SR 22

St. Peter's Engineering College (Autonomous) Dullapally (P), Medchal, Hyderabad – 500100. QUESTION BANK								
						Dept.	:	CSE(AIML)
						Academic Year 2024-25		
Subject Code	:	AS22-66PC02	Subject	:	AUTOMATA THEORY & COMPILER DESIGN			
Class/Section	:	B. Tech.	Year	:	II	Semester	:	II

BLOOMS LEVEL					
Remember	L1	Understand	L2	Apply	L3
Analyze	L4	Evaluate	L5	Create	L6

Q. No	Question (s)	Marks	BL	CO
	UNIT – V			
1	Define Semantics.	1M	L1	C224.6
	Define Semantic Rules.	1M	L1	C224.6
	Define Attribute.	1M	L1	C224.6
	Define Actual Parameters.	1M	L1	C224.6
	Define Formal Parameters.	1M	L1	C224.6
2	Discuss about Syntax Directed Translation.	3M	L2	C224.6
	Discuss about Syntax Directed Definition.	3M	L2	C224.6
	Explain in detail about Types of Attributes.	3M	L4	C224.6
	Discuss about Symbol Table.	3M	L2	C224.6
	Explain in detail about Intermediate Code Generation.	3M	L4	C224.6
3	Discuss about Activation Record.	5M	L2	C224.6
	Distinguish between S-attributes and L-attributes.	5M	L2	C224.6
	Explain in detail about Dependency Graph.	5M	L4	C224.6
	Explain in detail about 3 Address Code Representation.	5M	L4	C224.6
	Explain in detail about Run-Time Environment.	5M	L4	C224.6
4	Explain in detail about Representation and Operations on Symbol Table	10M	L4	C224.6
	Explain in detail about Intermediate Code Generation and 3 Representations of Intermediate Code Generator.	10M	L4	C224.6
	Explain in detail about Storage Allocation.	10M	L4	C224.6

1.

Define Semantics:

Semantics refers to the meaning of a program written in a programming language. It defines how the syntactically correct statements of a language behave during execution.

Define Semantic Rules:

Semantic rules specify the meaning of syntactically valid constructs in a language. They ensure that the program has a well-defined behavior and adheres to logical constraints.

Define Attribute:

An attribute is a value associated with a grammar symbol in a syntax tree or parse tree. Attributes store information such as type, scope, or value of a variable and are used in syntax-directed translation.

Define Actual Parameters:

Actual parameters are the values or expressions passed to a function or procedure during a function call. They provide input to the function for processing.

Define Formal Parameters:

Formal parameters are the variables declared in a function or procedure definition that receive values from actual parameters when the function is called.

2.

Discuss about Syntax Directed Translation.

Syntax Directed Translation (SDT) is a method of translating a source program based on its syntax. It associates actions (semantic rules) with grammar productions, which are executed during parsing to generate intermediate code, construct syntax trees, or check semantic correctness.

- SDT uses Syntax Directed Definitions (SDD) or translation schemes to specify actions.
- The actions can be embedded within the grammar and executed during parsing (e.g., using an LR parser or recursive descent parser).
- It plays a crucial role in intermediate code generation, type checking, and semantic analysis.

Example: A syntax-directed translation scheme for arithmetic expressions:

mathematica

CopyEdit

$E \rightarrow E1 + T \quad \{ E.value = E1.value + T.value \}$

$E \rightarrow T \quad \{ E.value = T.value \}$

Here, the semantic rule specifies how the values of expressions are computed.

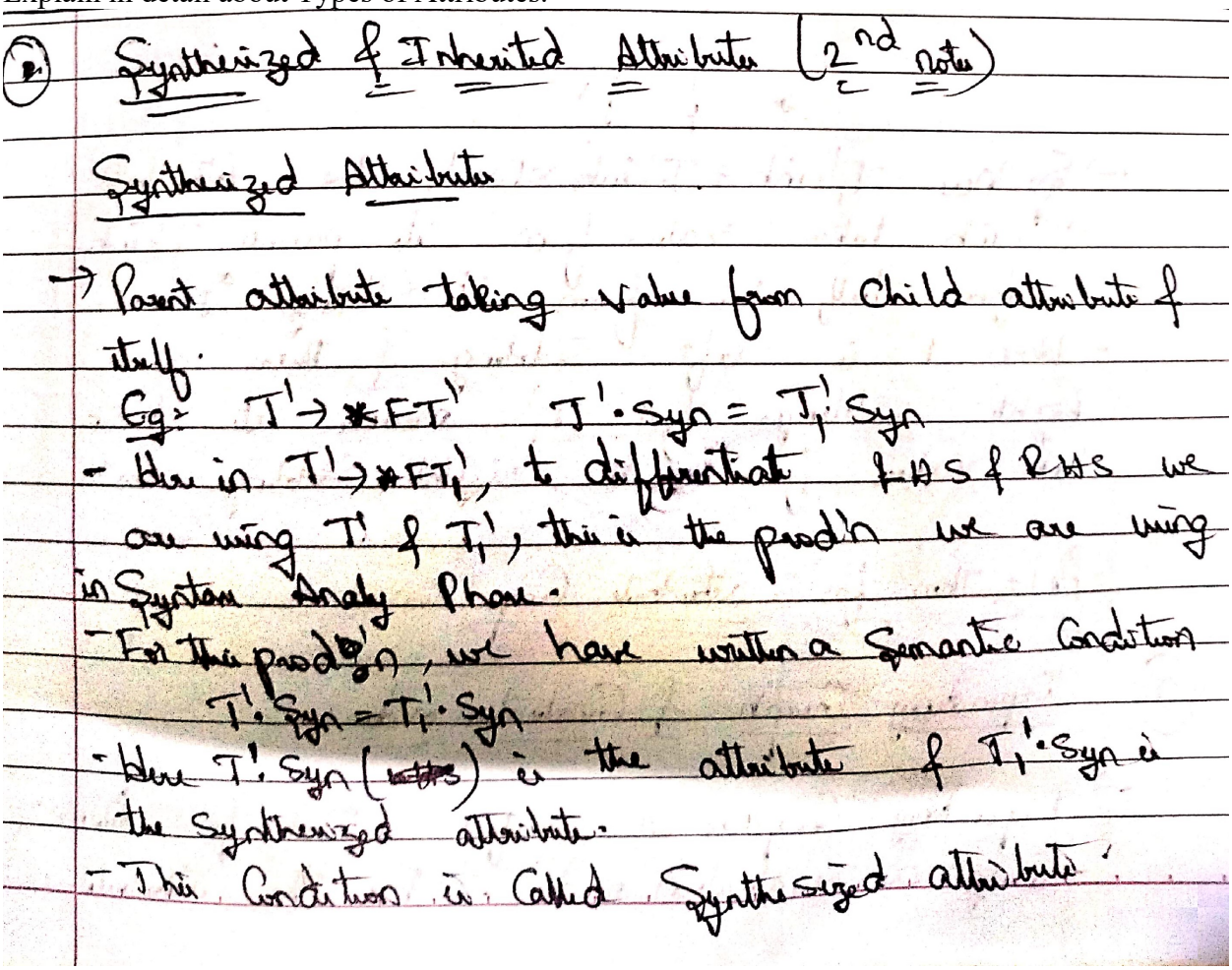
Discuss about Syntax Directed Definition (SDD)

Syntax Directed Definitions (SDD) extend context-free grammars by associating attributes with grammar symbols and defining semantic rules.

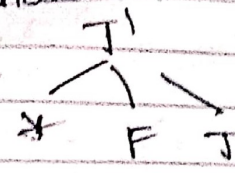
- Types of Attributes:
    1. Synthesized Attributes – Computed from the attributes of child nodes in a parse tree.
    2. Inherited Attributes – Derived from the attributes of parent or sibling nodes.
- Two types of SDD:
    1. S-attributed SDD – Uses only synthesized attributes and can be evaluated in a single bottom-up traversal.
    2. L-attributed SDD – Uses both synthesized and inherited attributes but follows left-to-right dependency rules.

Example of SDD:
kotlin
CopyEdit
E → E1 + T { E.val = E1.val + T.val }
E → T { E.val = T.val }
This SDD specifies how expressions are evaluated in a syntax-directed translation.

Explain in detail about Types of Attributes.



→ This can be expanded as



→ The parent attribute that takes values from its child (or) itself.

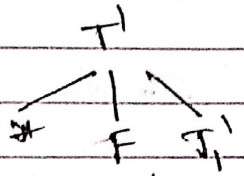
→ So here in the eg above eg, it takes values from its child.

Inherited Attribute

→ Attribute taking values from its parents, itself & its siblings.

- For eg the same eg:

$T' \rightarrow *FT'_1$ $\quad T'_1.inh = T'.inh * F.val$



→ So here $T'_1.inh = T'.inh * F.val$, the Attribute $T'_1.inh$ takes values from its parents $T'.inh$, from itself or from its siblings "F.val".

- Here F is left sibling & there are right siblings for T'_1 means it * can take values from the right siblings also.

Discuss about Symbol Table.

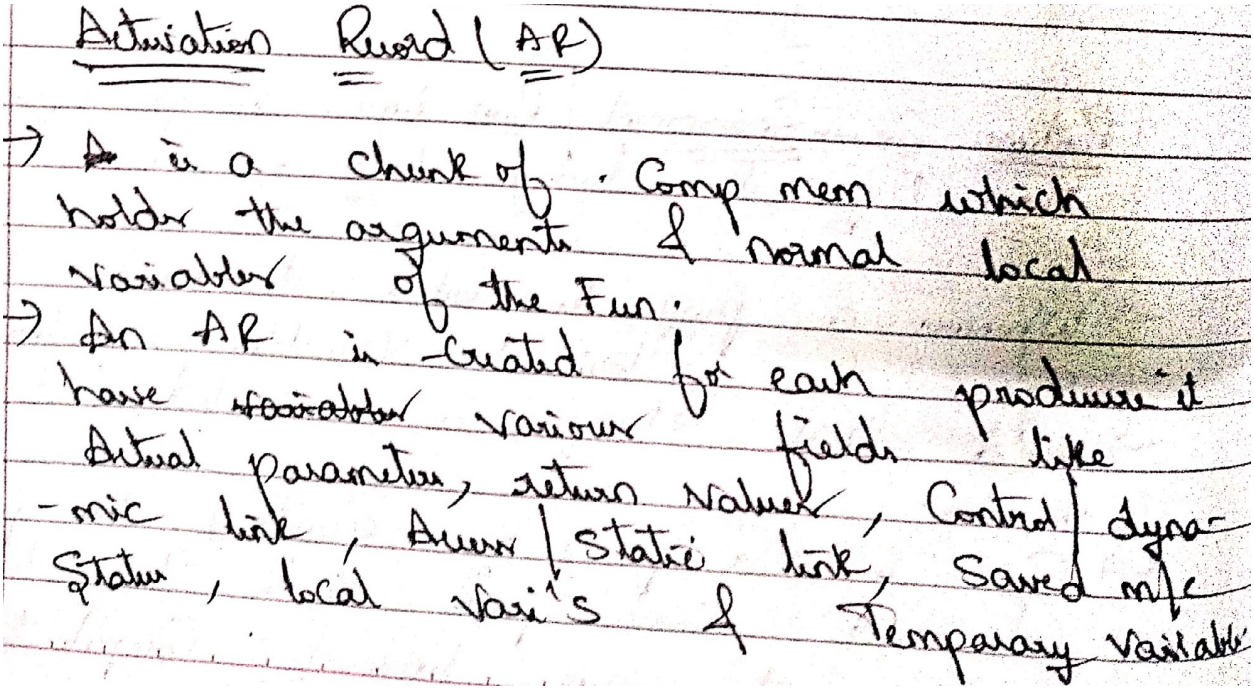
(Refer 4th Question Answers)

Explain in detail about Intermediate Code Generation.

(Refer 4th Question Answers)

3.

Discuss about Activation Record.



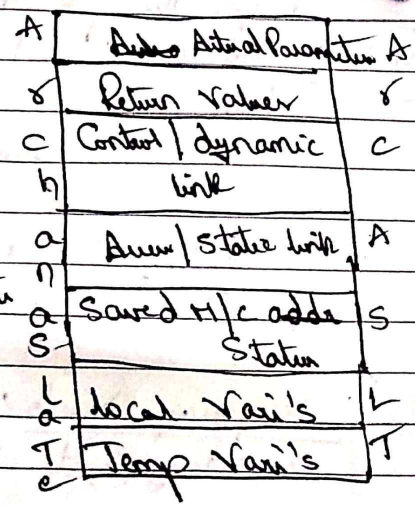
DOMS Page No.
Date / /

→ When we call a Fun F, we push a new activation record on the runtime Stack, which is particular to the Fun F.

→ It store the info needed for single Procedure Call.

(I) Actual Parameter:

It holds the actual parameter of Calling Fun.



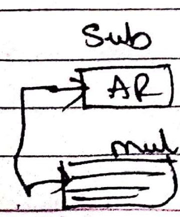

```
int sum(int x, int y)
{                         ↳ Formal parameter
  return (x+y);
}                         ↳ Returned Value
int main()
{
  int sum1 = sum(10,20);
     ↓               ↓
  return value    actual parameter.

  return 1;
}
```

(II) Returned Value: To store the result of fun Call. (eg: return (x+y);).

(III) Control / dynamic link: It pts to the AR of Calling fun.

who called this fun their AR addr is stored.

```
Sub()
{
  mul();
}
```



(IV) Access / static link: refers to the local data of called Fun but found in another AR.

(V) Saved M/c Status: holds the info about Sys / m before the procedure is called.

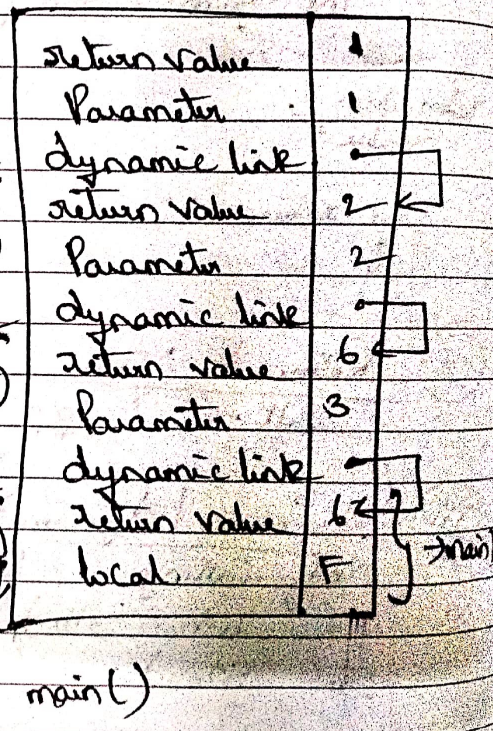
(VI) Local var's: These var's are local to the fun.

```
Sum()
{
int a=10;
}
```

local var's

(VII) Temporary Var's: which are stored & evalu according to the Requirement.

Eg:
```
main()
{
int F;
F = Fact(3);
}
int Fact(int n)
{
if(n==1)
return 1;
else
return (n * Fact(n-1));
}
```



Distinguish between S-attributes and L-attributes

S-attributes and L-attributes are used in **Syntax Directed Definitions (SDD)** for semantic analysis in compilers. The key differences between them are as follows:

Feature	**S-Attributed Definitions (S-attributed SDD)**	**L-Attributed Definitions (L-attributed SDD)**
Definition	Uses only **synthesized attributes**.	Uses both **synthesized and inherited attributes**.
Attribute Dependency	Attributes depend only on child nodes.	Attributes can depend on parent or left-side siblings.
Evaluation Order	Can be evaluated in a **single bottom-up traversal**.	Must be evaluated **top-down and left-to-right**.
Parsing Compatibility	Suitable for **bottom-up parsing (LR parsing)**.	Suitable for **top-down parsing (LL parsing)**.
Ease of Implementation	Easier to implement due to simple dependency rules.	More complex due to dependency on sibling attributes.
Example	E → E1 + T { E.val = E1.val + T.val }	T → num { T.type = parent.type }

Example Explanation

1. **S-Attributed Example (Bottom-Up)**

mathematica
CopyEdit
E → E1 + T { E.value = E1.value + T.value }

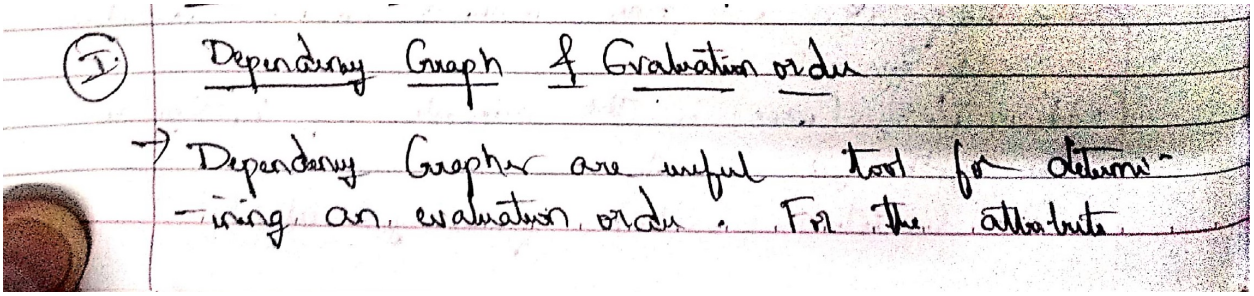
- Here, E.value is synthesized from E1.value and T.value, which are child nodes.

2. **L-Attributed Example (Top-Down)**

go
CopyEdit
T → num { T.type = parent.type }

- Here, T.type is inherited from its **parent node**, making it L-attributed.

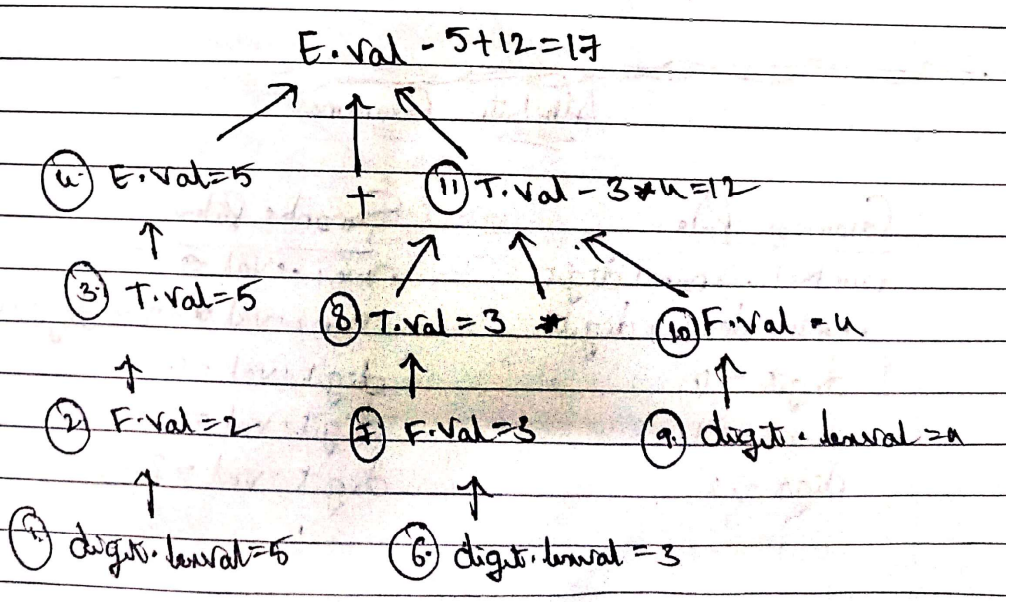
Explain in detail about Dependency Graph.

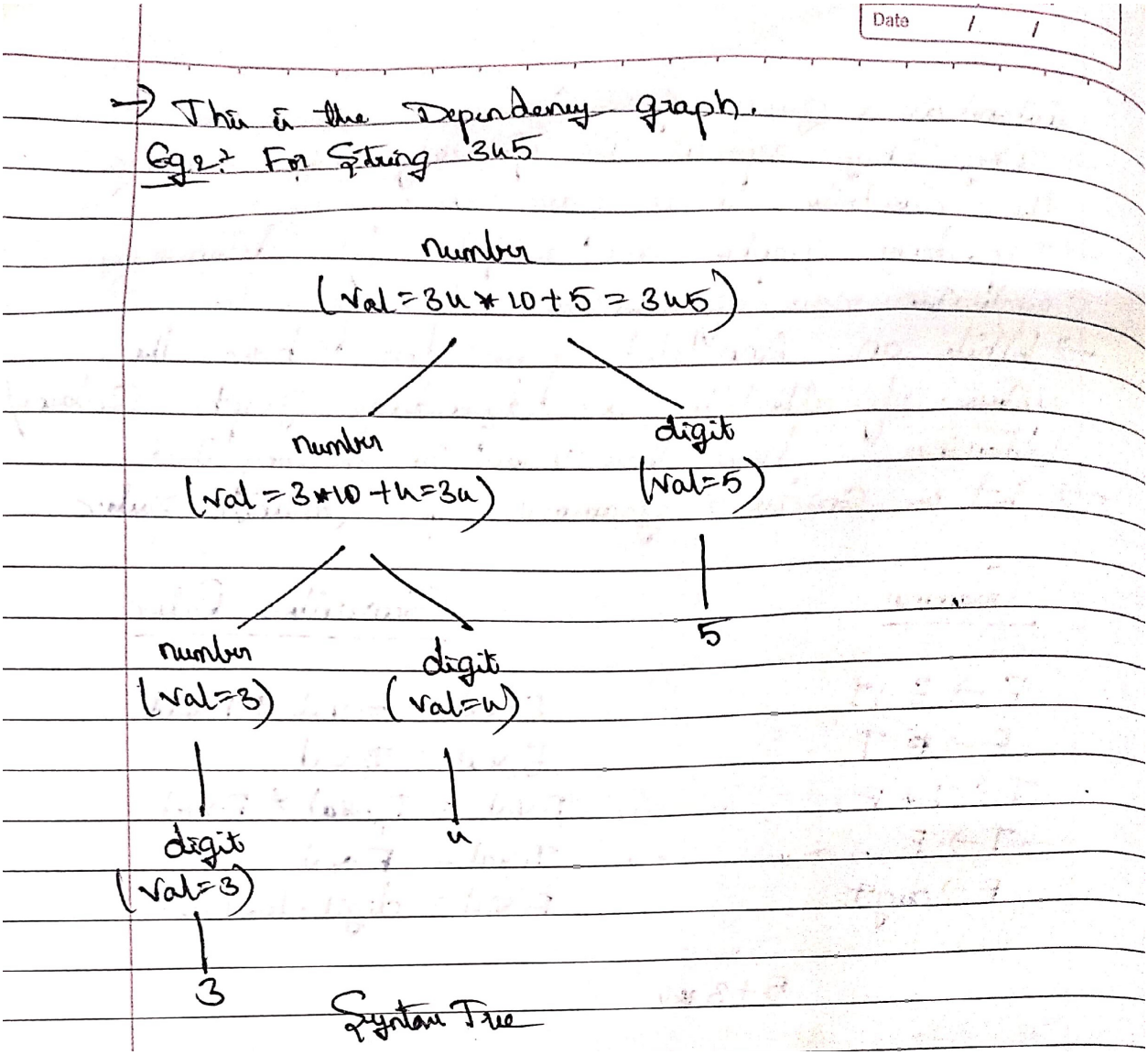


instance in a given parse tree.
→ They represent the flow of info among the attributes in a parse tree.
→ Dependency graphs are useful for determining evaluation order of attributes in parse tree.
→ While an Annotated parse tree shows the values of attributes a dependency graph helps determines how these values can be computed.
→ Let us consider a grammar & Semantic rules

Grammar	Semantic Rules
$E \to E+T$	$E.val = E.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T*F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to digit$	$F.val = digit.lexval$

5+3*4





Explain in detail about 3 Address Code Representation.

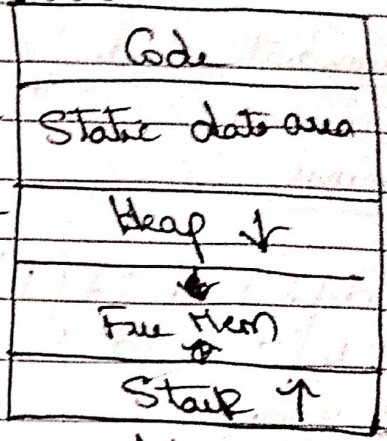
(Refer 4th Question Answers)

Explain in detail about Run-Time Environment.

→ Whenever a prog is executed, it needs some space to store its variables, registers etc.
→ And that storage is allocated by the OS.
→ So, when a prog is executing the prog does need some space to store its data at run time & Compile time.

→ Here it request OS to give some space (It gives a chunk of space).
→ This includes store low addr

Code :- It is a fixed size which is where the code can be stored (and intermediate code) (executable code).



Static data Area:

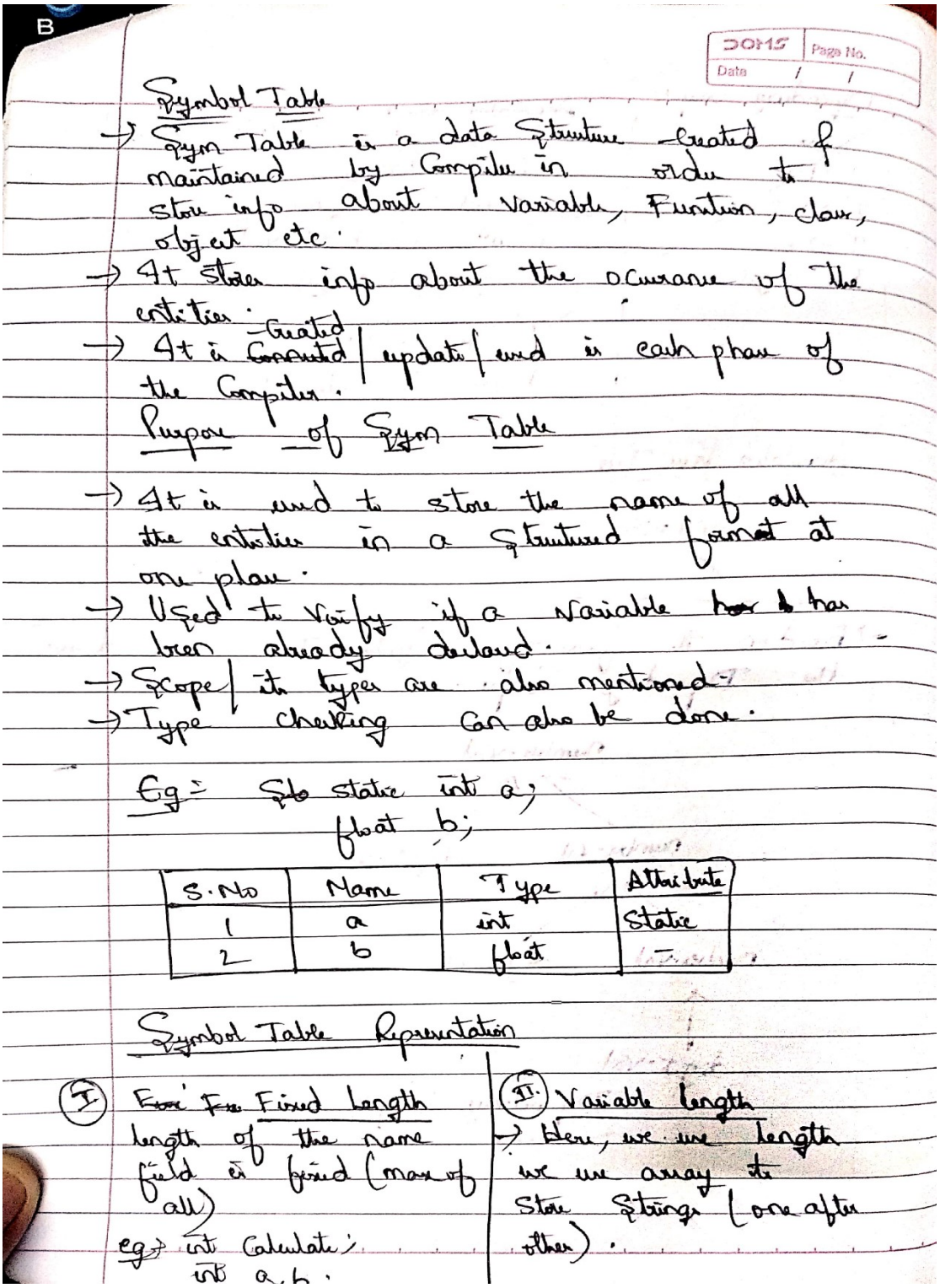
Static & Global Global high addr variables are stored.

Stack: Here stack grows from high to low addr space.
Whenever a procedure/Fun is called then an activation record is created & it is pushed on to stack.

Heap: To allocate mem at run time, we use Heap. (malloc, calloc, realloc, delete, free, etc).

4.

Explain in detail about Representation and Operations on Symbol Table.



Symbol Table

→ Sym Table is a data structure created & maintained by Compiler in order to store info about variable, Function, class, object etc.

→ It stores info about the occurance of the entities.

→ It is created / updated / used in each phase of the Compiler.

Purpose of Sym Table

→ It is used to store the name of all the entities in a structured format at one place.

→ Used to verify if a variable has been already declared.

→ Scope / its types are also mentioned.

→ Type checking can also be done.

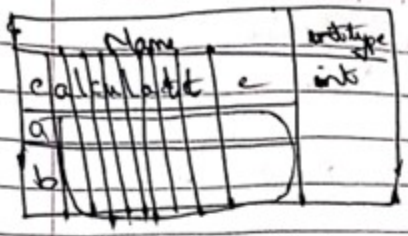
Eg:- Sub static int a;
float b;

S.No	Name	Type	Attribute
1	a	int	Static
2	b	float	—

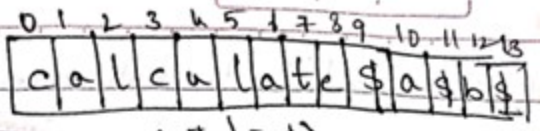
Symbol Table Representation

(I) Fixed length
length of the name field is fixed (max of all)
eg → int Calculate;
int a, b;

(II) Variable length
→ Here, we use length we use array to store strings (one after other).



Width or its fixed length



(embedding $)

Starting	length	type
0	10	int
10	2	int
12	2	int.

Operation on Symbol Table

(I) Insert ().
(II) lookup()
(III) Delete ()
(IV) Scope Mgmt

(I) Insert(): (insert (a, int ()) → Name → Type

→ It is more frequently used in analysis phase when tokens are identified & names are stored in table.
→ It takes the name & type of variable or argument.

eg: int x;
insert (x, int)

(II) lookup():
→ It is used to search a name & its metadata
→ To check check whether the variable already exists in scope or not.

lookup (Symbol)

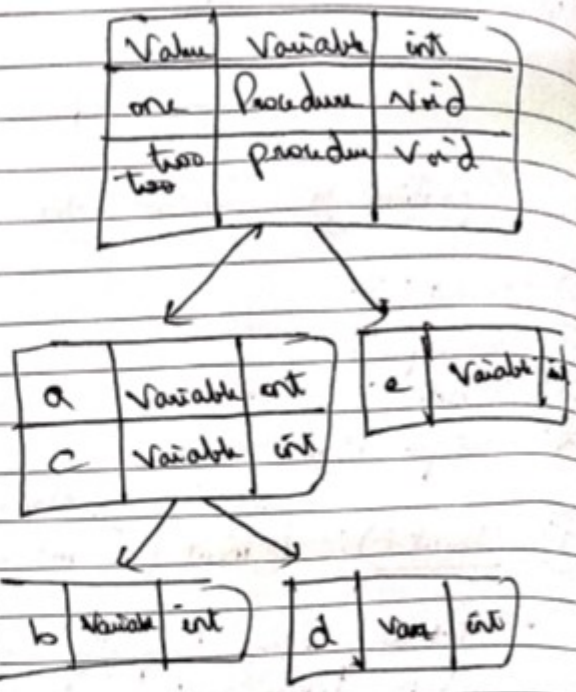
eg: lookup(x);

DOMS | Page No.
Date / /

(iii) delete (): To delete/remove a particular symbol.
delete (symbol)
delete (x)

(iv) Scope right : defining local/global ... in a tree structure.

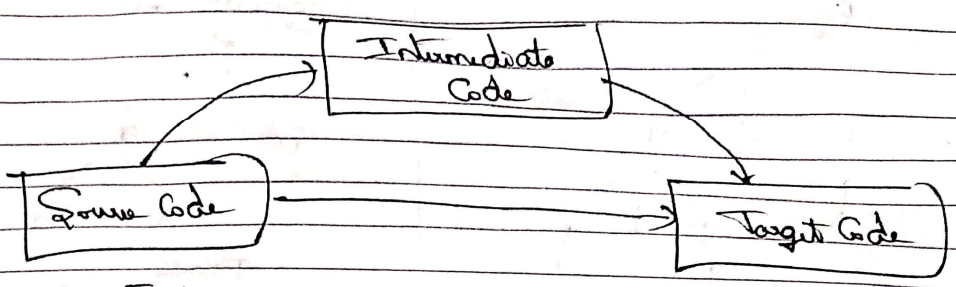
```
int value;
void one()
{
    int a;
    {
        int b;;
    }
    int c;
    {
        int d;
    }
}
void two()
{
    int e;
}
```



global
↓
local

Explain in detail about Intermediate Code Generation and 3 Representations of Intermediate Code Generator.

Intermediate Code Generation



→ Firstly any Source Code can be converted in target code but only issue with direct approach is that we need to design for each m/c or a full native Compiler.
→ Idea to avoid creating Separate Compilers / prog's
→ Here we generate Intermediate Code (i.e. 3 address Code) which can be converted and ran on a Target Code (which any m/c can understand).

3 Representations of Intermediate Code

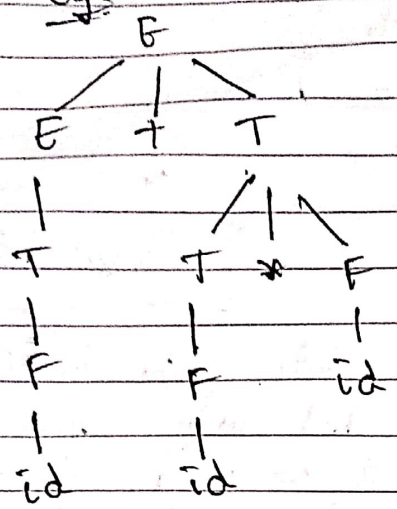
I. Syntax Tree (or) Abstract Syntax Tree
II. Post fix notation
III. 3 address Code representation.

I. Syntax Tree / Abstract Syntax Tree / Directed Acyclic Graph (DAG).

→ These are also abstract / Compact representation of parse Tree
→ DAG – Compressed Parse Tree

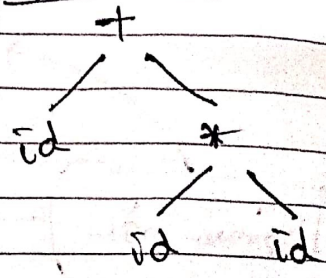
DOMS | Page No.
Date / /

Eg:



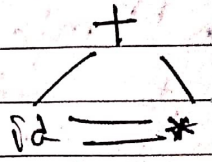
Parse Trees

Syntax Tree



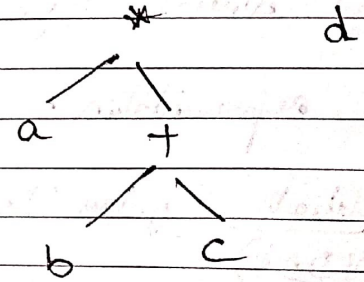
~~DAG~~

DAG



→ In Syntax Tree root / Parent internal nodes are operators (leaf node represent operands)

a * (b + c) / d



DOMS | Page No.
Date / /

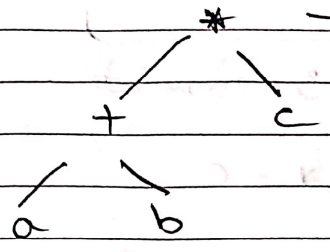
(II) Post fix or Post fix Notation:

→ Here the operators are placed on the right side like

a+b → Infix
ab+ → Post fix
+ab → Pre fix.

To avoid any confusion
draw Syntax Tree
do post fix traversal

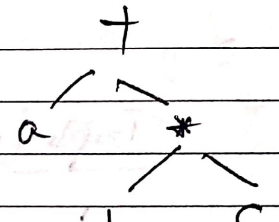
eg: (a+b)*c



(or)
(a+b)*c
ab+*c
ab+c*

ab+c*

eg 2: a+(b*c)



(or)
a+(b*c)
a+(bc*)
a+bc*
abc*+

abc*+

(III) 3-Code Address Code Representation

→ A Stmt involving not more than 3 address / references.
(2 for operands & one for result)
is called 3-Code address Code Stmt.

→ A Collection of three Statements is called Code.
(Note: At most 1 operator on RHS).
→ 3 addr Code can be represented in 3 ways
(a) Quadruple (4)
(b) Triple (3)
(c) Indirect Triple ((3) + Indirect

Eg:- $x + y * z$

$t_1 = y * z$
$t_2 = x + t_1$

→ TCG.

3 addrs Code.

Eg2: $a = b * -c + b * -c$

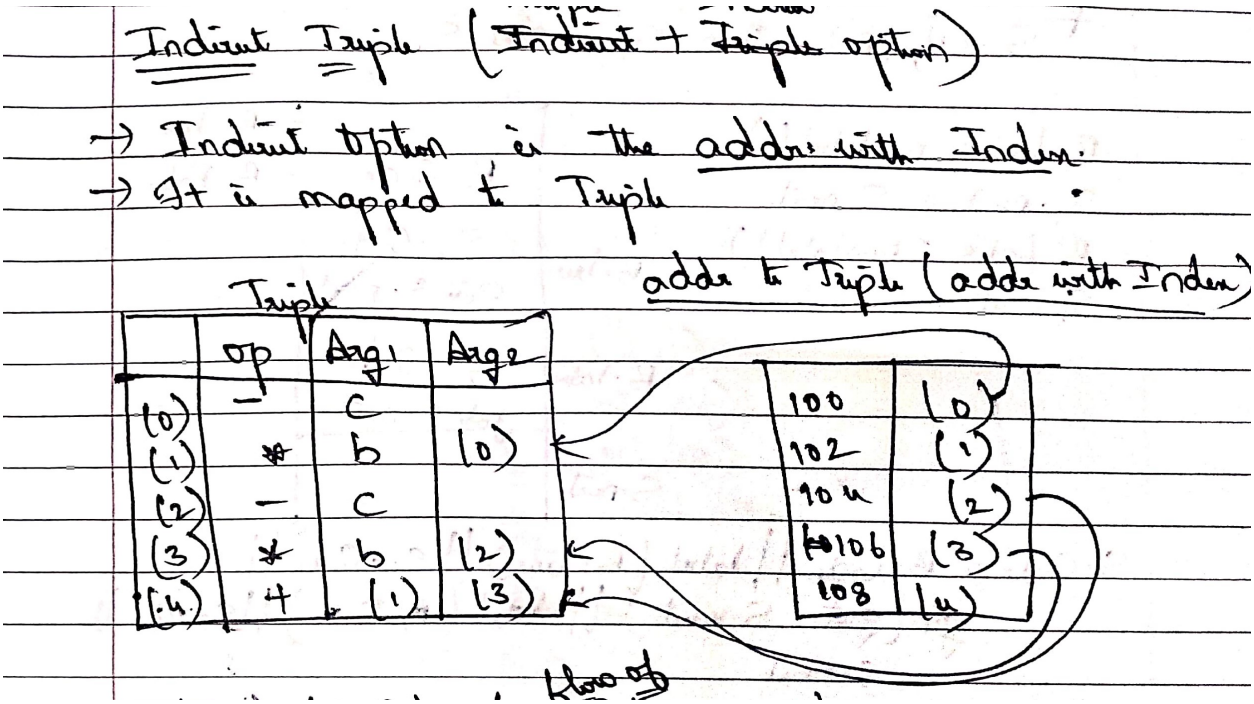
$t_1 = -c$ (Unary high priority)
$t_2 = b * t_1$
$t_3 = -c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$

Quadruple (4 arg)

	Op	arg1	arg2	result
(0)	−	c		t_1
(1)	*	b	t_1	t_2
(2)	−	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5

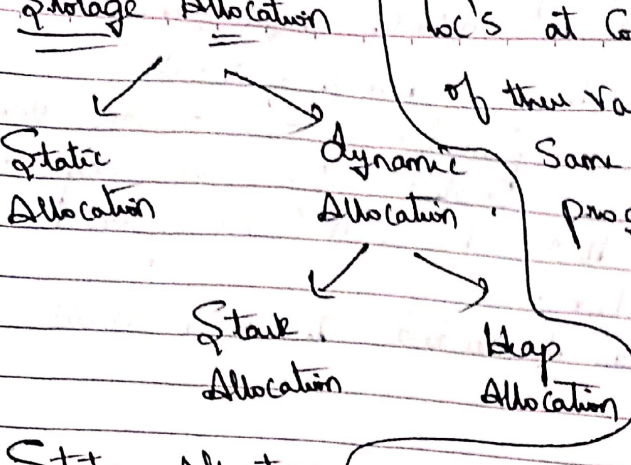
Triple (3 arg)

	Op	arg1	arg2
(0)	−	c	
(1)	*	b	(0)
(2)	−	c	
(3)	*	b	(2)
(4)	+	(1)	(3)



Explain in detail about Storage Allocation.

→ Static Alloc: All var's that are created will be assigned to mem locs loc's at Compile time of the addr's of these var's will remain the same throughout the prog execution.



Static Allocation

→ is a procedure which is used for allocation of all data obj's at Compile time.
→ is possible only when the Compiler known the size of data obj's at Compile time.

Drawbacks (interms of array)

→ The array/any data structure size should be known to us prior to the creation of data structure.
→ If more mem is allocated than required then mem will be wasted.
→ If less mem is allocated then we cannot change in run time or per use
→ Insertion & deletion operations are very expensive (as shifting should be done multiple times).

Dynamic Allocation (at run time)

→ is the process of assigning the mem space during the execution time or run time.

Stack Allocation

→ is a procedure in which Stack is used to

→ Stack Alloc: Dyn. Alloc means alloc of mem at run-time, Stack - Data Str that follows LIFO, Page No.
So whenever there a multiple AR - Created it will be pushed/popped in the o Stack as Activation begin & ends.

Organize the Storage (last in first out)

→ Whenever a fun or procedure call occures then an AR will be created for the Corresponding fun & that AR will be pushed on to the Stack.
(based on marks if needed to write about AR).

Heap Allocation:

→ is the most Flexible allocation Structure Scheme.
→ Allocation & deallocation of mem can be done at any time & at any place depending upon the user requirement.
→ Support the recursion procedure.
(using: malloc(), calloc(), free(), realloc() ...) are used to acquire this