SDT for Flow Control

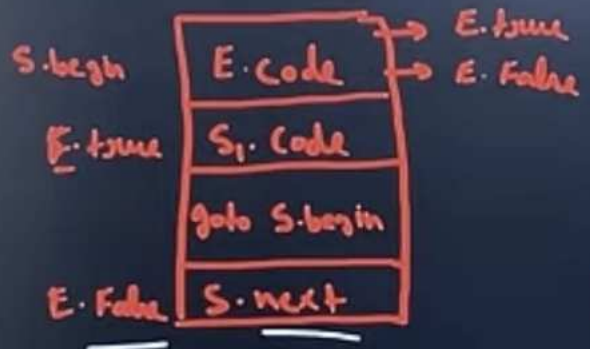
Production rule	Semantic action
Switch E case V1 : S1 case V2 : S2 ... case Vn–1 : Sn–1 default : Sn }	Evaluate E into t such that t = E goto check L1 : code for S1 goto last L2 :code for S2 goto last Ln :code for Sn goto last check:if t=V1 goto L1 if t = V2 goto L2 ... if t = Vn–1 goto Ln–1 goto Ln last

```
switch(ch)
{
    case 1 : c = a + b;
    break;
    case 2 : c = a – b;
    break;
}
```

```
if ch = 1 goto L1
if ch = 2 goto L2
L1: t1:=a+b
    c := t1
    goto last
L2: t2:= a – b
    c := t2
    goto last
```

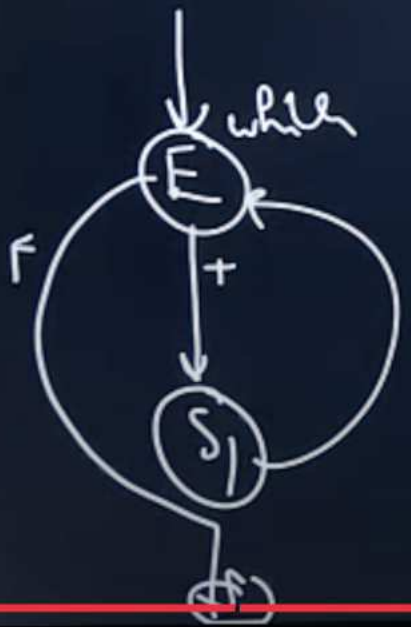
SDT for Flow Control

$S \rightarrow$ while E S_1



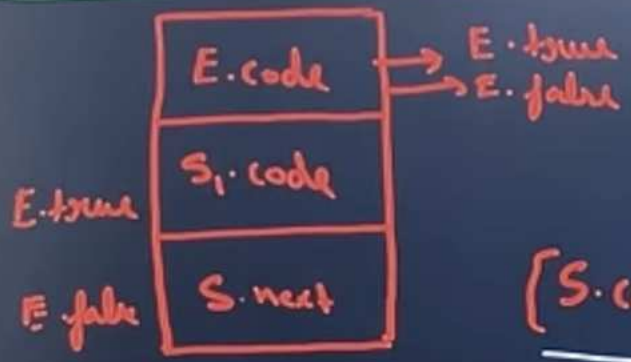
S.begin = newlabel()
E.true = newlabel
E.False = S.next
S_1.next = S.begin

S.code = gen(S.begin) || E.code || gen(E.true ':') || S_1.code || gen('goto' S.begin)



SDT for Flow Control

$S \rightarrow$ if E then S_1



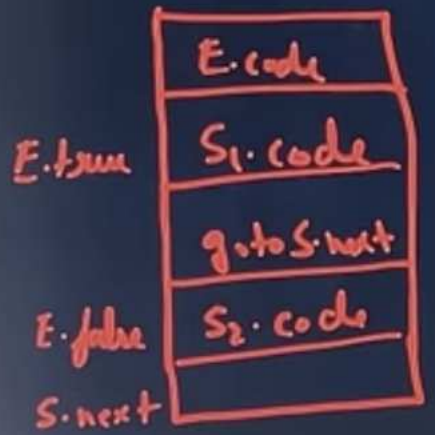
$E.true = newlabel$

$E.false = S.next$

$S_1.next = S.next$

$S.code = E.code \parallel gen(E.true ':') \parallel S_1.code$

$S \rightarrow$ if E then S_1 else S_2



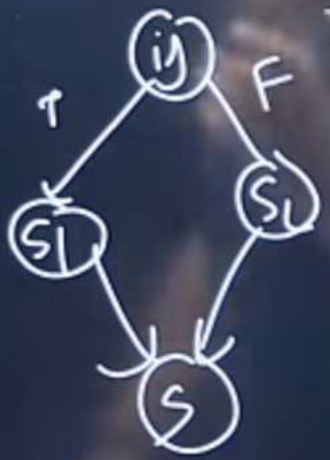
$E.true = new\ label$

$E.false = new\ label$

$S_1.next = S.next$

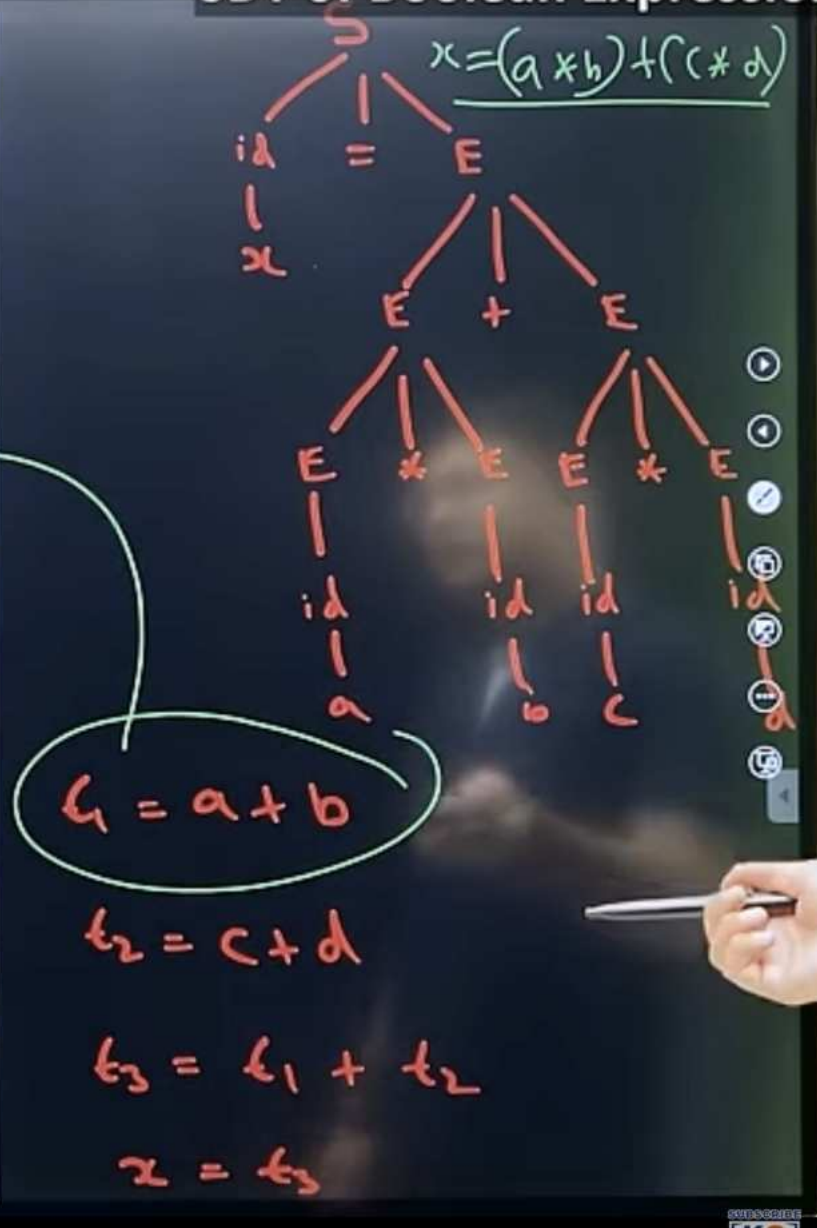
$S_2.next = S.next$

$S.code = E.code \parallel gen(E.true ':') \parallel S_1.code \parallel gen(S_2.next) \parallel gen(E.false ':') \parallel S_2.code$



SDT of Boolean Expression

Production rule	Semantic actions
S → id := E	{ id_entry := look_up(id.name); if id_entry != nil then append (id_entry ':=' E.place) else error; /* id not declared*/ }
E→E1 +E2	{ E.place := newtemp(); append (E.place ':=' E1.place '+' E2.place) }
E→E1 *E2	{ E.place := newtemp(); append (E.place ':=' E1.place '*' E2.place) }
E → – E1	{ E.place := newtemp(); append (E.place ':=' 'minus' E1.place) }
E → (E1)	{ E. place:= E1.place }
E → id	{ id_entry: = look_up(id.name); if id_entry != nil then append (id_entry ':=' E.place) else error; /* id not declared*/ }

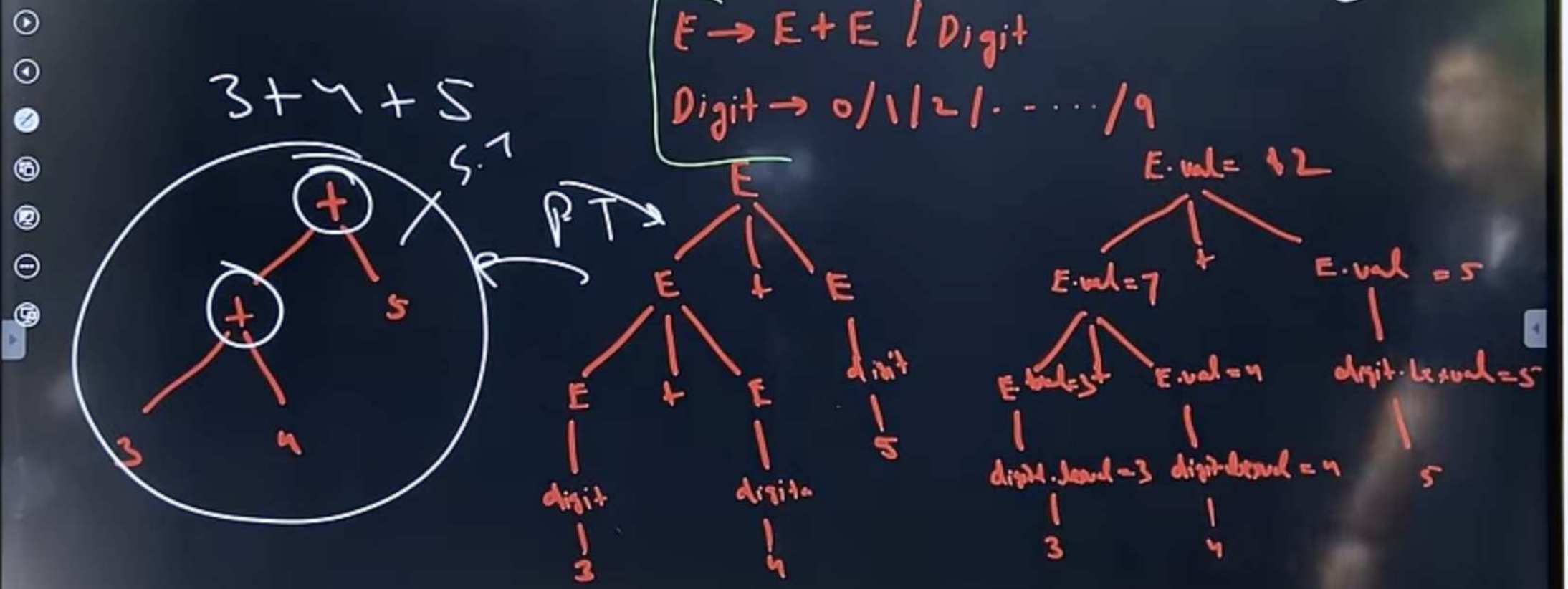


SDT of Boolean Expression

$E \rightarrow E_1$ OR E_2	E.place=newtemp(); Emit(E.place=E_1.place 'or' E_2.place);
$E \rightarrow E_1$ AND E_2	E.place=newtemp(); Emit(E.place=E_1.place 'and' E_2.place);
$E \rightarrow$ NOT E_1	E.place=newtemp(); Emit(E.place= 'not' E_1.place);
$E \rightarrow (E_1)$	E.place=E_1.place;
$E \rightarrow$ TRUE	E.place=newtemp(); Emit(E.place='1');
$E \rightarrow$ FALSE	E.place=newtemp(); Emit(E.place='0');

Chapter-3 (SYNTAX-DIRECTED TRANSLATION) Address statements in intermediate code

- **Syntax Tree**: An abstract representation of the syntactic structure of a program, omitting syntactical elements like brackets and punctuation.
- **Parse Tree**: A detailed tree diagram showing all the syntactical elements of a program, including brackets, punctuation, and keywords.
- **Annotated Parse Tree**: A parse tree enhanced with additional information like values, types, or variable bindings, useful for semantic analysis and code generation.



Types of 3-address Statements in intermediate Code

Assignment Statement	X = y op z
Assignment Instruction	X = op y
Copy Statement	X = y
Unconditional Jump	Goto L
Conditional Jump	If x relop y goto L
Procedure Call	Parm x1 Parm x2 . Parmxn Call p,n Return y
Array Statement	X = y[i] X[i] = y
Address and Pointer Assignment	X=&y X=*y *x=y

Q Consider the grammar with the following translation rules and E as the start symbol.

$E \rightarrow E_1 + T$ { E.nptr = mknode(E_1.nptr, +, T.ptr);}

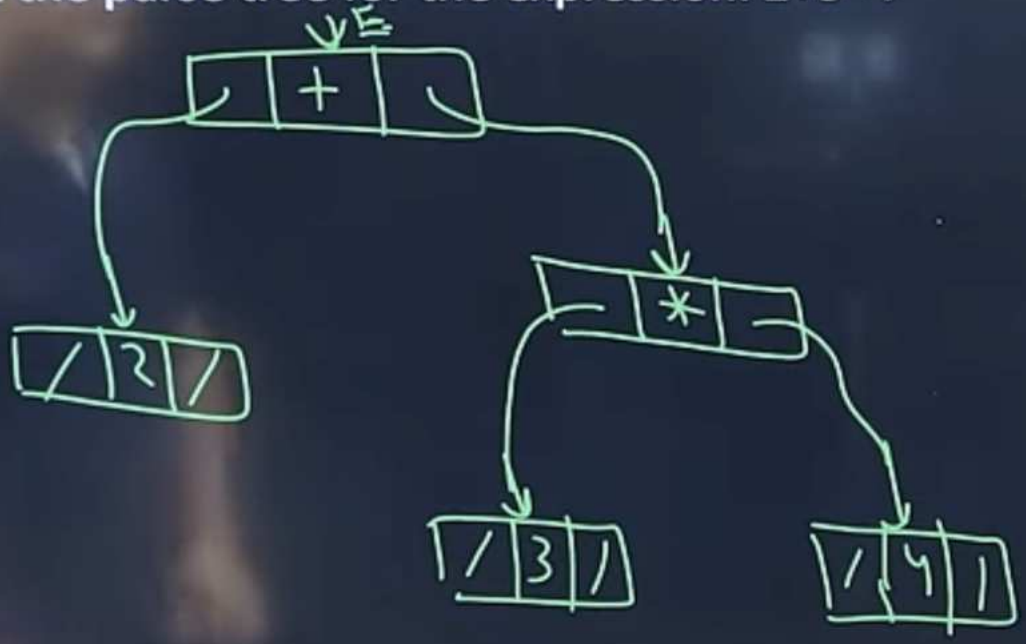
$E \rightarrow T$ { E.nptr = T.nptr }

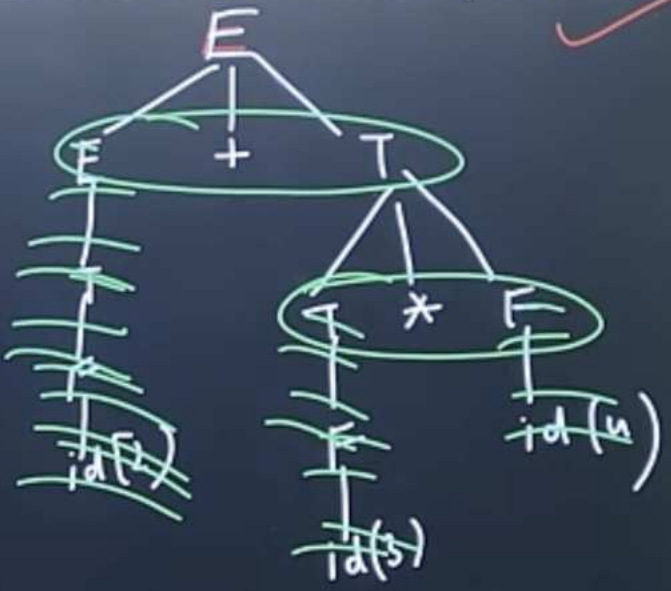
$T \rightarrow T_1 * F$ { T.nptr = mknode(T_1.nptr, *, F.ptr);}

$T \rightarrow F$ { T.nptr = F.nptr }

$F \rightarrow$ id { F.nptr = mknode(null, id name, null);}

Construct the parse tree for the expression: 2+3*4





Q Generate three address code for the following code :

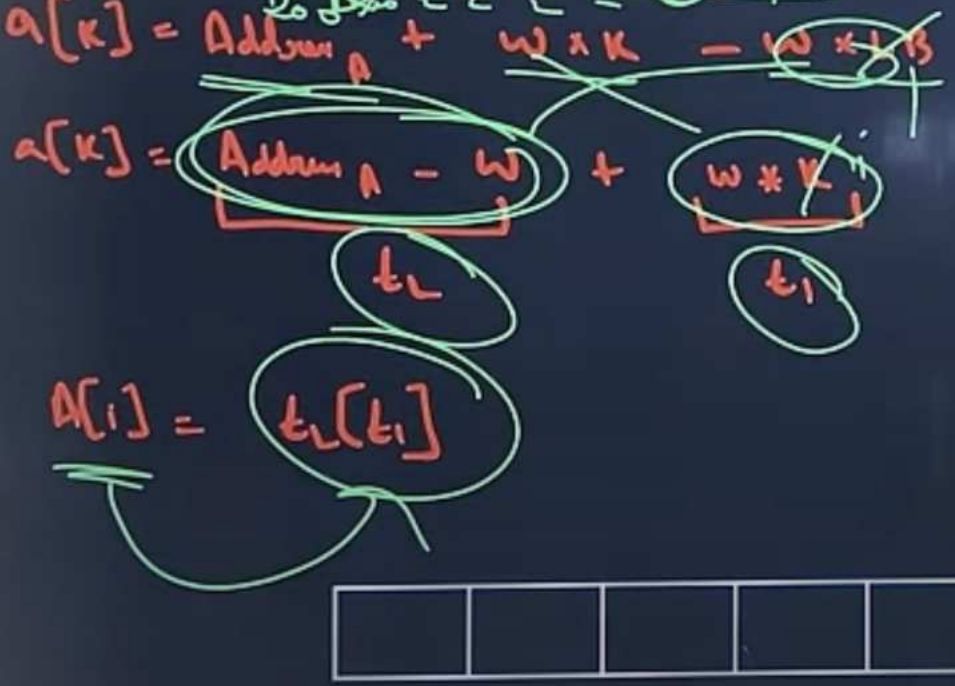
```
switch a + b
{
    case 1 : x = x + 1
    case 2 : y = y + 2
    case 3 : z = z + 3
    default : c = c – 1
}
```

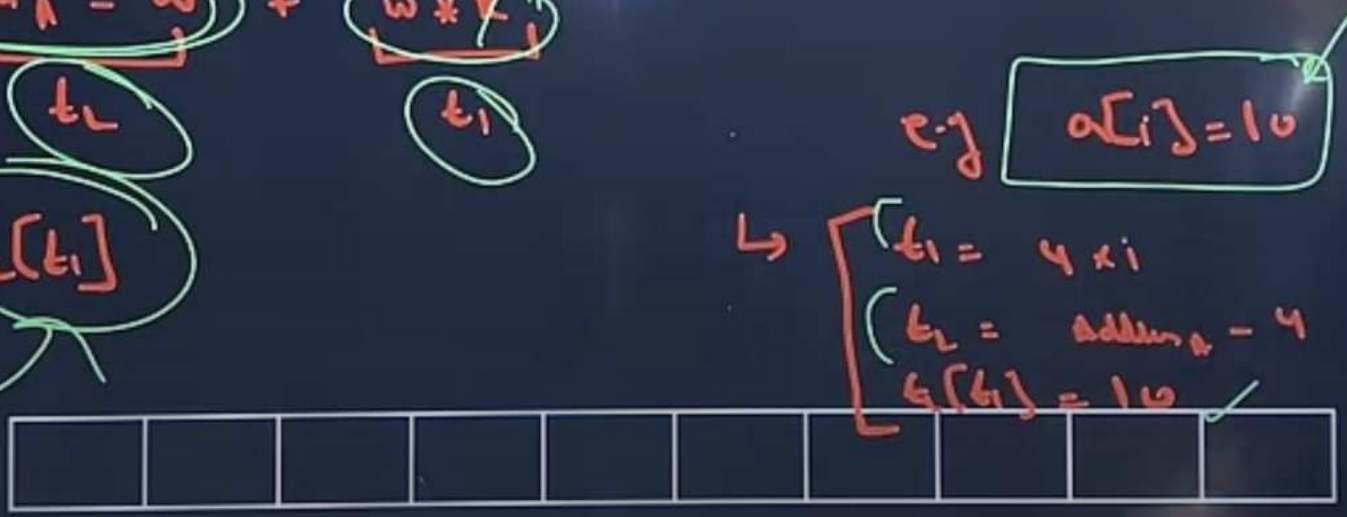
101: $t_1 = a + b$ goto 103
102 : goto 115
103 : t = 1 goto 105
104 : goto 107
105: $t_2 = x + 1$
106 : $x = t_2$
107 : if t = 2 goto 109
108 : goto 111
109: $t_3 = y + 2$
110 : $y = t_3$
111 : if t = 3 goto 113
112 : goto 115 113: $t_4 = z + 3$
114 : $z = t_4$
115: $t_5 = c - 1$
116 : $c = t_5$
117 : Next statement

One Dimensional array

- Address of the element at k^{th} index
  - a[k] = B + W*k
  - a[k] = B + W*(k – Lower bound)

- B is the base address of the array
- W is the size of each element
- K is the index of the element
- Lower bound index of the first element of the array
- Upper bound index of the last element of the array





Q Write the quadruples, triple and indirect triple for the following expression : **(x+y)* (y+z)+(x+y+z)** ?

$t_1 = x + y$

$t_2 = y + z$

$t_3 = t_1 + t_2$

$t_4 = t_1 + z$

$t_5 = t_3 + t_4$

Index	Operator	Arg1	Arg2
0	+	x	y
1	+	y	z
2	*	0	1
3	+	x	y
4	+	3	z
5	+	2	4

Pointer	Index
p0	0
p1	1
p2	2
p3	3
p4	4
p5	5

Triplet

	Operator	$Operand_1$	$Operand_2$
1)	+	a	b
2)	–	1	
3)	+	c	d
4)	*		3
5)	+		b
6)	+	5	c
7)	+	4	6

1) $t_1 = a+b$
2) $t_2 = -t_1$
3) $t_3 = c+d$
4) $t_4 = t_2 * t_3$
5) $t_5 = a+b$
6) $t_6 = t_5 + c$
7) $t_7 = t_4 + t_6$

...t wasted

Disadvantage

- Statement cannot be moved

Quadruples

	Operator	$Operand_1$	$Operand_2$	Result
1)	+	a	b	t_1
2)	-	t_1		t_2
3)	+	c	d	t_3
4)	*	t_2	t_3	t_4
5)	+	a	b	t_5
6)	+	t_5	c	t_6
7)	+	t_4	t_6	t_7

1) $t_1 = a+b$
2) $t_2 = -t_1$
3) $t_3 = c+d$
4) $t_4 = t_2 * t_3$
5) $t_5 = a+b$
6) $t_6 = t_5 + c$
7) $t_7 = t_4 + t_6$

3 Address Code

Types of 3 address codes

1) x = y operator z
2) x = operator z
3) x = y
4) goto L
5) A[i] = x
   y = A[i]
6) x = *p
   y = &x

x = -y
!y

3 Address Code

- Three-address code is a type of intermediate code where each instruction can have at most three operands and one operator, like a := b op c. It simplifies complex operations into a sequence of simple statements, supporting various operators for arithmetic, logic, or boolean operations.

a = b op c

(a+b) * (a + b + c)

Post fix

ab+ab+c+*

Three address code

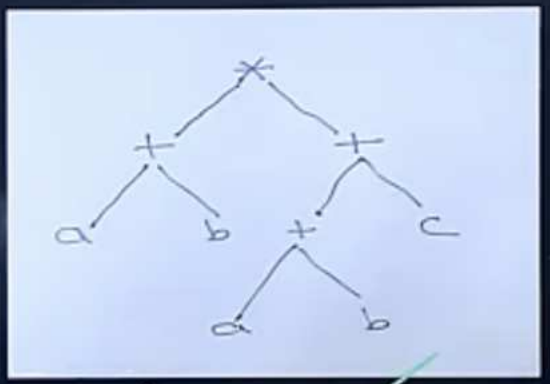
$t_1 = a+b$

$t_2 = a+b$

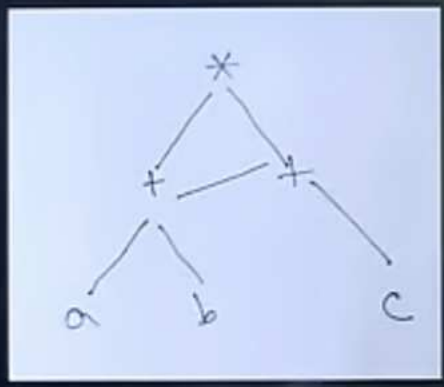
$t_3 = t_2 + c$

$t_4 = t_1 * t_3$

Syntax Tree



Direct Acyclic Graph



(1)

if c
then x
else
y

e ? x y → e x y ?

→

(2)

if a
then [if c-d
then a+c
else
a*c]
else
a+b

a ? [c-d ? a+c a*c] [a+b]

a ? [cd- ? ac+ ac*] ab+

a ? [cd- ac+ ac* ?] ab+

a cd-ac+ ac*? ab+?

Chapter-3 (SYNTAX-DIRECTED TRANSLATION)

Practice Question

Q Draw syntax tree for the arithmetic expressions : $a * (b + c) - d/2$. Also write the given expression in postfix notation ?

* /

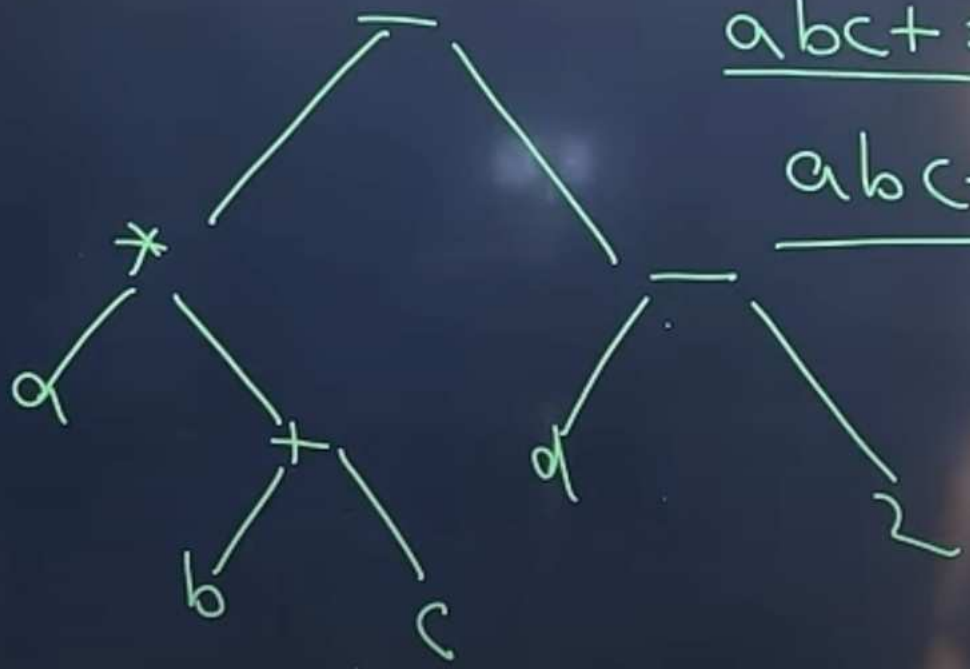
\+ −

a * (b * c) − d/2

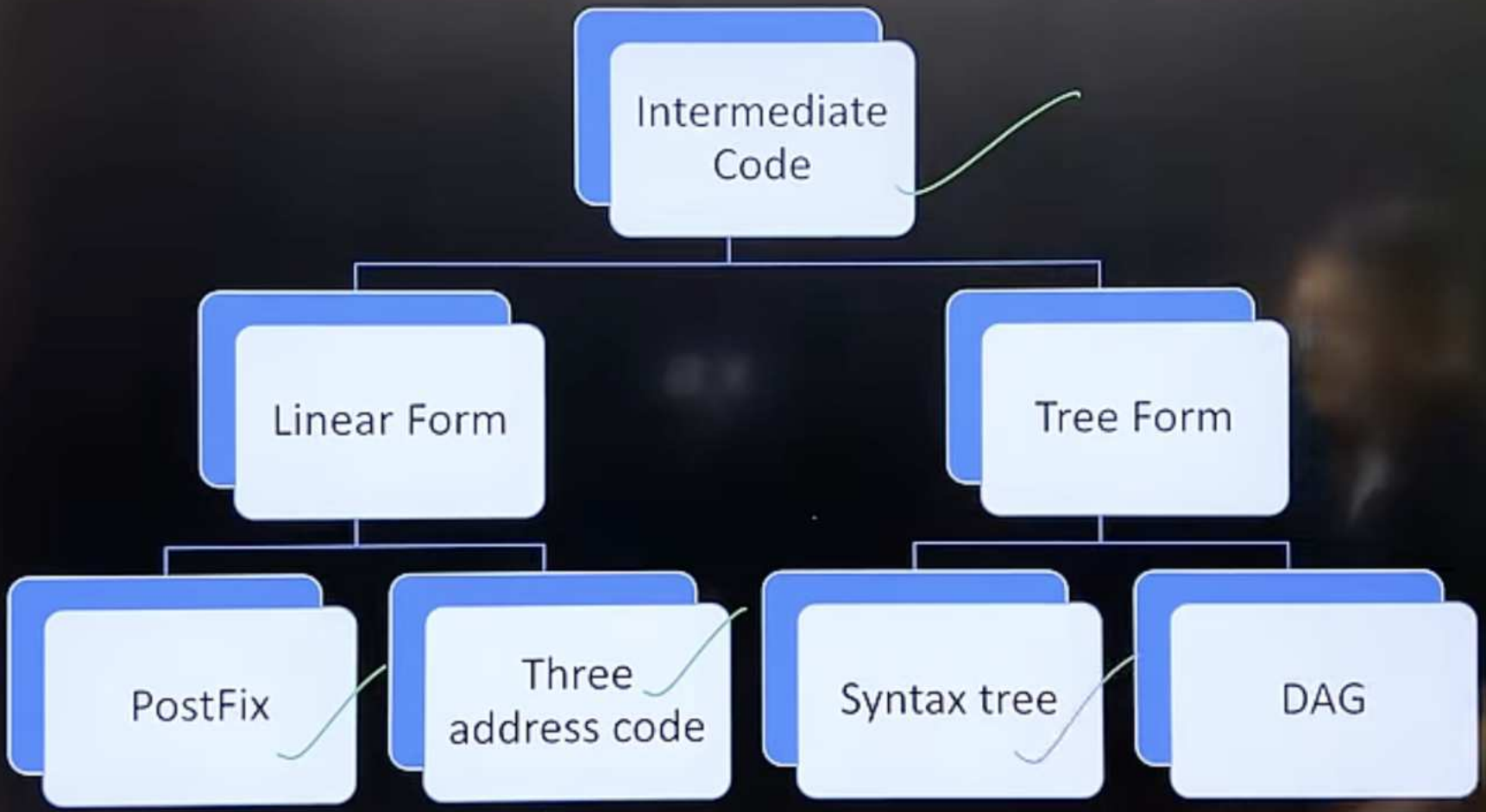
a * bc+ − d/2

abc+ * − d2/

abc+*d2/−



Intermediate Code Generation



Intermediate Code Generation

- Intermediate code generation in compilers creates a machine-independent, low-level representation of source code, facilitating optimization and making the compiler design more modular. This abstraction layer allows for:
  - **Portability**: Easier adaptation of the compiler to different machine architectures, as only the code generation phase needs to be machine-specific.
  - **Optimization Opportunities**: More efficient target code through optimizations performed on the intermediate form rather than on high-level source or machine code.
  - **Ease of Compiler Construction**: Simplifies the development and maintenance of the compiler by decoupling the source language from the machine code generation.

- $x = y + z * 60$

  $t_1 = z * 60$

  $t_2 = y + t_1$

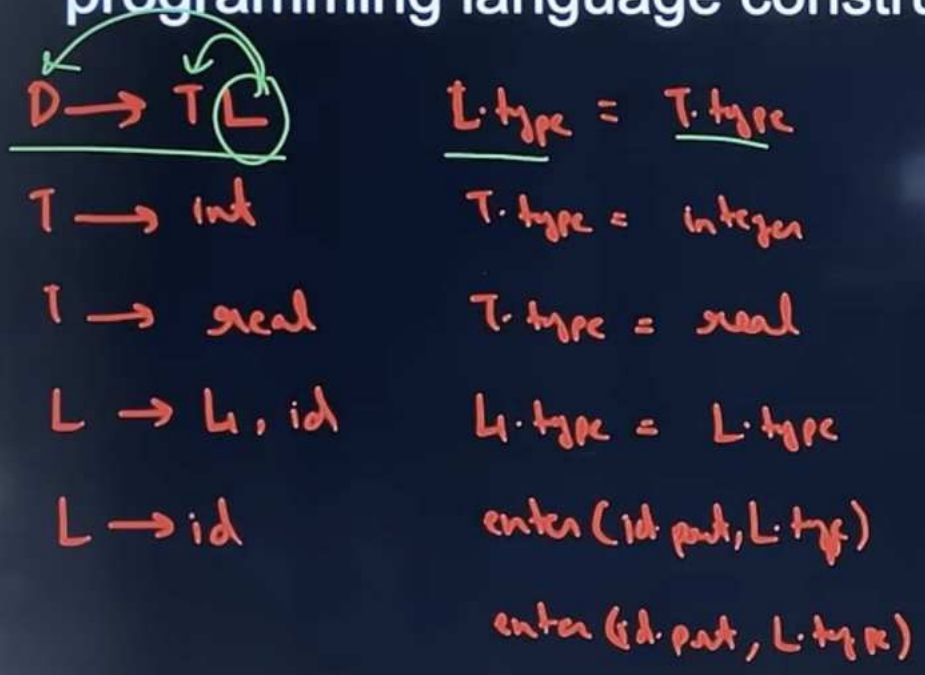
  $x = t_2$

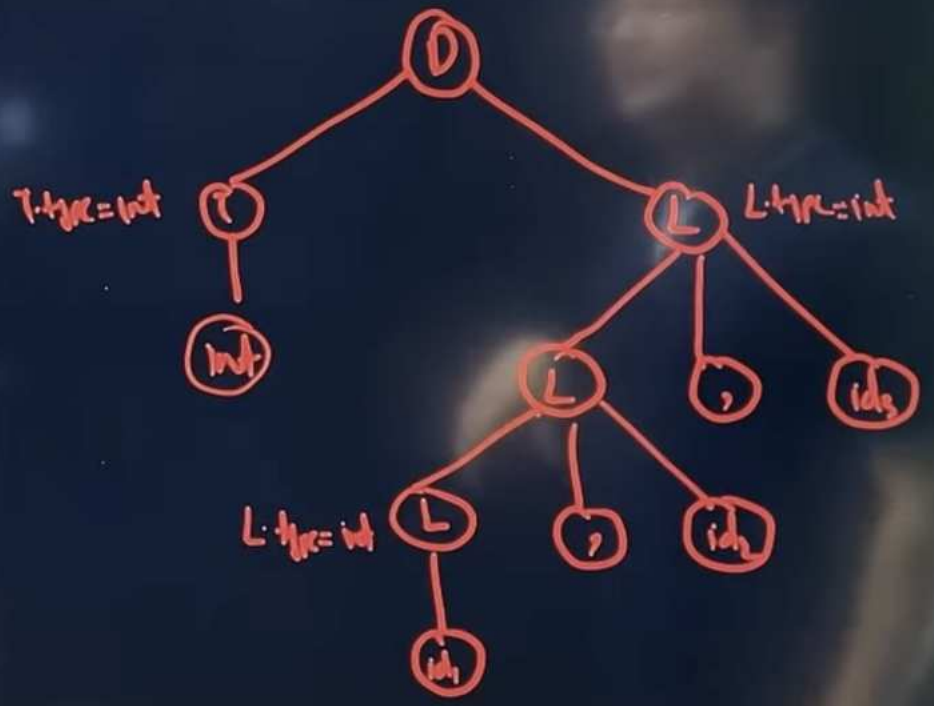
S-Attributed SDT	L-Attributes SDT
Uses only Synthesized attributes	Uses both inherited and synthesised attributes. Each inherited attribute is restricted to inherit either form parent or left sibling only.
Semantic actions are placed at extreme right on right end of production	Semantic actions are placed anywhere on right hand side of the production.
Attributes are evaluated during BUP	Attributes are evaluated by traversing parse tree depth first left to right.

Classification of Attributes

Synthesized Attributes	Inherited Attributes
Computed from the attribute values of a node's children in the parse tree.	Computed from the attribute values of a node's siblings and parent.
Used to pass information up the parse tree.	Used to pass information down or across the parse tree.
Examples include the evaluated value of an expression or the size of a data type.	Examples include the data type expected for a child node or the environment in which a node should be evaluated.
Often associated with bottom-up parsing techniques like LALR or SLR.	Often associated with top-down parsing techniques such as LL parsers.
They do not need context from parent nodes, only from children and the node itself.	They require context from parent or surrounding nodes to be computed.

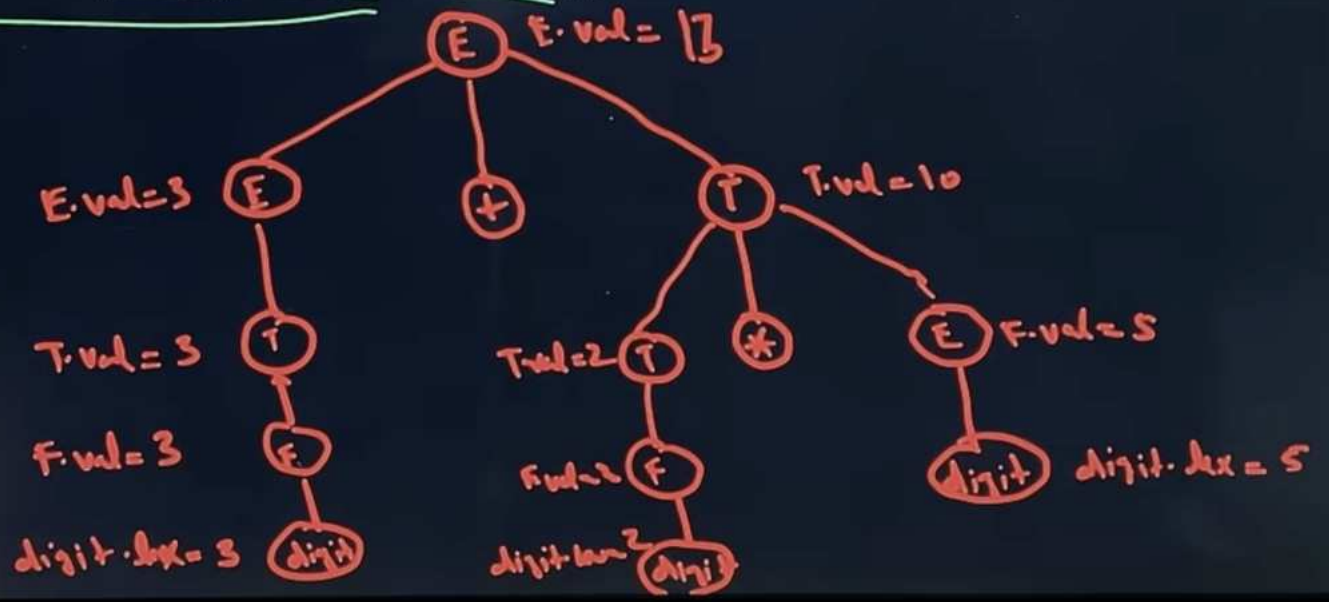
- **Inherited Attributes**: The attribute whose values are evaluated in terms of attribute value of parents & Left siblings is known as inherited attributes.
  - Inherited attributes are convenient for expressing dependence of a programming language construct on the context in which it appears.





- Synthesized attributes are derived from a node's children within a parse tree, and a syntax-directed definition relying solely on these attributes is termed as S-attributed.
- S-attributed definitions allow parse trees to be annotated from the leaves up to the root, enabling parsers to directly evaluate semantic rules during the parsing process.

$A \rightarrow XYZ \{A.S = f(X.S / Y.S / Z.S)\}$



Attributes

- Attributes attach relevant information like strings, numbers, types, memory locations, or code fragments to grammar symbols of a language, which are used as labels for nodes in a parse tree.
- The value of each attribute at a parse tree node is determined by semantic rules associated with the production applied at that node, defining the context-specific information for the language construct.

$X \longrightarrow Y + Z$

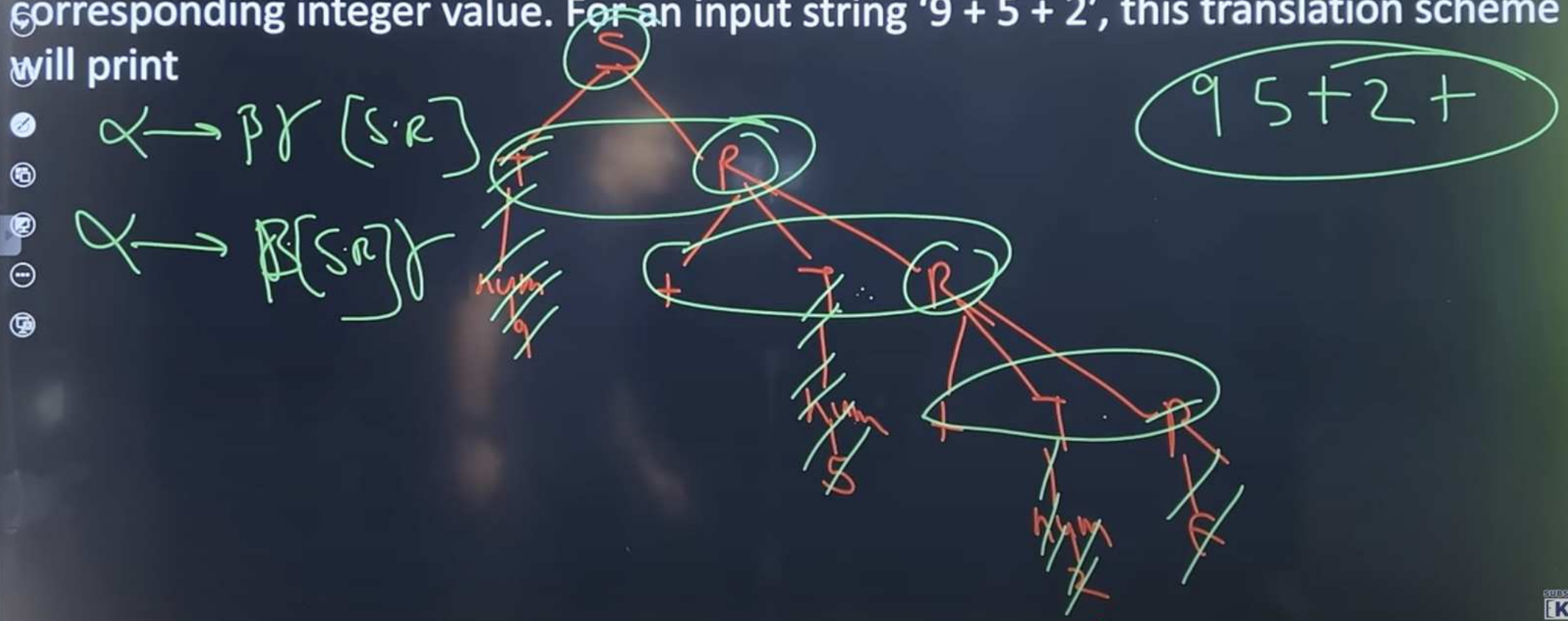
Chapter-3 (SYNTAX-DIRECTED TRANSLATION) Practice Question(Post fix Conversion

Q Consider the translation scheme shown below S → T R

R → + T {print ('+');} R / ε

T → num {print (num.val);}

Here num is a token that represents an integer and num.val represents the corresponding integer value. For an input string '9 + 5 + 2', this translation scheme will print



Chapter-3 (SYNTAX-DIRECTED TRANSLATION) Practice Question(Post fix Conversion

Q Consider the grammar with the following translation rules and E as the start symbol.

$E \rightarrow E_1 + T$ {print ('+');}

$E \rightarrow T$

$T \rightarrow T_1 * F$ {print ('*');}

$T \rightarrow F$

$F \rightarrow$ num {print ('num.val');}

Construct the parse tree for the string 2 + 3 * 4, and find what will be printed.

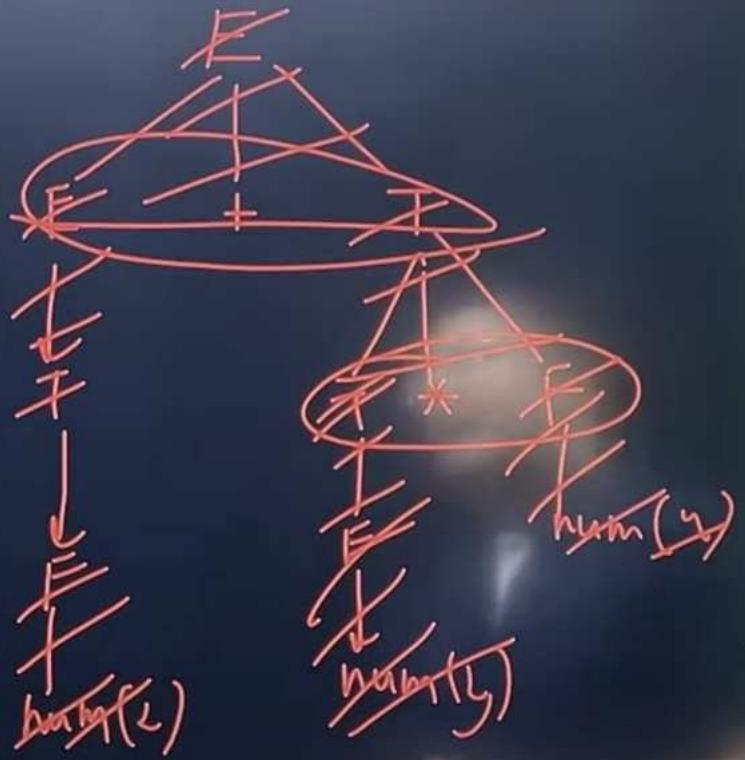
a+b

ab+

2+3*4

2+34*

234*+



2 3 4 * +

Q Consider the grammar with the following translation rules and E as the start symbol.

$E \rightarrow E_1 + T$ {print ('+');}

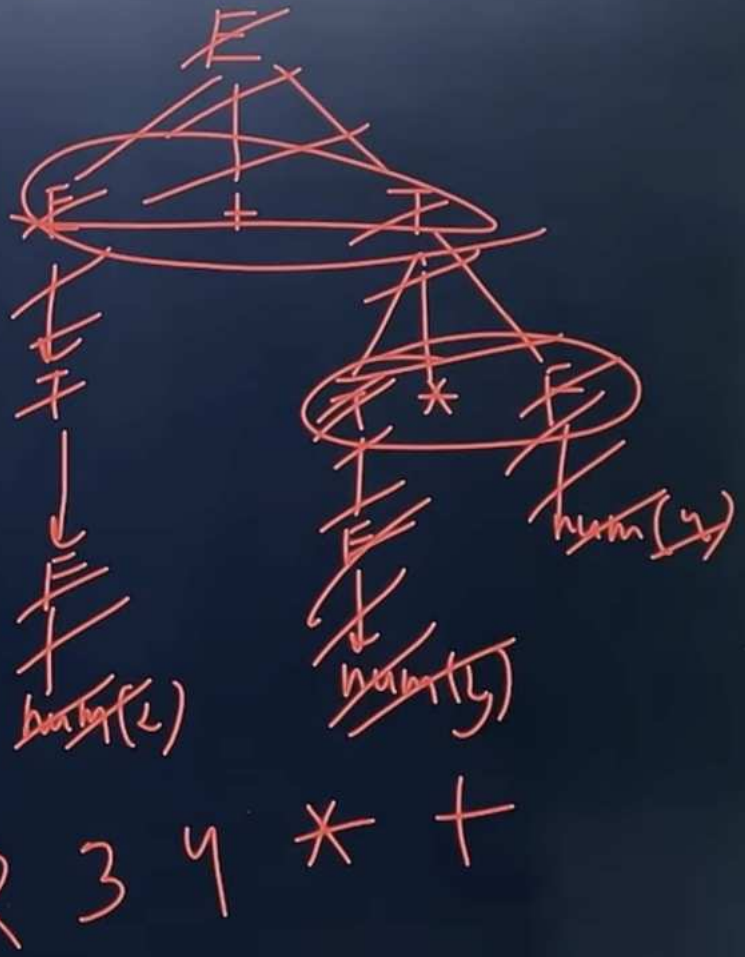
$E \rightarrow T$

$T \rightarrow T_1 * F$ {print ('*');}

$T \rightarrow F$

$F \rightarrow$ num {print ('num.val');}

Construct the parse tree for the string 2 + 3 * 4, and find what will be printed.



Q Consider the grammar with the following translation rules and E as the start symbol.

$E \rightarrow E_1$ # T { E.value = E_1.value * T.value }

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

$T \rightarrow T_1$ & F { T.value = T_1.value + F.value }

$T \rightarrow F$ { T.value = F.value }

$F \rightarrow$ num { F.value = num.value }

Compute E.value for the root of the parse tree for the expression: 2 # 3 & 5 # 6 & 4.

2 # 3 & 5 # 6 & 4

2 * 3 + 5 * 6 + 4

2 * 8 * 10

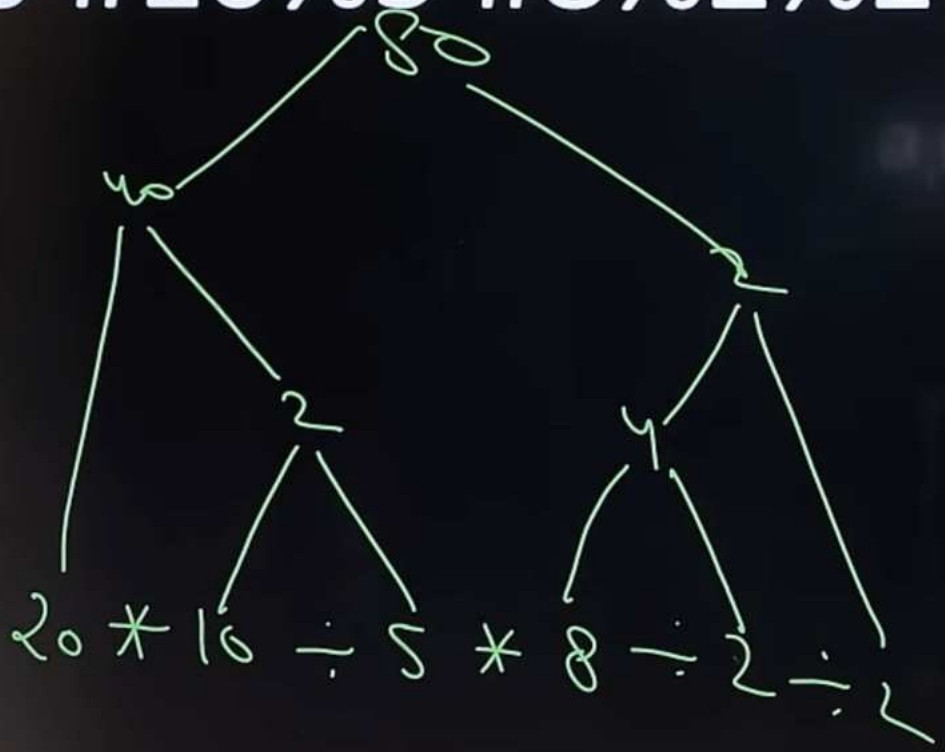
16 * 10 = 160

* (L → R)

& + (L → R)

Q Consider the following grammar along with translation rules. Here # and % are operators and id is a token that represents an integer and $id._{val}$ represents the corresponding integer value. The set of non-terminals is {S,T, R, P} and a subscripted non-terminal indicates an instance of the non-terminal.Using this translation scheme, the computed value of $S._{val}$ for root of the parse tree for the expression 20 #10%5 #8%2%2 is _____.

20 #10%5 #8%2%2



$S \rightarrow S_1 \# T$	$\{S._{val} = S_{1}._{val} * T._{val}\}$
$S \rightarrow T$	$\{S._{val} = T._{val}\}$
$T \rightarrow T_1 \% R$	$\{T._{val} = T_{1}._{val} \div R._{val}\}$
$T \rightarrow R$	$\{T._{val} = R._{val}\}$
$R \rightarrow id$	$\{R._{val} = id._{val}\}$

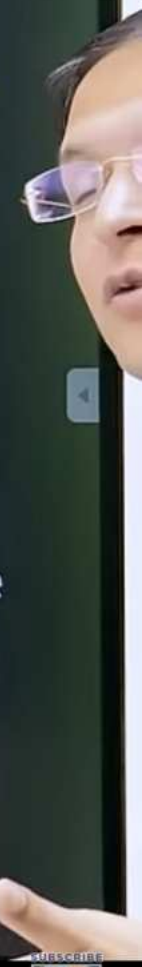
*, ÷ (L → R)

Semantic Analysis

- Grammar + Semantic Rule + Semantic Actions = Syntax Directed Translation. SDT is the generalization of CFG.

- With grammar we give meaningful rules, and apart from semantic analysis SDT can also be used to perform things like
  - Code generation
  - Intermediate code generation
  - Value in the symbol table
  - Expression evaluation
  - Converting infix to post fix

- Things can be done in parallel to parsing...so with semantic action and rule parsers become much powerful



Semantic Analysis

Input and output: Semantic analysis takes an Abstract Syntax Tree (AST) generated by the syntax analysis phase as its inputs.

Process: It performs type checking, scope resolution, and validates semantic consistency, ensuring that the operations and expressions in the source code ar according to the language's rules and semantics.