# Compiler Module 4 Important Questions

## 1. What do you mean by Syntax Directed Translation and SDD?

### Syntax directed translation

- Syntax directed translation scheme (SDT) is a context free grammar with program fragments embedded within production bodies
- The Program fragments are called semantic actions and can appear at any position within a production body
- By Convention we place curly braces around actions, if branches are needed as grammar symbols, then we quote them

**Simpler explanation**

Syntax directed translation is basically a set of rules with secret commands tucked inside them, marked with curly braces, and if we need to talk about special symbols like curly braces, we put quotation marks around them to avoid confusion.

Think of a grammar like a set of rules for writing sentences in a language. In the case of SDT, this grammar includes special instructions, which we'll call "program fragments," embedded within those rules.

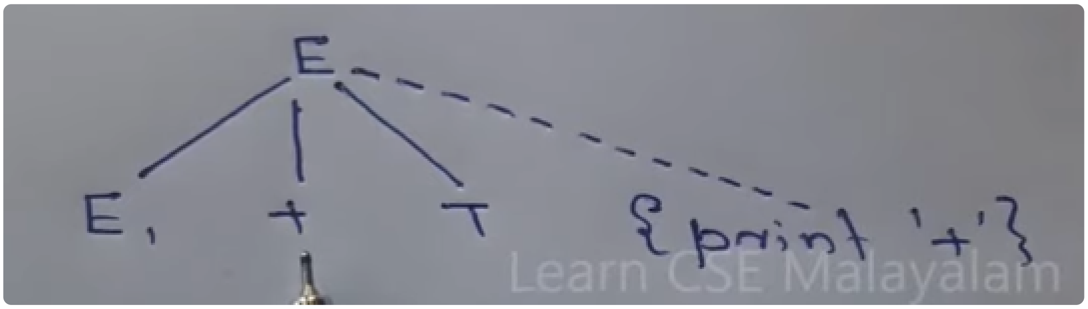
These program fragments are like hidden commands that tell the the computer what to do when it reads or processes different parts of the instructions.

### Example

- Example 1
  - Consider the production
    - E -> E1 + T {print '+'}
    - Here the action is {print '+'}
- Example 2
  - {print '{'}

### Implementation

- Any syntax directed translation can be implemented by first building a parse tree and then performing the actions in a left to right depth first order
- An extra leaf is constructed for semantic action

- 

- An action may be placed at any position within the body of a production. Its performed immediately after all symbols to its left are processed
- Example
    - B -> x {a} Y
    - Here X is processed first

### Postfix SDTs

- SDT's with all actions at the right end of the production bodies are called postfix SDTs

## 2. Differentiate S and L attributed definition with example.

- What is SDT?
    - Grammar production is associated with semantic rules
    - Example
        - Grammar Production = A -> BC
        - Semantic rule = {f(b.v,c.v)}
    - Grammar symbols are associated with some attributes
        - Like Type, Value
        - These attributes are evaluated with semantic actions, which are associated with grammar rules
    - These attributes can be
        - Synthesized Attribute
            - Can be evaluated from children
            - if A -> BC is a production
                - Then the semantic rule is
                    - Assume V is an attribute associated with non terminals
                    - {A.V = f(B.V,C.V)}

    - Inherited Attribute
        - Values can be inherited from parents or siblings
        - If A -> BCD is a production
        - The semantic rules are
            - B.V = A.V
                - B value is derived from parent A
            - C.V = D.V
                - C value is derived from sibling D
            - D.V = C.V
                - D value is derived from sibling C
        -

- There are 2 types of Syntax Directed Translation
- Those are
    - S-Attributed
    - L-Attributed

## S-Attributed

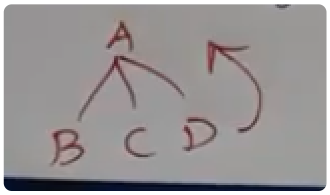
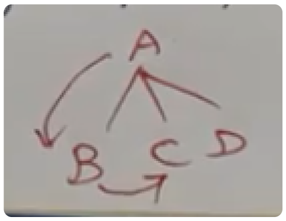
- S stands for synthesized
- If SDT uses only synthesized attributed its called as S-Attributed SDT
- S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

## L-Attributed

- L stands for one parse from left to right
- L-Attributed definitions contain both synthesized and inherited attributes
- Ie, If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from parent and left siblings only, it is called as L-attributed SDT.
    - For Example: A -> BCD
        - Here the semantic rules are
            - B.V = A.V
                - B inherits from parent
            - C.V = B.V

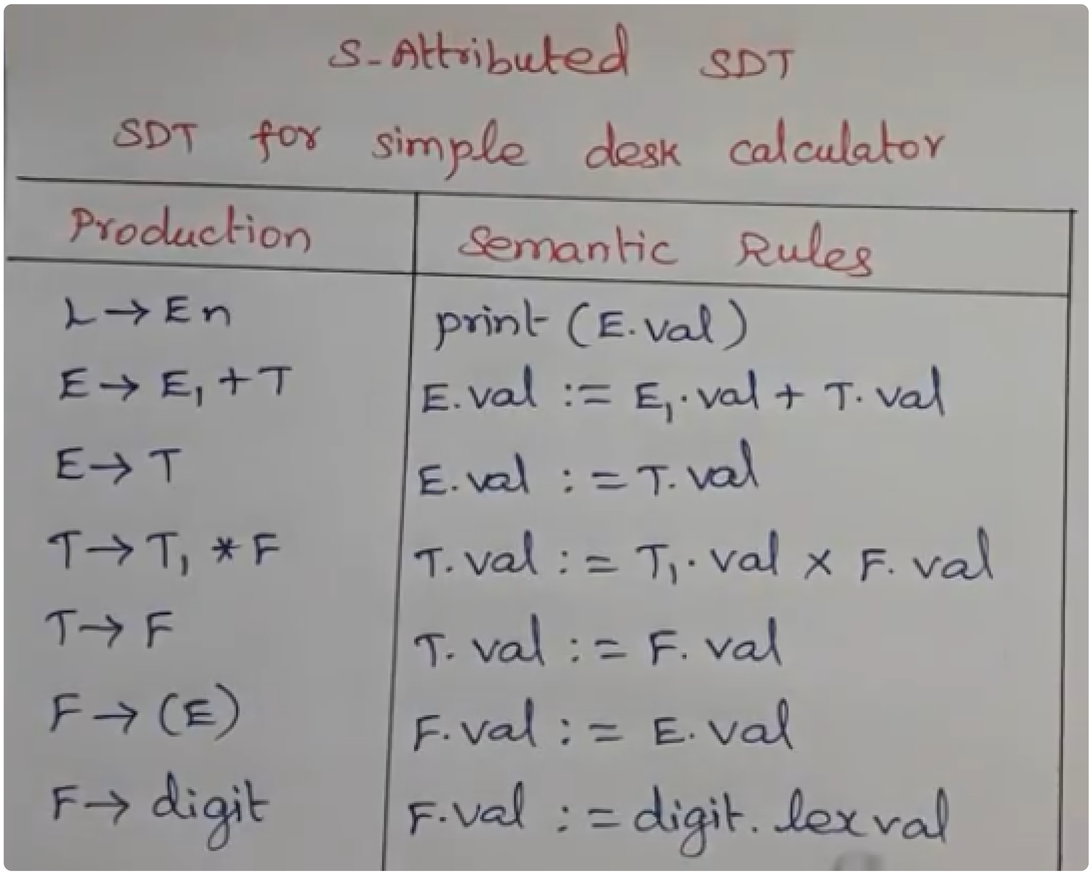
        - C inherits from left sibling, which is B
    - D.V = B.V
        - D inherits from left sibling which is B

- If an attribute is S attributed , it is also L attributed.

| **S-Attributed SDT** | **L-Attributed SDT** |
|---|---|
| Uses only synthesized Attribute | Uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from parent and left siblings only, |
| Semantic actions are placed at right end of production, Like so A-> BC {Semantic actions} | Semantic actions are placed anywhere on the RHS, Like so A -> {Semantic action}BC<br>A -> B {Semantic Action}C |
| Attributes are evaluated during Bottom up parsing | By traversing parse tree, depth first, left to right |
|  |  |

## S-Attributed SDT Example

### SDT for simple desk calculator



S-Attributed SDT

SDT for simple desk calculator

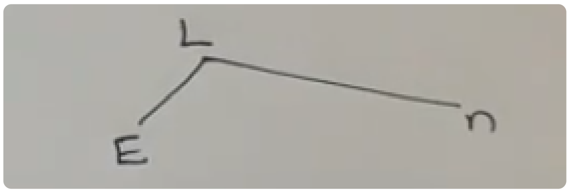
| Production | Semantic Rules |
|---|---|
| $L \to En$ | print (E.val) |
| $E \to E_1 + T$ | $E.val := E_1.val + T.val$ |
| $E \to T$ | $E.val := T.val$ |
| $T \to T_1 * F$ | $T.val := T_1.val \times F.val$ |
| $T \to F$ | $T.val := F.val$ |
| $F \to (E)$ | $F.val := E.val$ |
| $F \to digit$ | $F.val := digit.lexval$ |

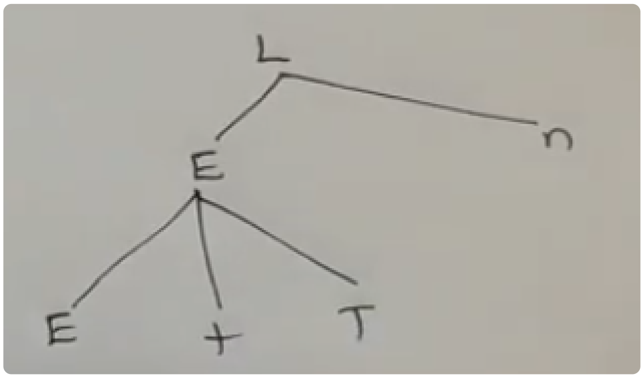
- Lets go over this line by line
  - L -> En
    - n represents newline
    - Used to print value of E
  - E -> E1 + T
    - Value of E is evaluated from children
    - Value of parent calculated from children
  - E -> T
    - Equating E with T
  - Similarly for the remaining
- Now we need to construct annotated parse tree for the example

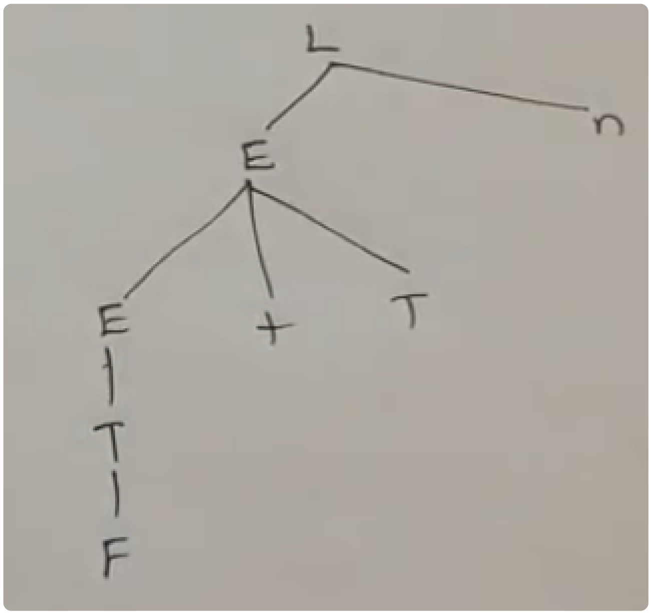
**2 + 4 * 5n**

**Drawing the Productions one by one**

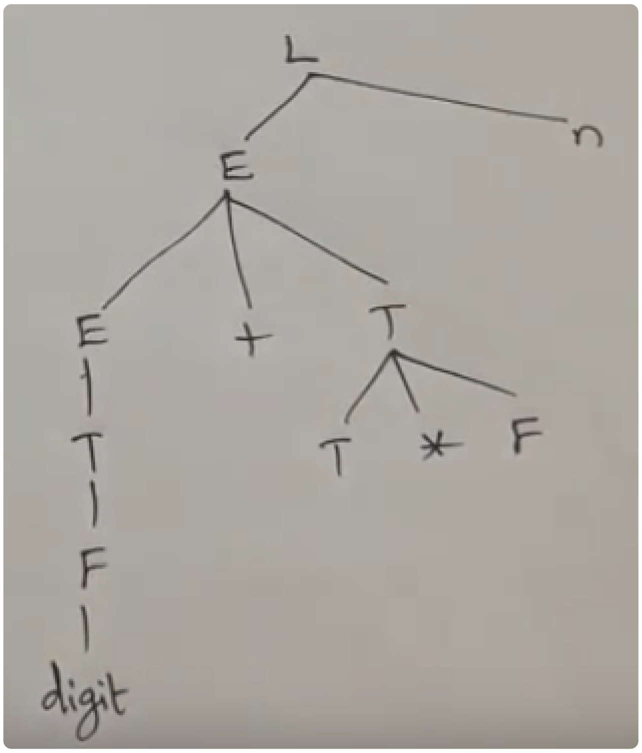
1. **L -> En**

    1. 

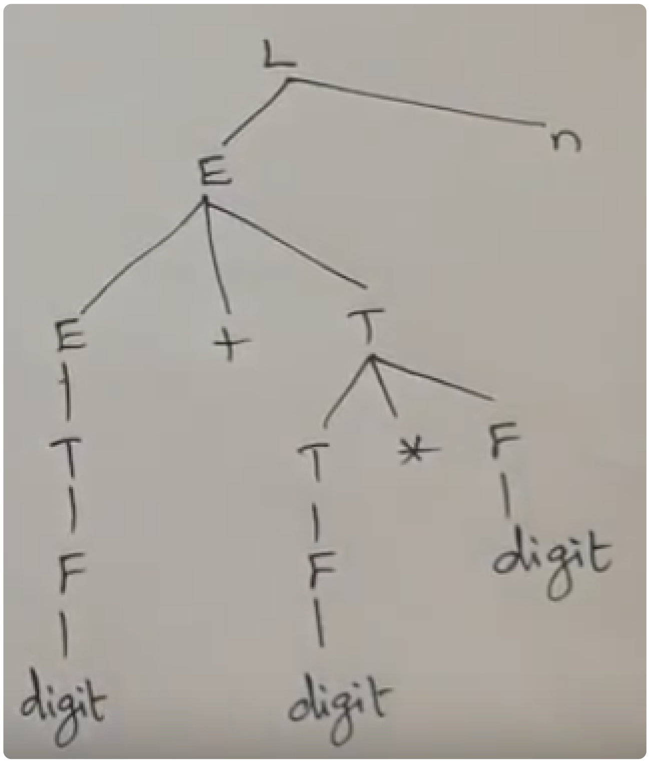
2. **E -> E1 + T**
    1. 

3. **E -> T and T -> F**
    1. 

4. **F->Digit and T->T1 * F**

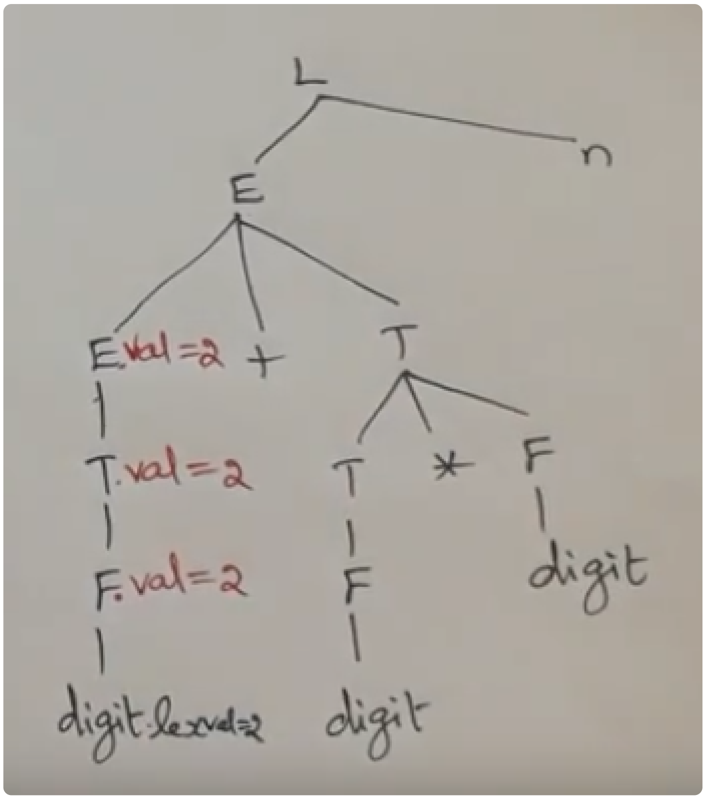
1. 

5. **T-> F and F-> digit**

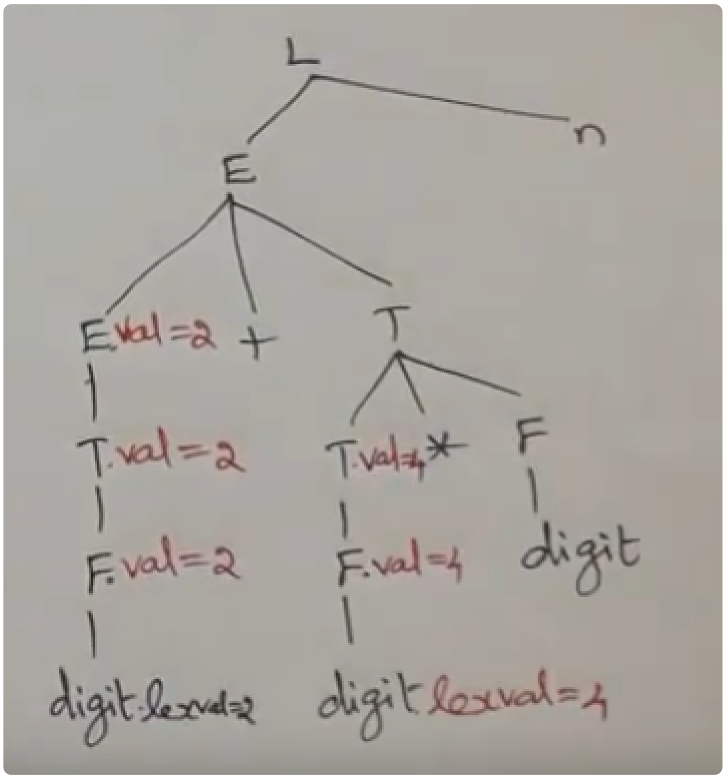
1. 

#### Applying the semantic rules

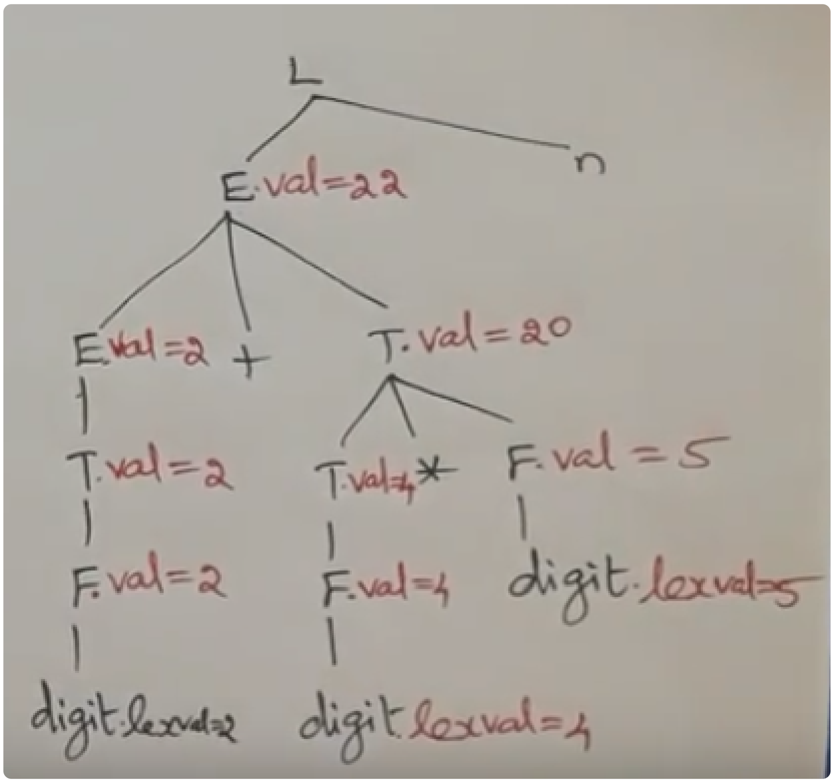
1. **F.val = digit.lexval**
   1. Here our first digit in the expression is 2, so digit.lexval = 2
   2. F.val = 2
   3. Above F we have T, we need to get the rule for that

2. **T.val = F.val**
    1. T.val = 2
    2. Above T we have E
3. **E.val = T.val**
    1. E.val = 2
    2. Next we have T, but the value is not calculated for that yet, so lets go down and get the digit
    3. 

4. **F.val = digit.lexval**
    1. Second digit is 4, digit.lexval = 4
    2. F.val = 4
5. **T.val = F.val**
    1. T.val = 4
    2. Next we Have F, on the other side, we need to get its value

    3. 

6. **F.val = digit.lexval**
    1. Next digit is 5
    2. digit.lexval = 5
    3. F.val = 5
7. **T=T1.val x F.val**
    1. T.val = 4 x 5
    2. T.val = 20
8. **E = E1.val + T.val**
    1. E = 2 + 20
    2. E.val = 22

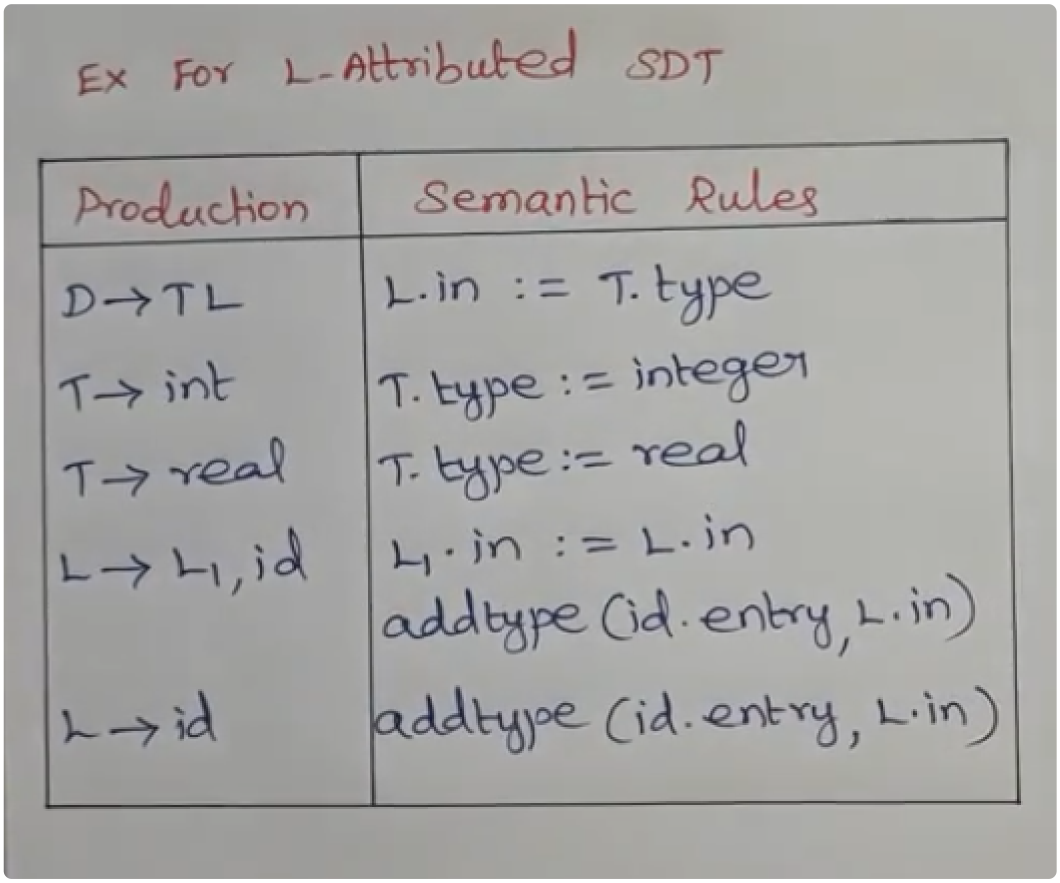
3. 

9. Next the print(E.val) is executed and 22 is printed

## L-Attributed SDT Example

We need to get the following list of identifiers

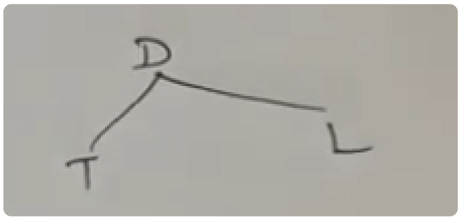
**int id1,id2,id3**

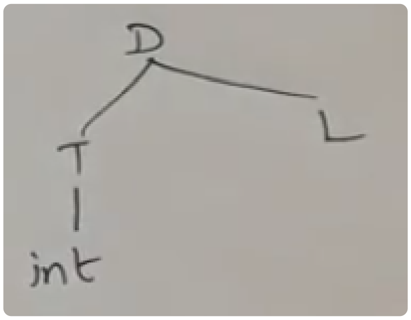


Ex For L-Attributed SDT

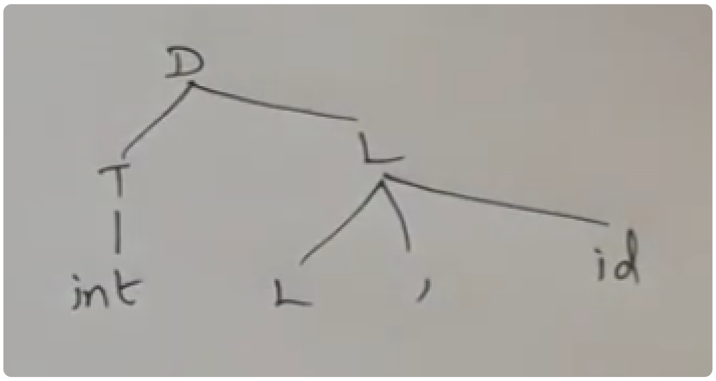
| Production | Semantic Rules |
|---|---|
| D→TL | L.in := T.type |
| T→int | T.type := integer |
| T→real | T.type := real |
| L→$L_1$,id | $L_1$.in := L.in<br>addtype (id.entry, L.in) |
| L→id | addtype (id.entry, L.in) |

### Drawing the production

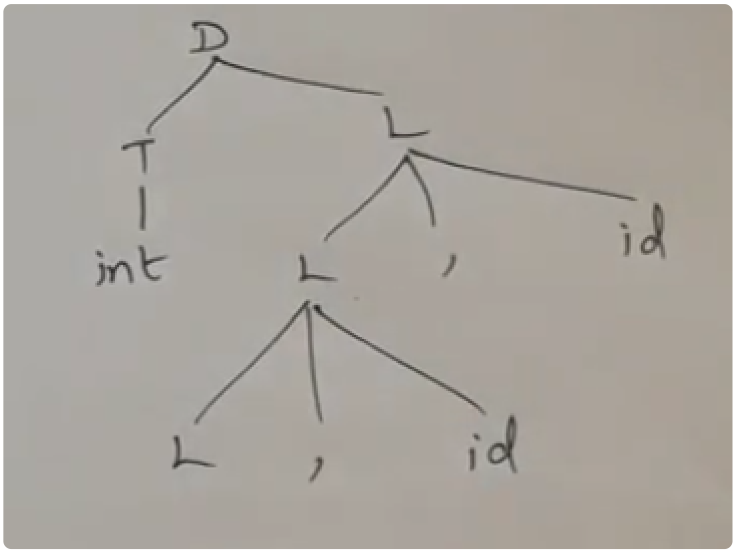
1. D -> TL
    1. 

2. T -> int
    1. 

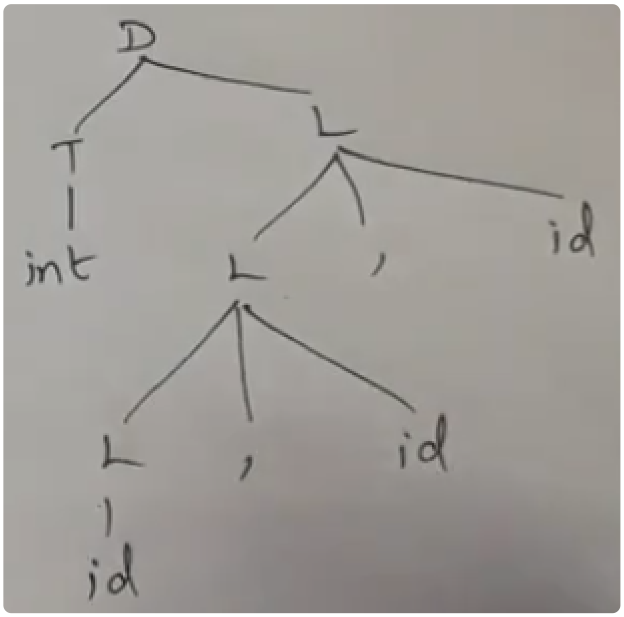
3. T -> real
    1. Ignoring, since its int id1,id2,id3 and not real
4. L -> L1,id

1. 

5. L -> L1,id

1. 

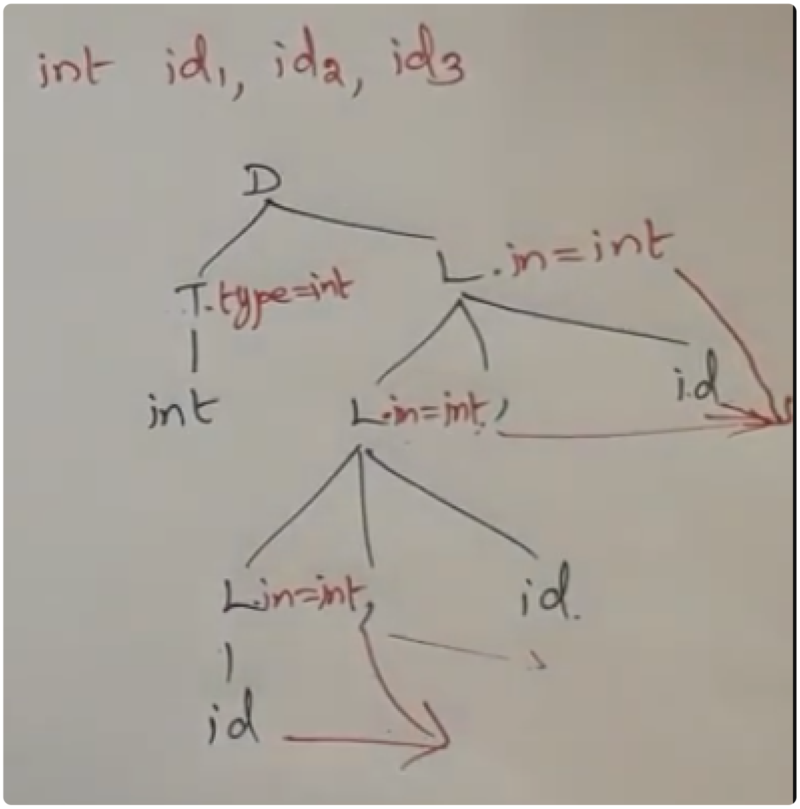
6. L -> id

1. 

2. Here We only need 3 ids, so we are stopping with L -> id

### Applying the semantic rules

1. T.type = integer

    1. Starting from the left most, which is T
    2. Next one is D
2. L.in = T.type
    1. Here T.type is int
    2. So L.in = int
    3. Going to the Left, next is L -> L1,id
3. L1.in = L.in
    1. L1.in = int, inheriting value from parent
    2. Next one is also L
4. L1.in = L.in
    1. L1.in = int
5. 

---

# 3. Write down the SDD for a simple desk calculator.

## SDD for calculator

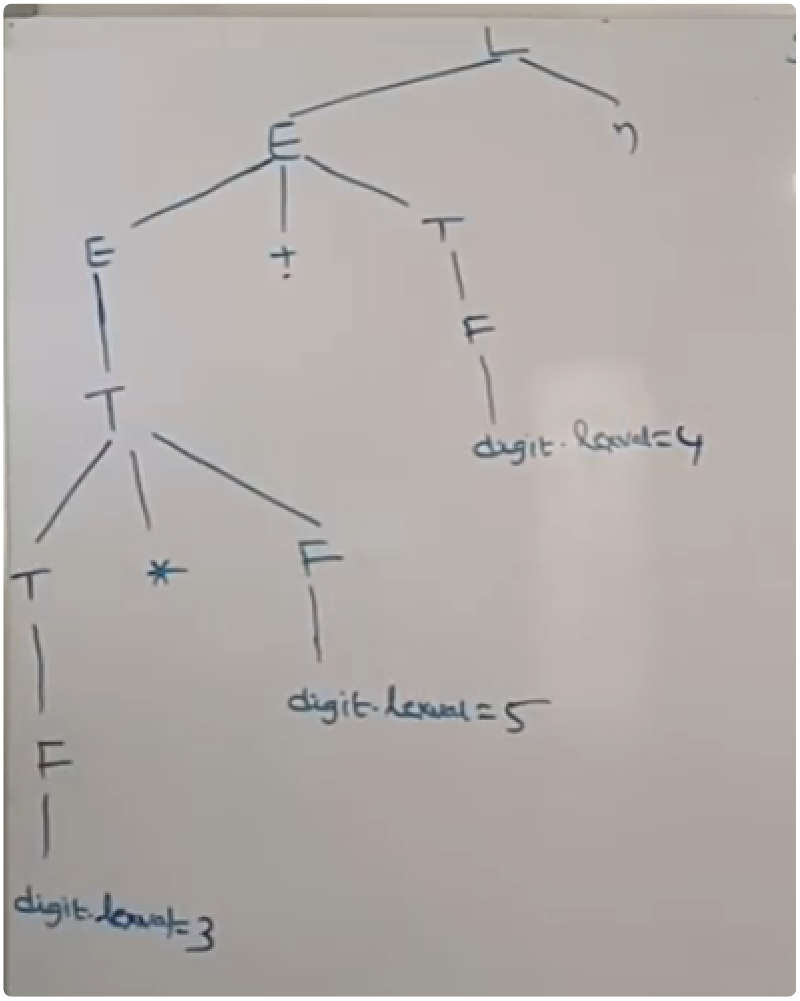
- Syntax directed definition
- It performs additon, multiplication

Evaluating the following expresiion

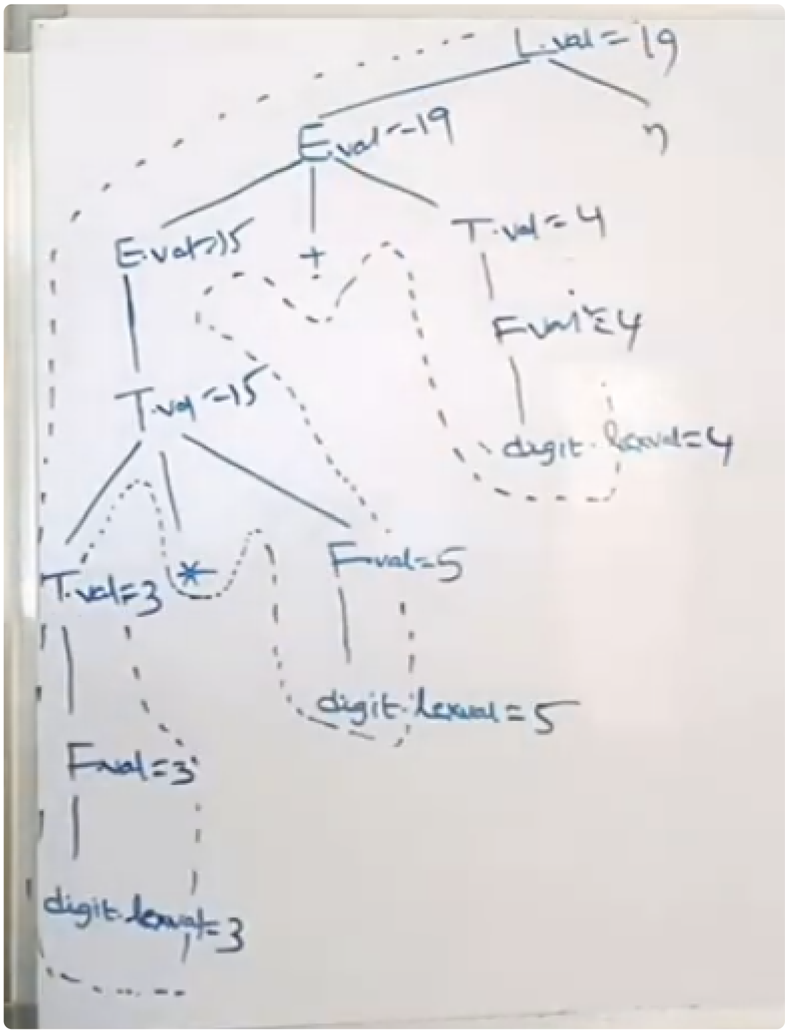
for 3 * 5 + 4

| Producers | Semantic Rules | Explanation |
|---|---|---|
| L -> En | L.val = E.val | Here n means newline |
| E -> E1+T | E.val = E.val + T.val | |
| E -> T | | |
| E -> T | E.val = T.val | |
| T -> T1 * F | T.val = T1.val | |
| T -> F | T.val = F.val | |
| F -> digit | F.val = digit.lexval | |
| | | |

- To solve the problem, we need to construct parse tree
- Then we need to make annotated parse tree



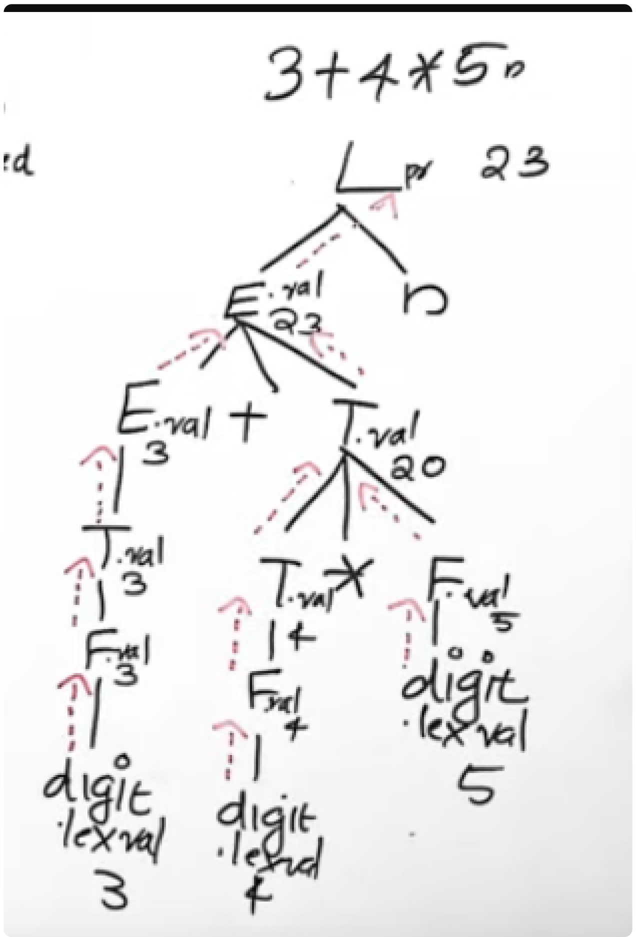
Annotated parse tree



## 4. Explain bottom-up evaluation of S attributed definition with example.

### Bottom up evaluation of S attributes

Consider this example



- We start from 3, inserting it
- Then we insert the Letter F
- Next Letter is T, Reducing T-> F
- Reducing E -> T

Here the stack is divided by space
for example +- and E3 and separate elements of stack

| i/p | stack | Production used |
|---|---|---|
| 3+4 * 5n | _ | _ |
| +4 * 5n | 3 | - |
| +4 * 5n | F3 | F-> digit |
| +4 * 5n | T3 | T -> F |
| +4 * 5n | E3 | E -> T |
| 4 * 5n | +- E3 | - |
| * 5n | 4- +- E3 | - |
| * 5n | F4 +- E3 | F -> digit |

| i/p | stack | Production used |
|---|---|---|
| * 5n | T4 +- E3 | T -> F |
| 5n | * - T4 +- E3 | - |
| n | 5- * - T4 +- E3 | - |
| n | F5 * - T4 +- E3 | - |
| n | T20 +- E3 | T -> T * F |
| n | E23 | E -> E + T |
| _ | n E23 | L -> En |

## 5. What do you mean by type checking.

- Type checking ensures that variables and expressions in a program are used according to their specified types. Here's a simpler breakdown:
- The process of verifying and enforcing the constraints of types is called type checking.
    - This may occur either at compile-time (a static check) or run-time (` dynamic check).
    - Static type checking is a primary task of the semantic analysis carried out by a compiler.
    - If type rules are enforced strongly (that is, generally allowing only those automatic type conversions which do not lose information), the process is called strongly typed, if not, weakly typed.

## 6. Explain the function used for writing SDD for the construction of the syntax tree.

- Following functions are used to create syntax tree
    - mknode(op,left,right):
        - Creates an operator node with label op and two fields containing pointers to left and right
    - mkleaf(id,entry)
        - Creates an identifier node with label id and a field containing entry, a pointer to the symbol table entry for identifier
    - mkleaf(id,entry)

  - Creates a number node with label num and a field containing val, the value of the number
- These functions return a pointer to a newly created node

## 7. What is THREE ADDRESS CODE?

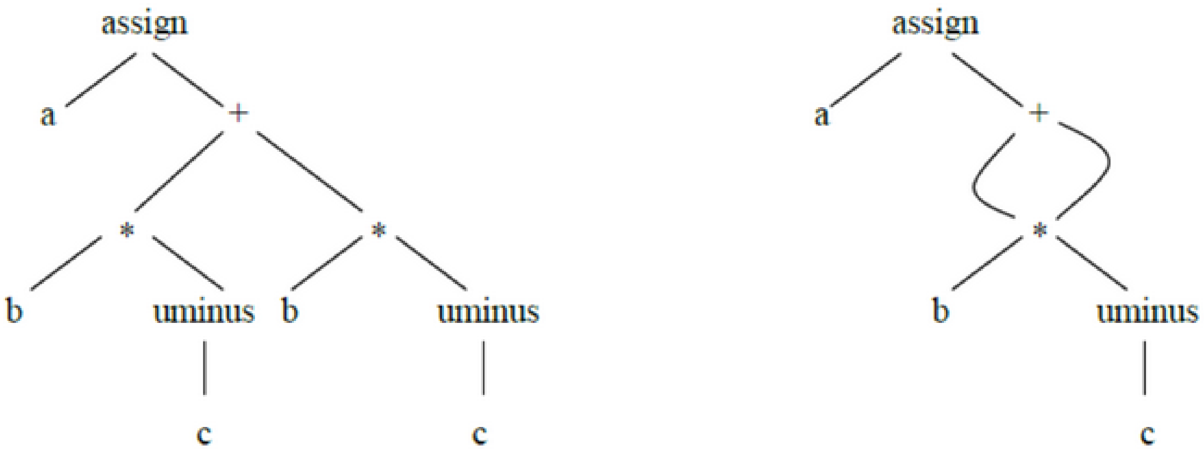
- In Three address statement, at most 3 addresses are used to represent any statement.
- The reason for the term “three address code” is that each statement contains 3 addresses at most. Two for the operands and one for the result.

## 8. What is DAG?

- Its a direct acyclic graph
- Graphical Intermediate Representation
- Dag also gives the hierarchical structure of source program but in a more compact way because common sub expressions are identified

**EXAMPLE**

**a=b*-c + b*-c**



**(a) Syntax tree**

**(b) Dag**

- Here b and uminus is a coommon subexpression which is being added, they are combined together in DAG