identifiers

Data structures used to implement symbol table are:

1 LIST - In this method, an array is used to store names and associated information
- implemented as array or linked list

PROs
↳ Insertion is fast O(1)
↳ It takes minimum amount of space

CONS
↳ Lookup is slow for large tables O(n)

2. BINARY SEARCH TREE - In this generally we add two link fields i.e. left and right child

PROS
↳ can grow dynamically

CONS
↳ insertion and lookup are $O(\log_2 n)$ on average

3. HASH TABLE - In this, two tables are maintained.
- A hash table is an array with an index range

PROS
↳ insertion and lookup can be made very fast O(1)
↳ quick to search is possible

CONS
↳ hashing is complicated to implement
↳ average time complexity O(n)

4. LINKED LIST - this implementation is using linked list by adding link field to each record

PROS
↳ insertion is fast O(1)

CONS
↳ lookup is slow for large tables - O(n) on average.