Module 4

Syntax Directed Definition (SDD).

→ It is a combination of context Free Grammars (CFGs) and 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 can be numbers, strings, references (addresses), datatypes etc.

eg:

Production.	semantic rule.
$E \rightarrow E + T$	$E.val = E.val + T.val$
$E \rightarrow T$	$E.val = T.val$

where E and T are grammar symbols and 'val' is the attribute associated with these symbols.

Types of attributes.

a) synthesized attribute b) Inherited attribute.

→ **synthesized attribute**

IF a node takes value from its child nodes, then it is synthesized attribute.

eg: $A \longrightarrow BCD$. (A → Parent node and B,C,D → child node)

Here A is taking value from B, C and D. So A is a synthesized attribute.

ie $A.S = B.S$ where S is the attribute associated
$A.S = C.S$ with the nodes.
$A.S = D.S$

→ Inherited attribute

IF a node takes value from either the parent or from its sibling nodes, then it is called inherited attribute.

eg: $A \longrightarrow BCD$.

A — parent node
B, C, D — child nodes.

(i is the attribute associated with A, B, C and D)

ie $C.i = A.i$
$B.i = A.i$
$D.i = A.i$

and

$C.i = A.i$ ('C' taking value from parent node)

$C.i = B.i$ ('C' is taking value from sibling node)

$C.i = D.i$ ('C' is taking value from sibling node)

Types of SDD

→ It is of 2 types:

a) S- attributed SDD or S- attributed definition or S- attributed grammar.

b) L- attributed SDD or L- attributed definition or L- attributed grammar.

→ S- attributed SDD

* An SDD that uses only synthesized attribute is called as S- attributed SDD.

eg: $A \longrightarrow BCD$

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

Here parent node A is taking value from child nodes B, C and D.

→ L- attributed SDD. (L stands for left sibling).

If an SDD uses both synthesized as well as inherited attribute, then it is known as L- attributed SDD.
Here there is a restriction to inherited attribute, that is the nodes should take value only from its parent node or from its left sibling node.

eg: $A \rightarrow xyz$.

$\{ y.s = A.s, \quad y.s = X.s, \quad y.s = Z.s \}$

Let's consider $y.s = A.s$ child node 'y' is inheriting value From parent node 'A'. so no issue.

$y.s = X.s$ child node 'y' is taking value From left sibling 'X'. so no issue.

$y.s = Z.s$ child node 'y' is taking value From right sibling 'Z', that is not possible in L-attributed SDD.

→ In S-attributed SDD, the semantic actions are always placed at the right end of the productions. so it is called postfix SDD.

→ whereas in L-attributed SDD, we can place semantic actions anywhere in the productions.

→ In S-attributed SDD, we use bottom up parsing to evaluate the attributes.

→ In L-attributed SDD, we use top down, left to right parsing to evaluate the attributes.

(a) $S \rightarrow MN \{ S.val = M.val + N.val \}$

$M \rightarrow PQ \{ M.val = P.val * Q.val \text{ and } P.val = Q.val \}$.

* $S \rightarrow MN \{ S.val = M.val + N.val \}$.

Parent — S

Child — M, N.

$S.val = M.val + N.val$ — parent is taking value From child nodes.

Here S is the synthesized attribute.

$M \rightarrow PQ$ { $M.val = P.val * Q.val$ and $P.val = Q.val$ }.

M — parent

P, Q — child.

$M.val = P.val * Q.val$ — parent node M is taking value from child nodes P and Q.

$P.val = Q.val$ — Here child node P is taking value from right sibling Q, that violates the condition of L-attributed SDD.

Q) $A \rightarrow QR$ { $R.i = F(A.i)$, $Q.i = F(R.i)$ }

$R.i = F(A.i)$ → inherited attribute. ∴ It is L-attribute.

$Q.i = F(R.i)$ → not L-attribute, as it is inherited from the right sibling.

Q) $A \rightarrow BC$ { $B.S = A.S$ }.

$B.S = A.S$ Inherited attribute ∴ L-attributed.

Bottom up Evaluation of S-attributed definition

→ Passes are of 2 types: Bottom up passes and top down passes.

→ Attributes are of 2 types: synthesized and inherited.

→ In S-attributed definition, we use bottom up passes to evaluate the attributes.

→ S-attributed definition uses only synthesized attributes. ie Parent node will be taking value only from its child nodes.

→ Consider the SDD:

~~$S \rightarrow ABE \quad \{ S.a = f(A.a, B.g, E.a) \}$~~

→ In this approach, the passes will keep the values of synthesized attributes associated with the grammar on its stack.

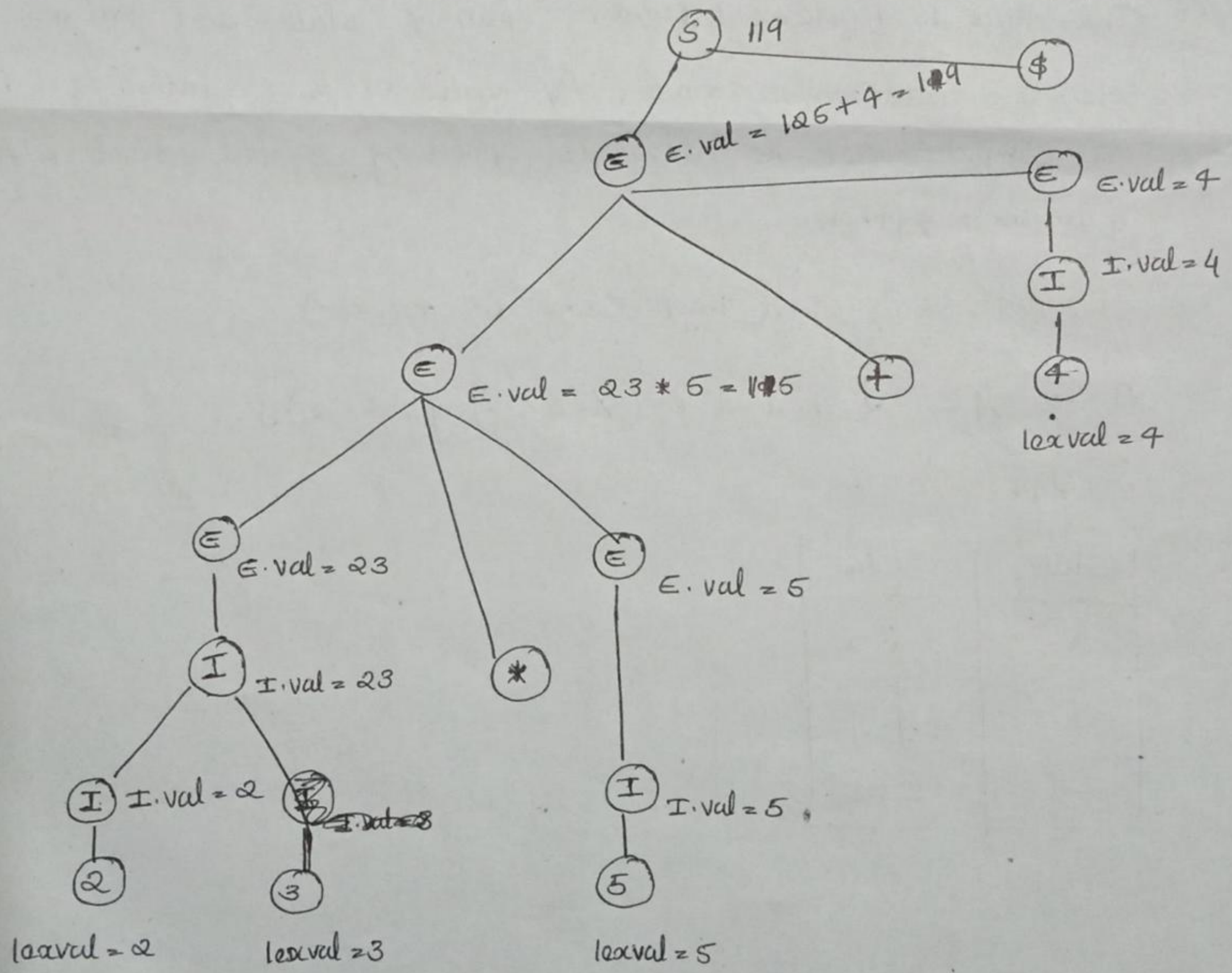
→ The stack is implemented as a pair of state and value.

→ When a reduction is made, the value of the synthesized attribute are computed from the attribute appearing on the stack for the grammar symbols.

eg: Implement SDT for simple desk calculator, and evaluate the expression 23 * 5 + 4 $ using SDT scheme.

Production.	Semantic action.
$S \rightarrow E\$$	{ print E.val }
$E \rightarrow E+E$	{ E.val = E.val + E.val }
$E \rightarrow E*E$	{ E.val = E.val * E.val }
$E \rightarrow (E)$	{ E.val = E.val }
$E \rightarrow I$	{ E.val = I.val }
$I \rightarrow I$ digit	{ I.val = I.val * 10 + Lexval }
$I \rightarrow$ digit	{ I.val = Lexval }.

→ Now we have to create parse tree for 23 * 5 + 4 $.



→ Here we have constructed the parse tree and simultaneously we have ~~Eval~~ evaluated the given expression also.

→ Here, IF we check the diagram, For calculating every parent nodes value, we used values of child nodes, ie S-attributed defenition as we are using synthesized attributes.

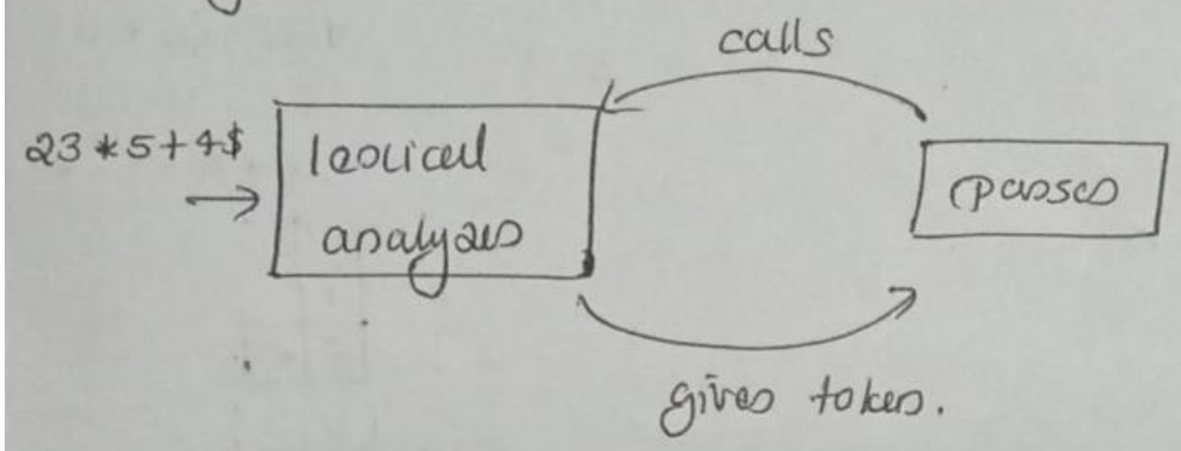
→ To implement this SDT, we have to do 2 things.

a) create a lexical analyzer

b) Create a bottom up passer.

→ How to construct a lexical analyzer ?

Give 23 * 5+4 $ as input to the lexical analyzer and lexical analyzer will Findout the tokens and will give to the passer.



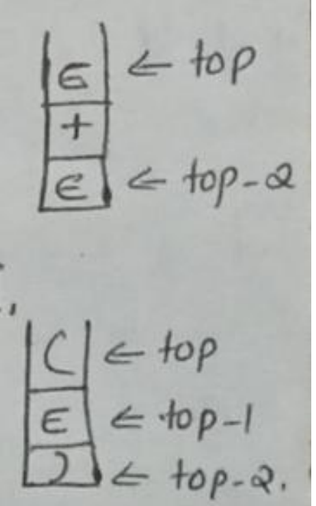
23 * 5+4$

Here the passer takes and process each tolans gives by the lexical analyzer.

→ Now, we have to construct LR(1) passer.

Here, we will be using a stack named 'val'.

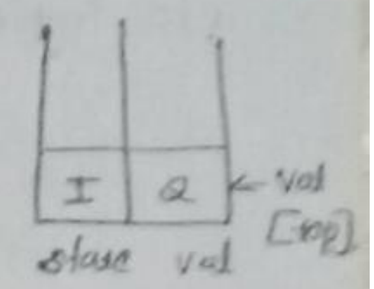
Production	Pgm Fragment
$S \to E\$$	print val[top]
$E \to E+E$	val[top] = val[top] + val[top-2].
$E \to E*E$	val[top] = val[top] * val[top-2].
$E \to (E)$	val[top] = val[top-1]. //
$E \to I$	———
$I \to I$ digit	val[top] = 10 * val[top] + lexval
$I \to$ digit.	val[top] = lexval.

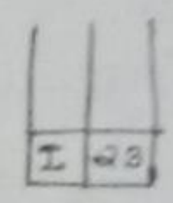


) will not be used For evalo ∴ top-1 th Item only considered

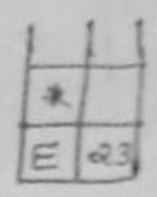
→ Implementation using stack.

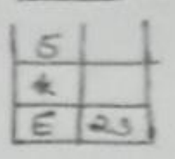
Input	stack	val	Production used.
23 * 5+4$	—	—	—
3 * 5+4$	2	—	—
3 * 5+4 $	I (2 is a digit. ∴ POP 2 push I)	2	I → digit.
* 5+4 $	I3 (3 is on top of stack.)	2	—
* 5+4 $	I (POP I3 push I)	23	I → I digits
* 5+4 $	E	23	E → I
5+4 $	E *	23_	—
+4 $	E * 5	23__	—
+4 $	E * I	23_5	I → digit.
+4 $	E * E	23_5	E → I
+4 $	E (~~E * E~~)	115	E → E * E
4 $	E +	115	
$	E + 4	115	—
$	E + I	115_	I → digit
$	E + E	115_4	E → I
$	E	119	E → E+E
	E$	119	
	S		Print 116

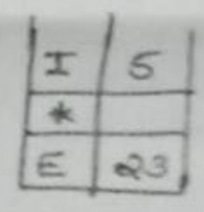


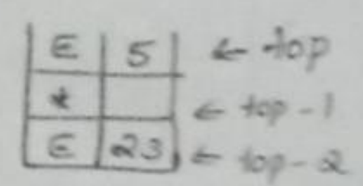


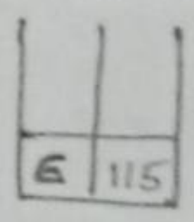
val [top] * 10 + lexval.
2 * 10 + 3 = 23.

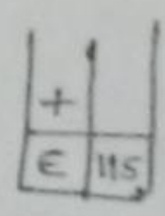


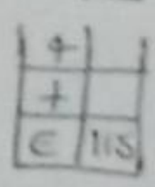


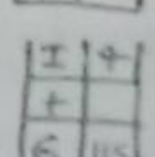


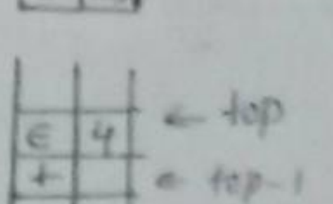


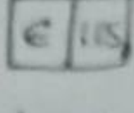












Run time Environment or runtime storage management

→ when we write any Pgms and saves, it get stored to harddisk

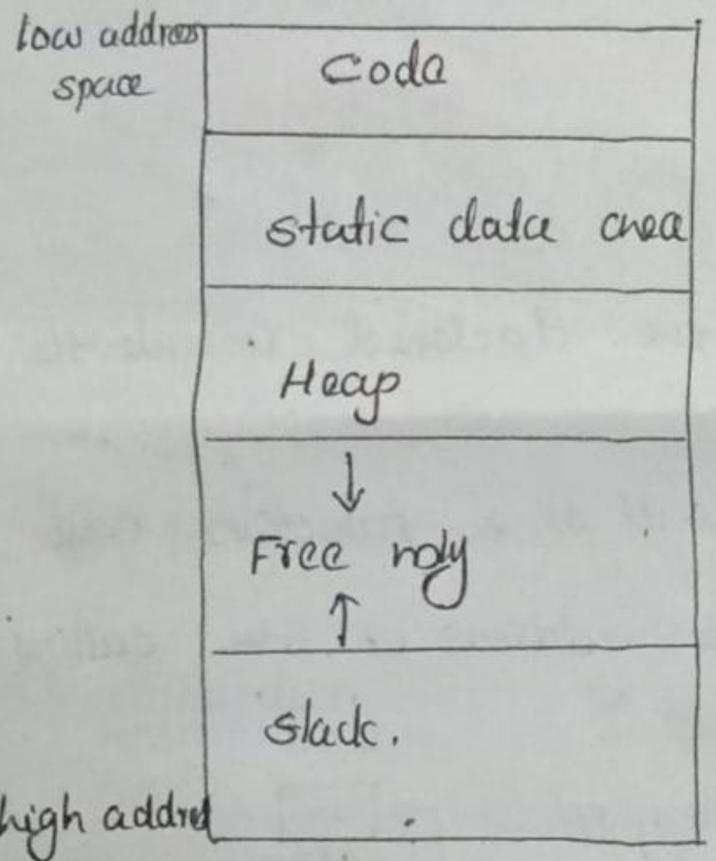
→ But the execution happens only when the Pgm is in main memory.

→ so, after compilation, the compiler demands a block of memory from the OS. inorder to store Pgms in main memory. so, OS allocates a free block of mly to the corresponding Pgm.

→ so, the compiler uses that free block of memory inorder to store the Pgms.

This is called runtime storage management. (ie how the Pgm is stored during execution time).

→ The main memory is divided as Follows:



→ During compilation only, the size of the code is decided. The executable code will get stored to the code area.

→ After compilation, linking will be happing linking means, combining multiple Files into a single File.

→ Linker produces a File called executable File (ie .exe Files.). This will get stored to the code area of main memory.

→ static data area mainly stores static variables and global variables.

→ Heap and stack are mainly used to use the space in an effective manner.

Heap grows from low address space to high address space, whereas stack grows from high address space to low address space.

→ stack works in LIFO manner. whenever a function is called, then an activation record gets created. Inorder to store the activation record info, stack is used. The activation record info will get pushed on to the top of the stack.

Model of activation record.

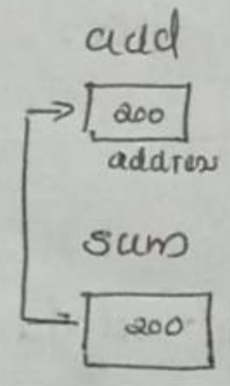
Actual parameters
Returned value
control or dynamic link.
Access or static link.
saved machine status
Local variables
Temporary variables.

→ Actual parameters – are parameters which are declared inside the calling function.

→ Return value – used to store the result of a function call

→ control or dynamic link – Inorder to store the address of the calling function.

```
add()   ← calling F.D
{
  sum()
}
```



→ Access link – It refers to the local data of called function, but found in another activation record.

→ saved machine status. – stores address of next instruction to be executed.

→ Local variables – These variables are local to a function.

→ Temporary variables – Needed for evaluating expression.

Heap — Heap is a dynamic data structure, in ordes to allocate memory at runtime, we uses heap.

→ If there are too many functions in a pgm, then stack is used more often. so Because, the free memory is occupied more by the stack.

→ IF there are many pointer variables, and we use heap to allocate memory to such pointer variables. so, heap will uses the free mly more often.

Source lang Issues.

→ what all things to be considered while we are writing a Pgmg lang.

They are:

* Procedures
* Activation Trees
* control stacks
* The scope of declaration
* Binding of names.

a) Procedure

A procedure definition is a declaration that associates with an identifies with a statement.

The identifies is the procedure name and the stmt is the procedure body.

eg: The Following is the definition of procedure named 'readarray' Pascal lang.

```
Procedure readarray ;
  var i ; Integer ;
  begin
  For i := 1 to 9 do read (a[i])
  end ;
```

When a procedure name appears within an executable stmt, the procedure is said to be called at that point. ie. If we call this readarray Function, in another Function, the entire fn will get

b) Activation ~~Tree~~ Tree

→ An activation tree is used to depict the way control enters and leaves activations. In an activation tree,

* Each node represents the activation of the main pgm. procedure
* The root represents the activation of the main pgm.
* The node for 'a' is the parent of the node for 'b' if and only if the control flows from activation 'a' to 'b'.
* The node for 'a' is to the left of the node for 'b' if and only if the lifetime of 'a' occurs before the lifetime of 'b'.

c) control stack

→ used to track live procedure activations.

→ The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.

→ The contents of the control stack are related to paths to the root of the activation tree.

→ when node 'n' is at the top of control stack, the stack contains the nodes along the path from 'n' to the root.

d) The scope of a declaration.

→ A declaration is a syntactic construct that associates info with a name. (uses can define)

declaration may be explicit, such as:

var i : integer ;

or may be implicit. eg: any variable name starting with 'I' is assumed to denote an integer. The portion of the pgm to which a declaration applies is called the scope of that declaration.

e) Binding of names.

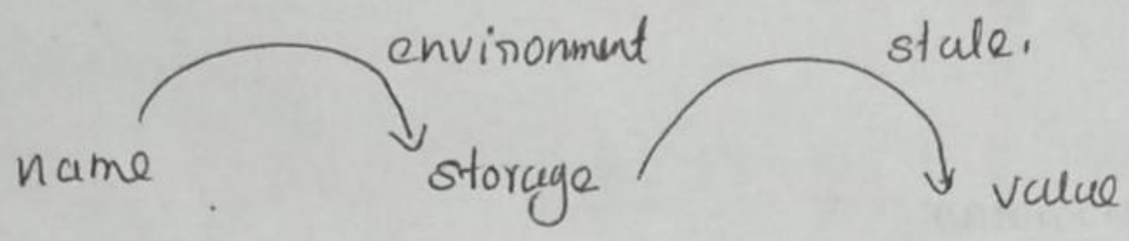
→ Even if each name is declared once in a pgm, the same name may denote different data objects at run time.

→ "Data object" corresponds to a storage location that holds value.

→ The term environment refers to a function that maps a name to a storage location.

→ The term state refers to a function that maps to a storage location to the value held there.

→ When an environment associates storage location "s" with a name "x", we say that x is bound to s. This association is referred to as binding of x.



Two stage mapping from names to values.

eg: a = 5

storing variable name 'a' to a storage locating ⟶ environment

assigning value 5 to that variable ⟶ state.

Storage allocation strategies

Mainly there are 3 allocation strategies:

1) static allocation.
2) stack allocation } Dynamic allocation.
3) Heap allocation. }

1) Static allocation – Allocation of memory during compilation time. once mfy is allocated during compilation time, it is not possible to change the size of variables, or array during run time or execution time.

Drawbacks of static mory allocation:

→ we must know the size of the array in advance

→ If more memory is allocated than required, then memory will get wasted.

eg: mory allocated for 100 elements, and we used only 40 elements. ∴ mory got wasted here.

→ If less memory is allocated than required, then it is not possible to perform the corresponding operations.

→ For insertion and deletion of no's we have to perform too many shifting operations, which can be more expensive, if the no. elements are more in the array.

* Inorder to overcome all these problems, we use dynamic allocation.

② <u>Stack allocation.</u>

→ stack works in LIFO manner.

→ whenever a Function or procedure call occurs, then an activation record will be created for the corresponding Function, and that activation record will get pushed on to the stack.

If there are 5 Functions in our program, 5 activation records will get created.

(we have already discussed about activation record).

Drawbacks of stack allocation:

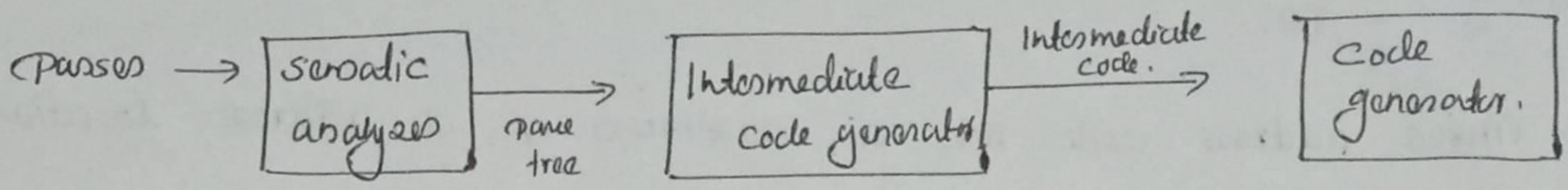
* It supports dynamic allocation, but slower than static allocation
* It supports recursion but references to non local variables after activation records cannot be retained.

→ Inorder to overcome these problems, we use heap allocation.

* Heap allocation supports recursion and the references to non local variables after activation records can be retained.
* In 'C' language, we have malloc, calloc, realloc and Free Function for dynamic memory allocation.

Intermediate code generation.

→ It is the 4th Phase of compiler.



→ Some code can be directly converted to ~~Vector~~ Target code? Then what is the need for intermediate code. ?

→ If there is no intermediate code generation, then a full native compiler is required.

→ Intermediate code eliminates the need of a full native compiler for every machine keeping the analysis portion same for all the compilers.

→ Intermediate code can be language dependent or language independent.

lang specific ⟶ byte code for java.
lang independent → Three address code.

Intermediate languages.

3 ways of intermediate representation :

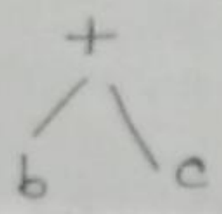
1) syntax tree or Abstract syntax Tree.
2) Postfix notation
3) Three address code.

a) syntax tree or Abstract syntax Tree.

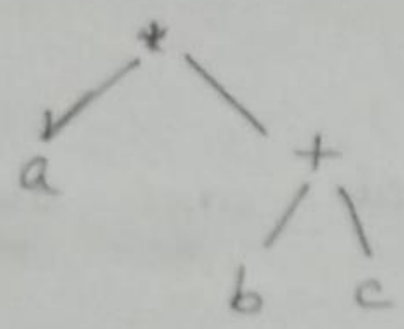
→ Each parent node will be the operators and leaf node will be operands.

eg: a * (b+c) /d

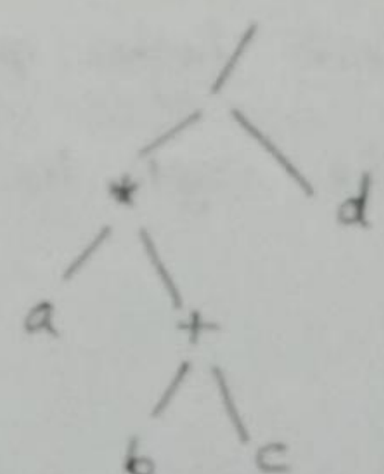
Here parantheis will have highes priority ∴ (b+c) will get executed first.



* and / have equal priority. Then choose the expression based on associativity. For arithmetic operators, associativity is From left to right. ∴ '*' will get executed next.



Next '/' will get executed.



b) Postfix notation

If the operator appears after the operands, then it is called postfix notation.

consider the infix notation : $(a+b) * c$.

( ) has highes priority ∴ $(a+b)$ will get executed

∴ postfix expression will be $ab+c*$

IF Infix is $a + (b * c)$ then postfix = $abc*+$

Infix : $(a-b) * (c/d)$ postfix = $ab- cd/*$

c) <u>Three address code instn.</u>

In a 3 address code instn each instn should contain at most 3 addresses and the R.H.S should contain at most 1 operator.

3 address code can be represented in 3 ways :

① quandruples ② Triples ③ Indirect triple.

eg1: $x + y * z$

* has high priority ∴ $y * z$ will get executed first.

$t1 = y * z$ Here in RHS we have only 1 operator.

$t2 = x + t1$ and each instn contain only 3 addresses.

eg2: $a = b * -c + b * -c$

① quandruples : It contain 4 Fields operator, arg1, arg2, result.

unary minus has highes priority.

1st represent the expn in 3 address code.

$t1 = -c$

$t2 = b * t1$

$t3 = -c$

$t4 = b * t3$

$t5 = t2 + t4.$

we can write like this also because $-c$ is already there in $t1$.

$t3 = b * t1$

$t4 = t2 + t3$

Now we have to represent this 3 address code in quandruple.

address to store instr	operator	arg1	arg2	result
(0)	−	c		t1
(1)	*	b	t1	t2
(2)	−	c	t	t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5

This is how we are storing 3 address code in the form of quadruples.

→ one disadvantages is that we have too many temporary variables to store.

b) Triples : It contains operator, arg1, and arg2.

	operator	arg1	arg2
(0)	−	c	
(1)	*	b	(0) ← since we stored t1 in 0th address.
(2)	−	c	
(3)	*	b	(2)
(4)	+	(1)	(3)

Here we avoid the use of temporary variables, instead we are giving the location of those variables.

∴ with less amount of memory, we can execute the instructions.