# Unit – 3
# Semantic Analysis & Syntax Directed Translation

Mrs. Ruchi Sharma

ruchi.sharma@bkbiet.ac.in

# Topics to be covered

- Introduction of Syntax-directed translation
- Syntax-Directed Definitions
- Inherited Attributes
- Synthesized Attributes
- Evaluating an SDD at the Nodes of a Parse
- Annotated Parse Tree
- S- Attributed Translation & L- Attributed Translation
- Example of STD
- Type  checking
- Intermediate Code generation
- Types of errors
- Error recovery strategies

# Syntax Directed Translation

# Syntax-directed translation

- Syntax-directed translation (SDT) refers to a method of compiler implementation where the source language translation is completely driven by the parser.
- **The parsing process and parse trees are used to direct semantic analysis and the translation of the source program.**
- We can augment grammar with information to control the semantic analysis and translation. Such grammars are called **attribute grammars.**

# Syntax-directed translation

- Associate attributes with each grammar symbol that describes its properties.
- **An attribute has a name and an associated value.**
- With each production in a grammar, give semantic rules or actions.
- The general approach to **syntax-directed translation is to construct a parse tree or syntax tree and compute the values of attributes** at the nodes of the tree by visiting them in some order.

# Syntax Directed Translation

Syntax directed translation

Grammar + Semantic rules = SDT

SDT for evaluation of expression

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

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

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

$/F \quad \{ T.value = F.value \}$

$F \rightarrow num \quad \{ F.val = num.lvalue \}$

# Syntax-directed translation

There are two ways to represent the semantic rules associated with grammar symbols.

- **Syntax-Directed Definitions (SDD)**
- **Syntax-Directed Translation Schemes (SDT)**

# Syntax-Directed Translation

1. We associate information with the programming language constructs by attaching attributes to grammar symbols.

2. Values of these attributes are evaluated by **the semantic rules** associated with the production rules.

3. Evaluation of these semantic rules:
   - ❖ may generate intermediate codes
   - ❖ may put information into the symbol table
   - ❖ may perform type checking
   - ❖ may issue error messages
   - ❖ may perform some other activities
   - ❖ in fact, they may perform almost any activities.

4. An attribute may hold almost any thing.
   - ❖ **a string, a number, a memory location, a complex record.**

# Syntax-Directed Translation

- Conceptually with both the syntax directed translation and translation scheme we
  - Parse the input token stream
  - Build the parse tree
  - Traverse the tree to evaluate the semantic rules at the parse tree nodes.

**Input string** → **parse tree** → **dependency graph** → **evaluation order**

**semantic rules for**

Conceptual view of syntax directed translation

# Syntax-Directed Definition

1. A syntax-directed definition is a generalization of a context-free grammar in which:
   - Each grammar symbol is associated with a set of attributes.
   - This set of attributes for a grammar symbol is partitioned into two subsets called
     - **synthesized and**
     - **inherited attributes of that grammar symbol.**
   - Each production rule is associated with a set of semantic rules.

2. The value of an attribute at a parse tree node is defined by the semantic rule associated with a production at that node.
3. **The value of a synthesized attribute at a node is computed from the values of attributes at the children in that node of the parse tree**
4. **The value of an inherited attribute at a node is computed from the values of attributes at the siblings and parent of that node of the parse tree**

# Syntax-Directed Definition

**Examples:**

**Synthesized attribute : E→E1+E2 { E.val =E1.val + E2.val}**

**Inherited attribute :A→XYZ {Y.val = 2 * A.val}**

1. ***Semantic rules*** set up dependencies between attributes which can be represented by a *dependency graph*.

2. This ***dependency*** *graph* determines the evaluation order of these semantic rules.

3. Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

# Syntax-Directed Definitions

- A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules.
- Attributes are associated with grammar symbols and rules are associated with productions.

| PRODUCTION | SEMANTIC RULE |
|---|---|
| $E \rightarrow E1 + T$ | E.code = E1.code \|\| T.code \|\| '+' |

# Syntax-Directed Definitions

- SDDs are highly readable and give high-level specifications for translations.
- But they hide many implementation details.
- **For example, they do not specify order of evaluation of semantic actions.**
- Syntax-Directed Translation Schemes (SDT) embeds program **fragments called semantic actions within production bodies**
- SDTs are more efficient than SDDs as they **indicate the order of evaluation of semantic actions associated with a production rule.**

# Inherited Attributes

**An INHERITED ATTRIBUTE for a non-terminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N.**

The production must have B as a symbol in its body.

An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

# Annotated Parse Tree

1. **A parse tree showing the values of attributes at each node is called an annotated parse tree.**
2. Values of Attributes in nodes of annotated parse-tree are either,
   - initialized to constant values or by the lexical analyzer.
   - determined by the semantic-rules.
3. The process of computing the attributes values at the nodes is called annotating (or decorating) of the parse tree.
4. Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

# Syntax-Directed Definition

In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

$$b=f(c_1,c_2,\ldots,c_n)$$

where *f* is a function and *b* can be one of the followings:

➔ ***b*** **is a synthesized attribute of A and** $c_1,c_2,\ldots,c_n$ **are attributes of the grammar symbols in the production (** $A \rightarrow \alpha$ **).**

**OR**

➔ ***b*** **is an inherited attribute one of the grammar symbols in** $\alpha$ **(on the right side of the production), and** $c_1,c_2,\ldots,c_n$ **are attributes of the grammar symbols in the production (** $A \rightarrow \alpha$ **).**

# Attribute Grammar

- So, a semantic rule $b=f(c_1,c_2,\ldots,c_n)$ indicates that the attribute b *depends on* attributes $c_1,c_2,\ldots,c_n$.

- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values.

- An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

# Syntax-Directed Definition -- Exan

| Production | Semantic Rules |
|---|---|
| $L \rightarrow E\ n$ | print(E.val) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | E.val = T.val |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | T.val = F.val |
| $F \rightarrow ( E )$ | F.val = E.val |
| $F \rightarrow digit$ | F.val = digit.lexval |

1. Symbols E, T, and F are associated with a synthesized attribute *val*.
2. The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).
3. Terminals are assumed to have synthesized attributes only. Values for attributes of terminals are usually supplied by the lexical analyzer.
4. The start symbol does not have any inherited attribute unless otherwise stated.

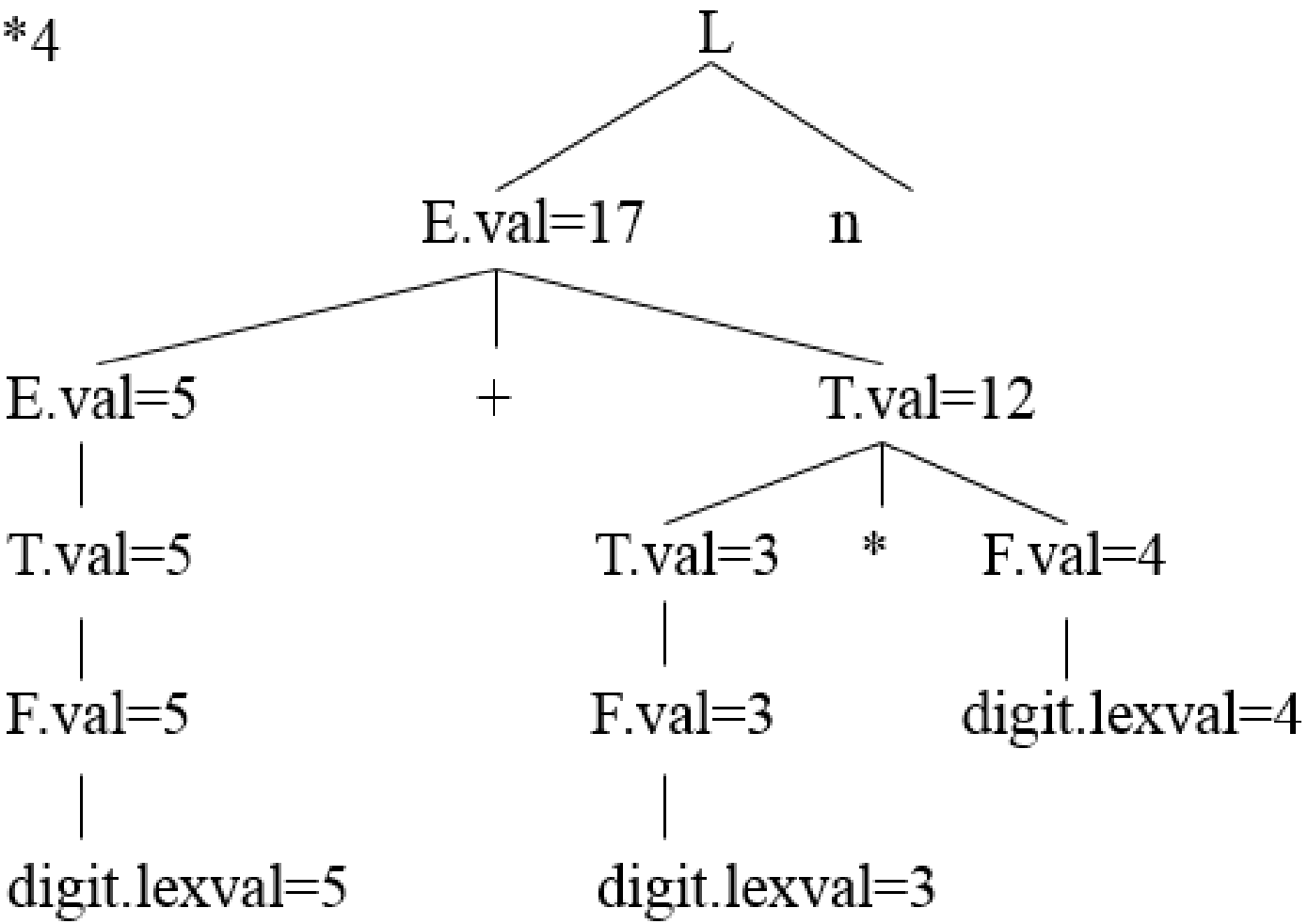
# S-attributed definition

- A syntax directed translation that uses synthesized attributes exclusively is said to be a **S-attributed definition.**

- **A parse tree** for a S-attributed definition can be annotated by evaluating the semantic rules for the attributes at each node, bottom up from leaves to the root.

# Annotated Parse Tree -- Example

Input: 5+3*4



# Inherited attributes

- **An inherited value at a node in a parse tree is defined in terms of attributes at the parent and/or siblings of the node.**

- Convenient way for expressing the dependency of a programming language construct on the context in which it appears.

- We can use inherited attributes to keep track of whether an identifier appears on the left or right side of an assignment to decide whether the address or value of the assignment is needed.

- **Example: The inherited attribute distributes type information to the various identifiers in a declaration**.

# Syntax-Directed Definition – Inherited Attributes

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

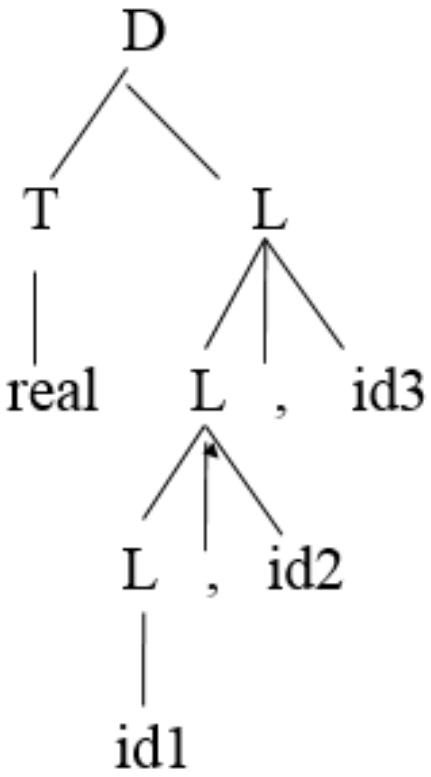
1. Symbol T is associated with a synthesized attribute *type*.

2. Symbol L is associated with an inherited attribute *in*.

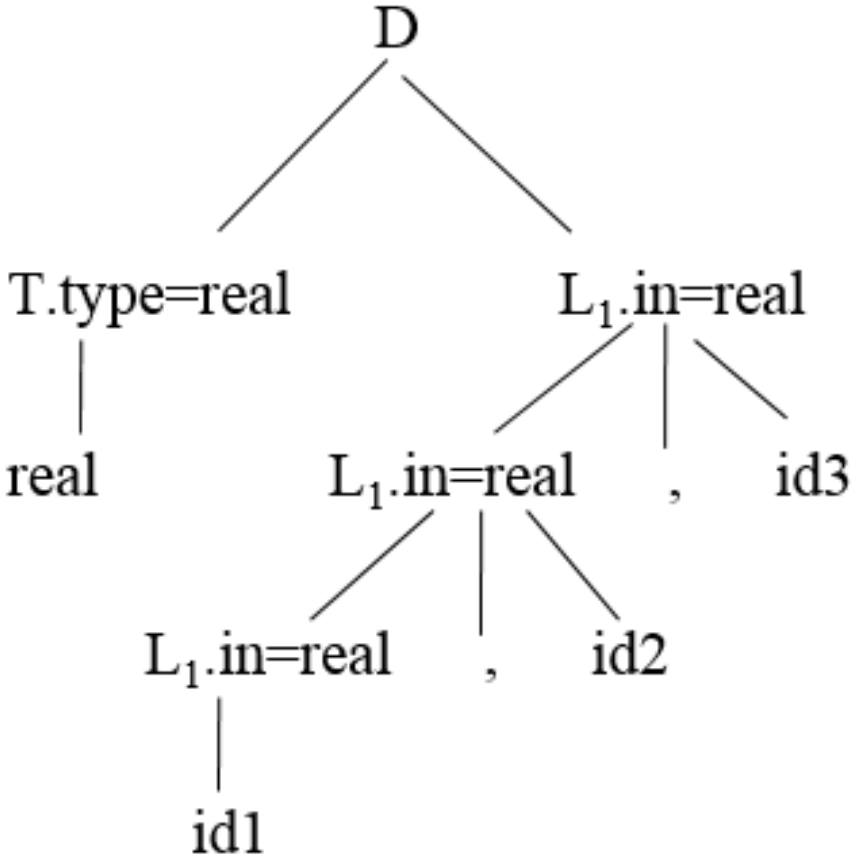
# Annotated parse tree

Input: `real p,q,r`

*parse tree*




*annotated parse tree*



# Dependency Graph

## Directed Graph

- Shows interdependencies between attributes.
- If an attribute b at a node depends on an attribute c, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c.

## Construction:

- Put each semantic rule into the form $b=f(c_1,\ldots,c_k)$ by introducing dummy synthesized attribute b for every semantic rule that consists of a procedure call.

E.g.,

**L → E n** ***print*(E.*val*)**

**Becomes:** ***dummy* = *print*(E.*val*)**

- The graph has a node for each attribute and an edge to the node for b from the node for c if attribute b depends on attribute c.

# Dependency Graph Construction

for each node *n* in the parse tree do

for each attribute a of the grammar symbol at *n* do

construct a node in the dependency graph for a

for each node *n* in the parse tree do

for each semantic rule $b = f(c_1, \ldots, c_n)$

associated with the production used at *n* do

for i= 1 to n do

construct an edge from

the node for $c_i$ to the node for *b*

# Dependency Graph Construction

## Example

| Production | Semantic Rule |
| --- | --- |
| E→E1 + E2 | E.val = E1.val + E2.val |

- E.val is synthesized from E1.val and E2.val
- The dotted lines represent the parse tree that is not part of the dependency graph.

# Dependency Graph

| | |
|---|---|
| D → T L | L.in = T.type |
| T → **int** | T.type = integer |
| T → **real** | T.type = real |
| L → $L_1$ **id** | $L_1$.in = L.in, addtype(**id**.entry,L.in) |
| L → **id** | addtype(**id**.entry,L.in) |

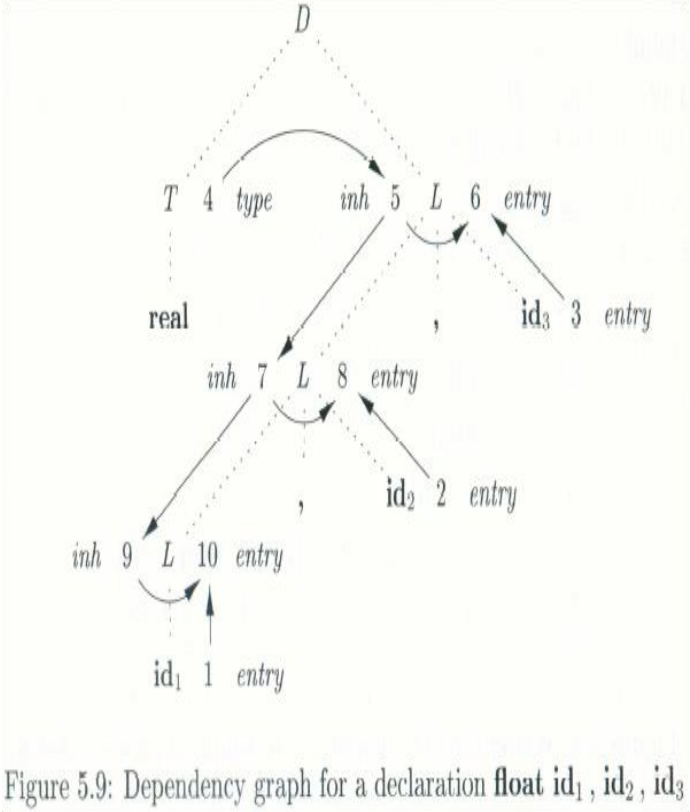


Figure 5.9: Dependency graph for a declaration **float** $id_1$ , $id_2$ , $id_3$

# Evaluation Order

- A topological sort of a directed acyclic graph is any ordering m1,m2…mk of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes.
- . i.e if there is an edge from $m_i$ to $m_j$ them $m_i$ appears before $m_j$ in the ordering
- Any topological sort of dependency graph gives a valid order for evaluation of semantic rules associated with the nodes of the parse tree
- The dependent attributes c1,c2….ck in b=f(c1,c2….ck ) must be available before f is evaluated.
- Translation specified by Syntax Directed Definition
- **Input string ⟶ parse tree ⟶ dependency graph ⟶ evaluation order for semantic rules**

# Topological Sort

- which we can evaluate the attributes at various nodes of a parse tree.
- If there is an edge from node M to N, then attribute corresponding to M first be evaluated before evaluating N.
- **Thus the allowable orders of evaluation are N1, N2, ..., Nk such that if there is an edge from Ni to Nj then i < j.**
- Such an ordering embeds a directed graph into a linear order, and is called a ***topological sort*** *of the graph.*
- If there is any cycle in the graph, then there are no topological sorts.

# Evaluation Order

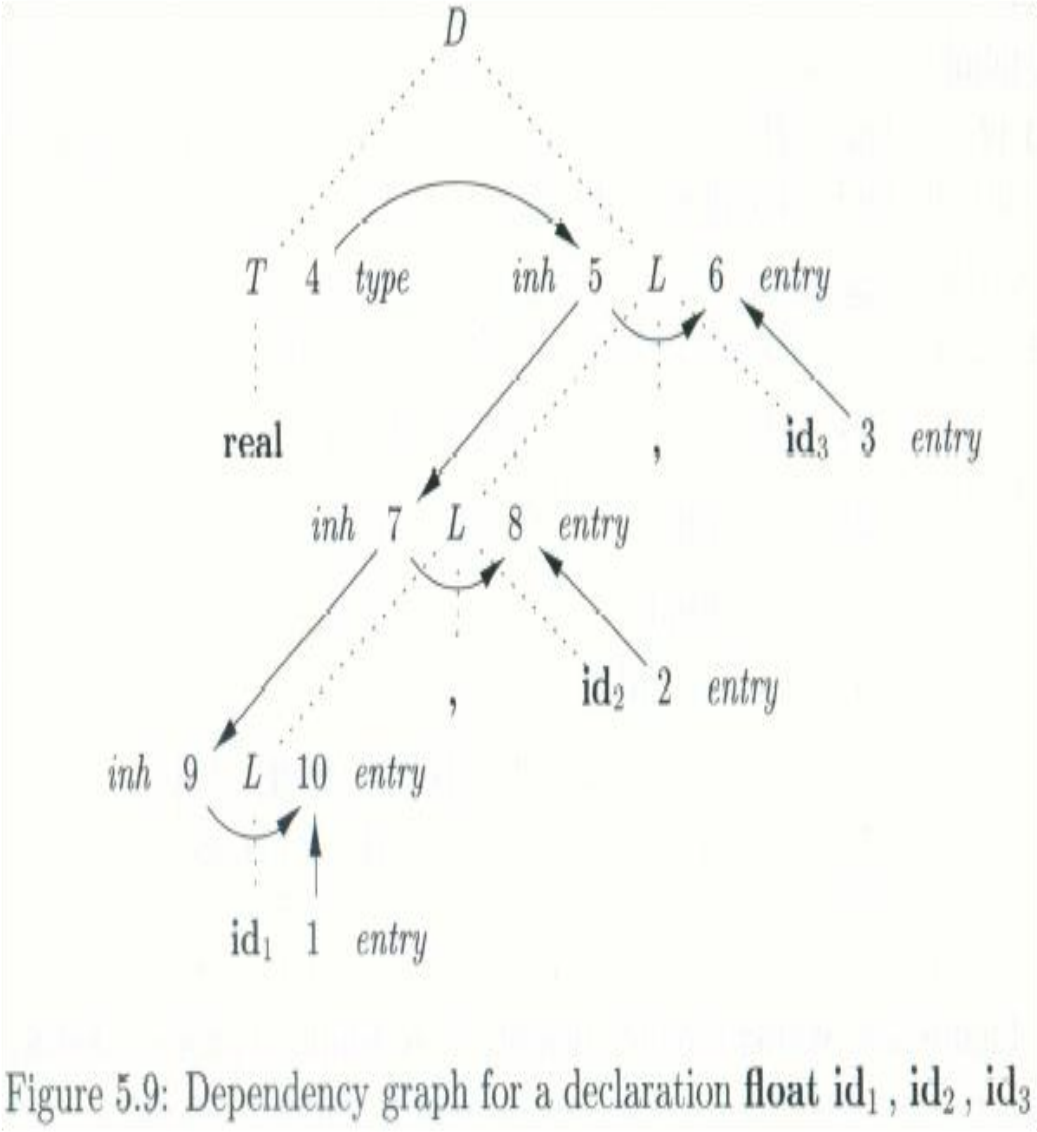



Figure 5.9: Dependency graph for a declaration **float** $id_1$ , $id_2$ , $id_3$

**a4=real;**
**a5=a4;**
**addtype(id3.entry,a5);**
**a7=a5;**
**addtype(id2.entry,a7);**
**a9=a7;**
**addtype(id1.entry,a5);**

# Evaluating Semantic Rules

- **Parse Tree methods**
  - At compile time evaluation order obtained from the **topological sort of dependency graph.**
  - Fails if dependency graph has a cycle
- **Rule Based Methods**
  - Semantic rules analyzed by hand or specialized tools at compiler construction time
  - Order of evaluation of attributes associated with a **production is pre-determined at compiler construction time**
- **Oblivious Methods**
  - Evaluation order is chosen without considering the semantic rules.
  - Restricts the class of syntax directed definitions that can be implemented.
  - If translation takes place during parsing order of evaluation is forced by parsing method.

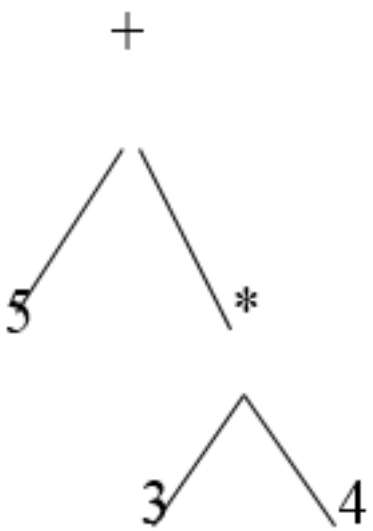
# Syntax Trees

- an intermediate representation of the compiler's input.
- A condensed form of the parse tree.
- **Syntax tree shows the syntactic structure of the program while omitting irrelevant details.**
- Operators and keywords are associated with the interior nodes.
- Chains of simple productions are collapsed.

**Syntax directed translation can be based on syntax tree as well as parse tree.**

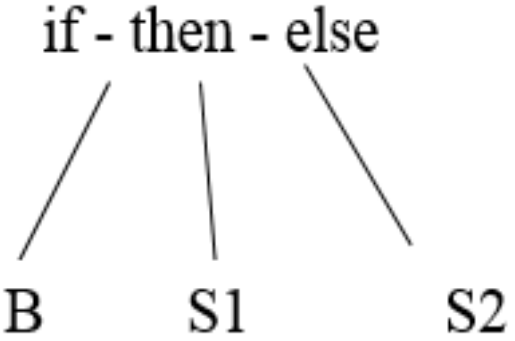
# Syntax Tree-Examples

Expression:




- Leaves: identifiers or constants
- Internal nodes: labelled with operations
- Children: of a node are its operands

if B then S1 else S2




Statement:

- Node's label indicates what kind of a statement it is
- Children of a node correspond to the components of the statement

# Constructing Syntax Tree for Expressions

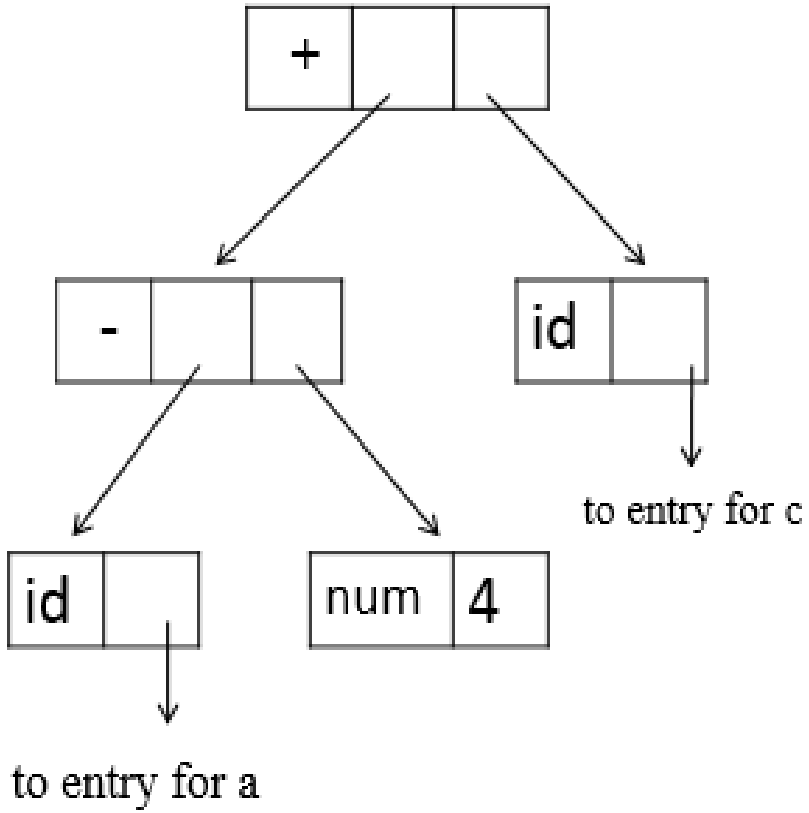
- Each node can be implemented as a record with several fields.
- **Operator node:** one field identifies the operator (**called *label of the node***) and remaining fields contain pointers to operands.
- The nodes may also contain fields to hold the values (**pointers to values**) of attributes attached to the nodes.
- Functions used to create nodes of syntax tree for expressions with binary operator are given below.
  - **mknode(op,left,right)**
  - **mkleaf(id,entry)**
  - **mkleaf(num,val)**

  **Each function returns a pointer to a newly created node.**

# Constructing Syntax Tree for Expressions

Example: a-4+c

1. p1:=mkleaf(id,entrya);
2. p2:=mkleaf(num,4);
3. p3:=mknode(-,p1,p2)
4. p4:=mkleaf(id,entryc);
5. p5:= mknode(+,p3,p4);




- **The tree is constructed bottom up.**

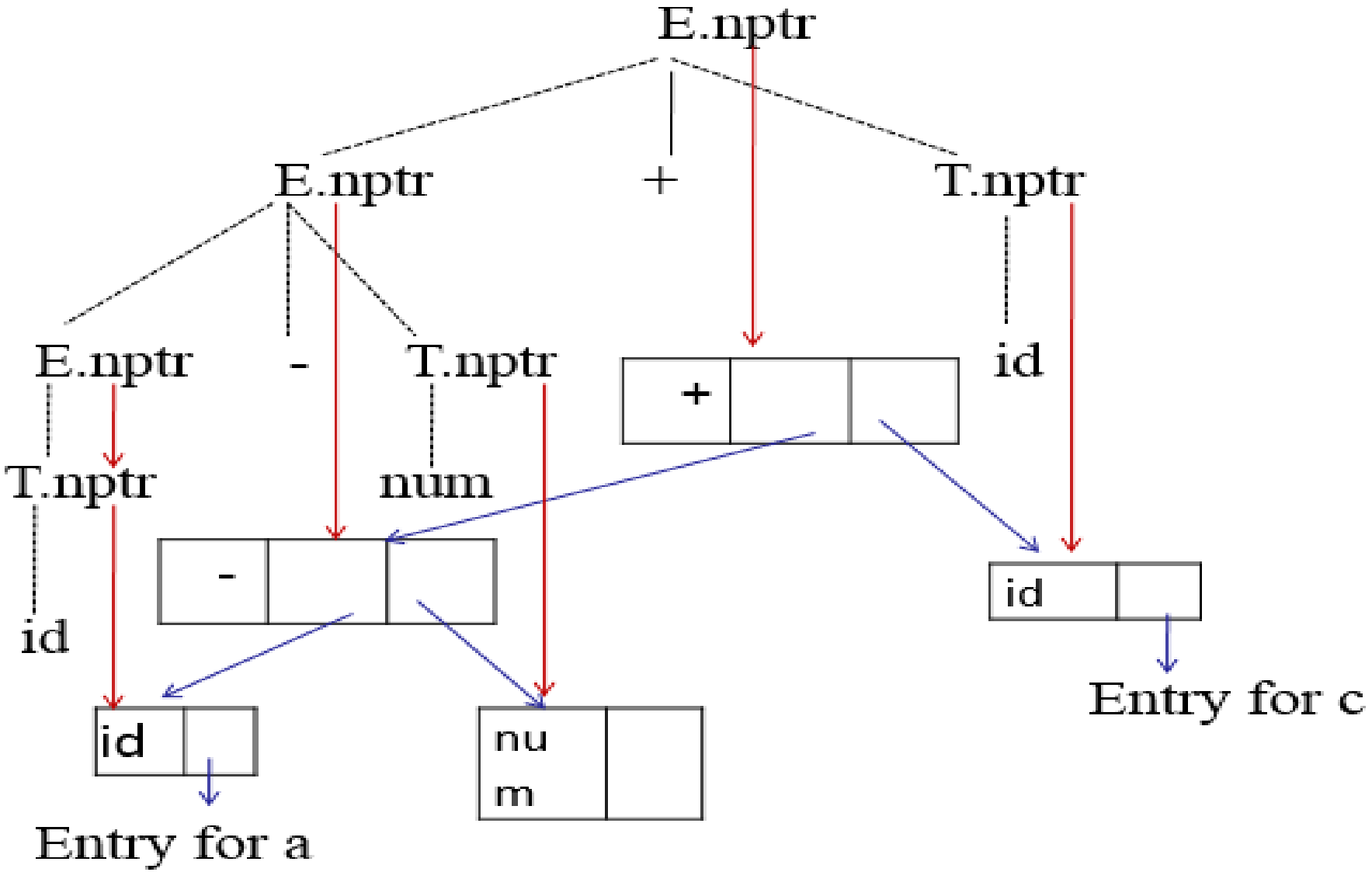
# A syntax Directed Definition for Constructing Syntax Tree

1. It uses underlying productions of the grammar to schedule the calls of the functions ***mkleaf*** and ***mknode*** to construct the syntax tree
2. Employment of the synthesized attribute ***nptr*** (**pointer**) for **E** and **T** to keep track of the pointers returned by the function calls.

**PRODUCTION SEMANTIC RULE**

| | |
|---|---|
| $E \rightarrow E_1 + T$ | E.*nptr* = *mknode*("+",$E_1$.*nptr* ,T.*nptr*) |
| $E \rightarrow E_1 - T$ | E.*nptr* = *mknode*("-",$E_1$.*nptr* ,T.*nptr*) |
| $E \rightarrow T$ | E.*nptr* = T.*nptr* |
| $T \rightarrow (E)$ | T.*nptr* = E.*nptr* |
| $T \rightarrow id$ | T.*nptr* = *mkleaf*(id, id.*lexval*) |
| $T \rightarrow num$ | T.*nptr* = *mkleaf*(num, num.*val*) |

# Annotated parse tree depicting construction syntax tree for the expression a-4+c



# S-Attributed Definitions

1. Syntax-directed definitions are used to specify syntax-directed translations.
2. To create a translator for an arbitrary syntax-directed definition can be difficult.
    1. We would like to evaluate the semantic rules during parsing (i.e. in a single pass, we will parse and we will also evaluate semantic rules during the parsing).
    2. We will look at two sub-classes of the syntax-directed definitions:
        - ❖ **S-Attributed Definitions: only synthesized attributes used in the syntax-directed definitions.**
        - ❖ All actions occur on the right hand side of the production.

# S-Attributed Definitions

- ❖L-Attributed Definitions: in addition to synthesized attributes, we may also use inherited attributes in a restricted fashion.

3. To implement **S-Attributed Definitions and L-Attributed Definitions** we can evaluate semantic rules in a single pass during the parsing.

4. Implementations of **S-attributed Definitions** are a little bit easier than implementations of **L-Attributed Definitions**

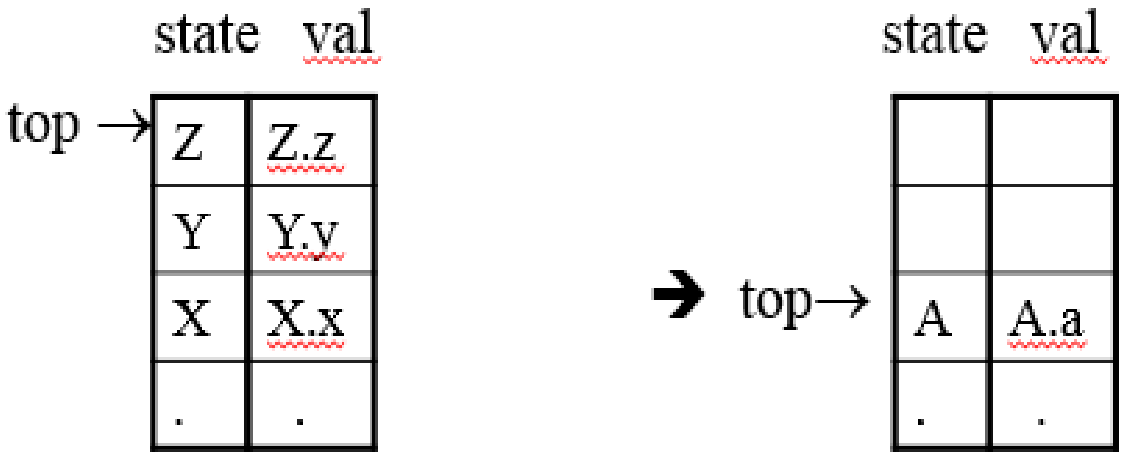
# Bottom-Up Evaluation of S-Attributed Definitions

- A translator for an **S-attributed definition** can often be implemented with the help of an LR parser.
- From an S-attributed definition the parser generator can construct a translator that evaluates attributes as it parses the input.
- We put the **values of the synthesized attributes of the grammar symbols a stack that has extra fields to hold the values of attributes.**
  - The stack is implemented by a pair of arrays *val & state*
  - **If the $i^{th}$ state symbol is A the val[i] will hold the value of the attribute associated with the parse tree node corresponding to this A.**

# Bottom-Up Evaluation of S-Attributed Definitions

- We evaluate the values of the attributes during reductions.

$A \rightarrow XYZ$ A.a=f(X.x,Y.y,Z.z) where all attributes are synthesized.




- Synthesized attributes are evaluated before each reduction.
- Before XYZ is reduced to A, the value of Z.z is in val[top], that of Y.y in val[top-1] and that of X.x in val[top-2].
- After reduction top is decremented by 2.
- If a symbol has no attribute the corresponding entry in the array is undefined.

# Bottom-Up Evaluation of S-Attributed Definitions

| Production | Semantic Rules |
|---|---|
| $L \rightarrow E\ n$ | print(val[top-1]) |
| $E \rightarrow E_1 + T$ | val[ntop] = val[top-2] + val[top] |
| $E \rightarrow T$ | |
| $T \rightarrow T_1 * F$ | val[ntop] = val[top-2] * val[top] |
| $T \rightarrow F$ | |
| $F \rightarrow ( E )$ | val[ntop] = val[top-1] |
| $F \rightarrow digit$ | |

1. At each shift of **digit**, we also push **digit.lexval** into ***val-stack***.
2. At all other shifts, we do not put anything into ***val-stack*** because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).

# Bottom-Up Evaluation -- Example

- **At each shift of digit, we also push digit.lexval into *val-stack*.**

| Input | state | val | semantic rule |
|---|---|---|---|
| 5+3*4n | - | - | |
| +3*4n | 5 | 5 | |
| +3*4n | F | 5 | F → digit |
| +3*4n | T | 5 | T → F |
| +3*4 n | E | 5 | E → T |
| 3*4n | E+ | 5- | |
| *4 n | E+3 | 5-3 | |
| *4n | E+F | 5-3 | F → digit |
| *4n | E+T | 5-3 | T → F |
| 4n | E+T* | 5-3- | |
| n | E+T*4 | 5-3-4 | |
| n | E+T*F | 5-3-4 | F → digit |
| n | E+T | 5-12 | $T \rightarrow T_1 * F$ |
| n | E | 17 | $E \rightarrow E_1 + T$ |
| | En | 17- | L → E n |
| | L | 17 | |

# L-Attributed Definitions

1. When translation takes place during parsing, order of evaluation is linked to the order in which the nodes of a parse tree are created by parsing method.
2. A natural order can be obtained by applying the procedure ***dfvisit*** to the root of a parse tree.
3. We call this evaluation **order *depth first order*.**

# L-Attributed Definitions

4. L-attributed definition is a class of syntax directed definition whose attributes can always be evaluated in depth first order( L stands for left since attribute information flows from left to right).

```
dfvisit(node n)
{
  for each child m of n, from left to right
 {
 evaluate inherited attributes of m
  dfvisit(m)
}
evaluate synthesized attributes of n
}
```

# L-Attributed Definitions

A syntax-directed definition is **L-attributed** if each inherited attribute of $X_j$, **where $1 \leq j \leq n$, on the right side of $A \rightarrow X_1X_2...X_n$ depends only on**

1. The attributes of the symbols $X_1,...,X_{j-1}$ to the left of $X_j$ in the production
2. The inherited attribute of A

**Every S-attributed definition is L-attributed, since the restrictions apply only to the inherited attributes (not to synthesized attributes).**

# A Definition which is *not* L-Attributed

| Productions | Semantic Rules |
|---|---|
| A → L M | L.in=l(A.i) |
| | M.in=m(L.s) |
| | A.s=f(M.s) |
| A → Q R | R.in=r(A.in) |
| | Q.in=q(R.s) |
| | A.s=f(Q.s) |

This syntax-directed definition is not L-attributed because the semantic rule **Q.in=q(R.s)** violates the restrictions of **L-attributed** definitions.

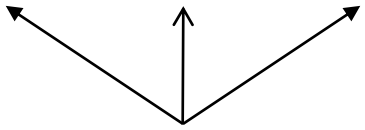
- When **Q**.in must be evaluated before we enter to **Q** because it is an inherited attribute.
- But the value of **Q**.in depends on **R**.s which will be available after we return from **R**. So, we are not be able to evaluate the value of **Q**.in before we enter to **Q**.

# Translation Schemes

- In a syntax-directed definition, we do not say anything about the evaluation times of the semantic rules (when the semantic rules associated with a production should be evaluated).
- Translation schemes describe the order and timing of attribute computation.
- **A translation scheme is a context-free grammar in which:**
  - ❖ ***attributes are associated with the grammar symbols and***
  - ❖ ***semantic actions enclosed between braces {} are inserted within the right sides of productions.***

Each semantic rule can only use the information compute by already executed semantic rules.

- **Ex:** **A → { … } X { … } Y { … }**



**Semantic Actions**

# Translation Schemes for S-attributed Definitions

- Useful notation for specifying translation during parsing.
- Can have both synthesized and inherited attributes.
- If our syntax-directed definition is S-attributed, the construction of the corresponding translation scheme will be simple.
- Each associated semantic rule in a S-attributed syntax-directed definition will be inserted as a semantic action into the end of the right side of the associated production.

| **Production** | **Semantic Rule** | |
|---|---|---|
| **E → E1 + T** | **E.val = E1.val + T.val** | **a production of a syntax directed definition** |
| **⇓** | | |
| **E → E1 + T { E.val = E1.val + T.val }** | | **the production of the corresponding translation scheme** |

# A Translation Scheme Example

- A simple translation scheme that converts infix expressions to the

**corresponding postfix expressions.**

**E → T R**

**R → + T { print(“+”) } R1**

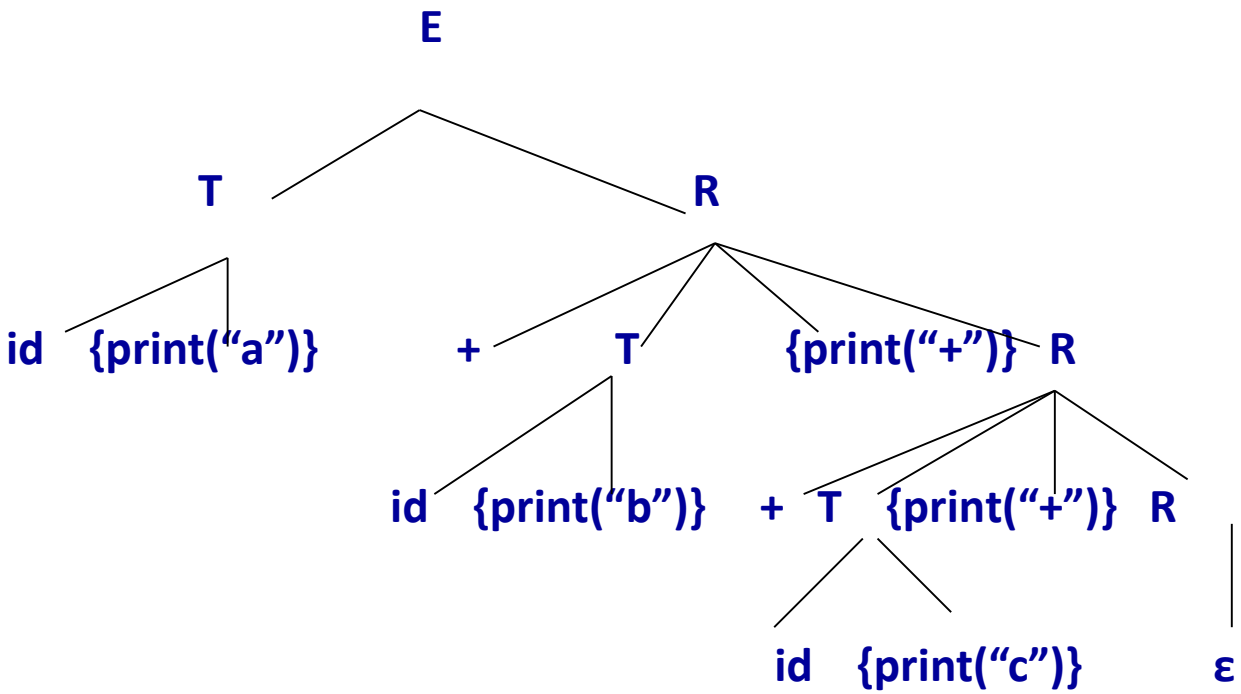
**R → ε**

**T → id { print(id.name) }**

**a+b+c** **ab+c+**

**infix expression** **postfix expression**

# A Translation Scheme Example (cont.)




*The depth first traversal of the parse tree (executing the semantic actions in that order) will produce the postfix representation of the infix expression.*

# Inherited Attributes in Translation Schemes

- If a translation scheme has to contain both synthesized and inherited attributes, we have to observe the following rules to ensure that the attribute value is available when an action refers to it.

  1. **An inherited attribute of a symbol on the right side of a production must be computed in a semantic action before that symbol.**

  2. **A semantic action must not refer to a synthesized attribute of a symbol to the right of that semantic action.**

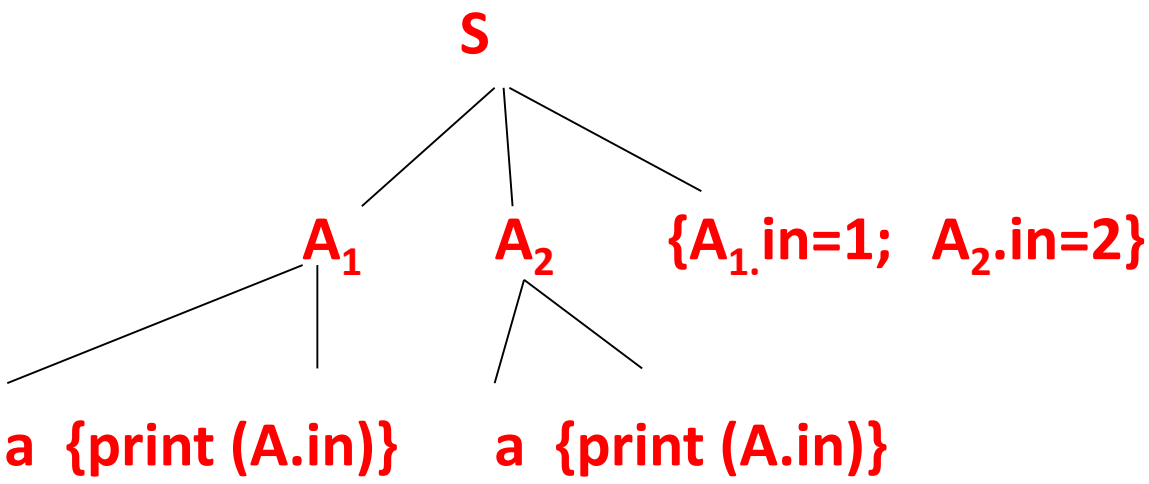
  3. **A synthesized attribute for the non-terminal on the left can only be computed after all attributes it references have been computed (we normally put this semantic action at the end of the right side of the production).**

- With a L-attributed syntax-directed definition, it is always possible to construct a corresponding translation scheme which satisfies these three conditions (This may not be possible for a general syntax-directed translation).

# Inherited Attributes in Translation Schemes: Example

$S \rightarrow A_1 A_2 \quad \{A_1.in=1; \; A_2.in=2\}$

$A \rightarrow a \quad \{ print (A.in)\}$



# A Translation Scheme with Inherited Attributes

D → T {L.in = T.type } L

T → int { T.type = integer }

T → real { T.type = real }

L → {L1.in = L.in } L1, id {addtype(id.entry,L.in)}

L → id {addtype(id.entry,L.in)}

**This is a translation scheme for an L-attributed definitions**

# Bottom Up evaluation of Inherited Attribute

## Removing Embedding Semantic Actions

- In bottom-up evaluation scheme, the semantic actions are evaluated during reductions.
- During the bottom-up evaluation of S-attributed definitions, we have a parallel stack to hold synthesized attributes.

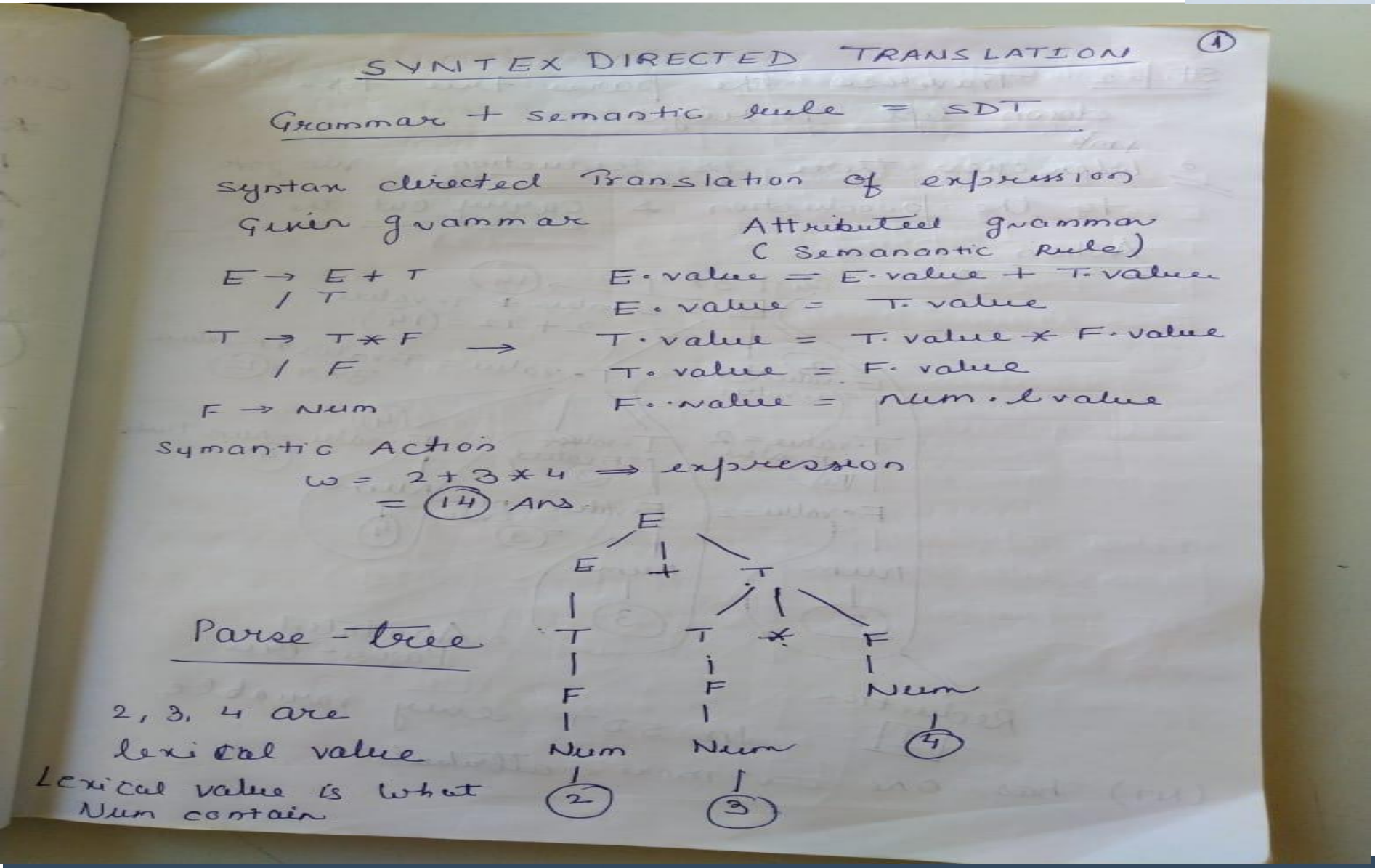
***Problem*: where are we going to hold inherited attributes?**

***A Solution*:**

We will convert our grammar to an equivalent grammar to guarantee to the followings.

1) ***All embedding semantic actions in our translation scheme will be moved into the end of the production rules.***
2) ***All inherited attributes will be copied into the synthesized attributes (most of the time synthesized attributes of new non-terminals).***
3) ***Thus we will be evaluate all semantic actions during reductions, and we find a place to store an inherited attribute.***

# Syntax Directed Translation



SYNTEX DIRECTED TRANSLATION ①

Grammar + semantic rule = SDT

syntax directed Translation of expression

| Given grammar | Attributed grammar (Semanantic Rule) |
|---|---|
| $E \rightarrow E + T$ / $T$ | E.value = E.value + T.value / E.value = T.value |
| $T \rightarrow T * F$ / $F$ | T.value = T.value * F.value / T.value = F.value |
| $F \rightarrow$ Num | F.value = Num.lvalue |

Symantic Action

$w = 2 + 3 * 4 \Rightarrow$ expression

$= (14)$ Ans.

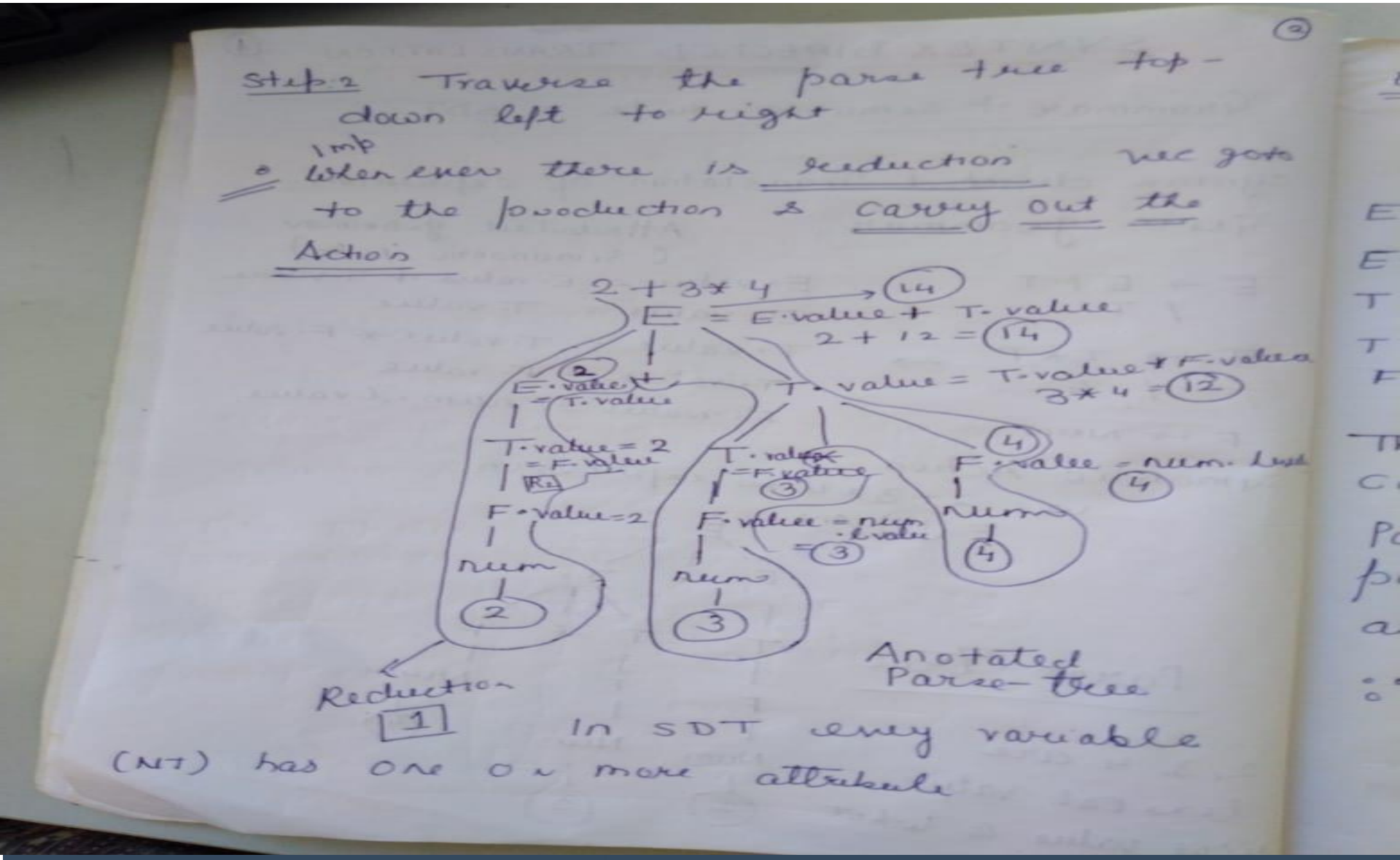
Parse - tree




2, 3, 4 are lexical value

Lexical value is what Num contain

# Syntax Directed Translation for solving Expression Example :1



# Solving -Expression

Syntax directed translation

Grammar + Semantic rules = SDT

SDT for evaluation of expression

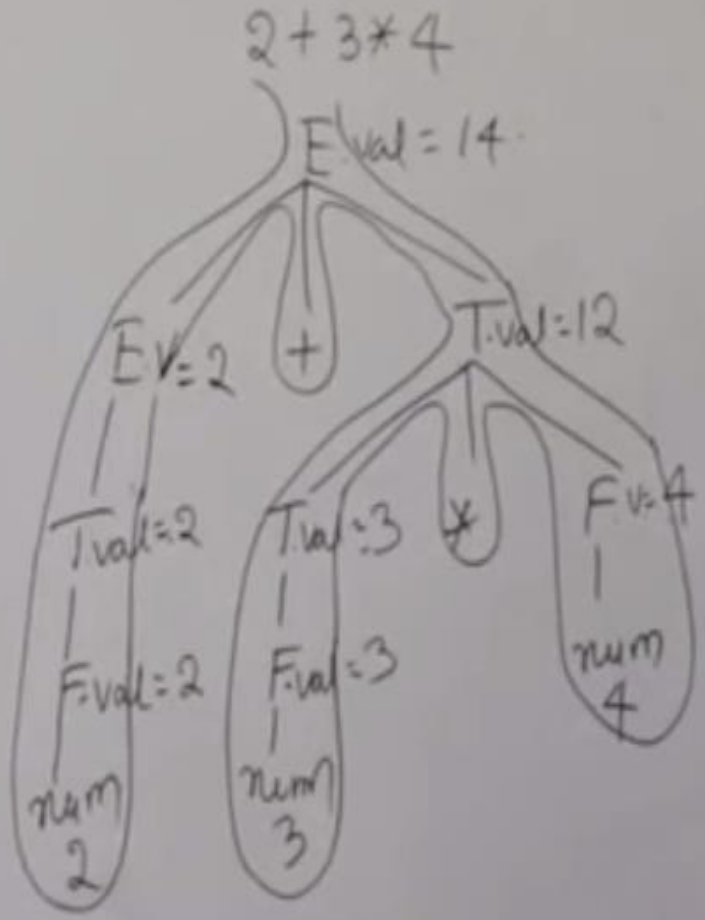
$E \rightarrow E_1 + T \quad \{E.value = E_1.value + T.value\}$

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

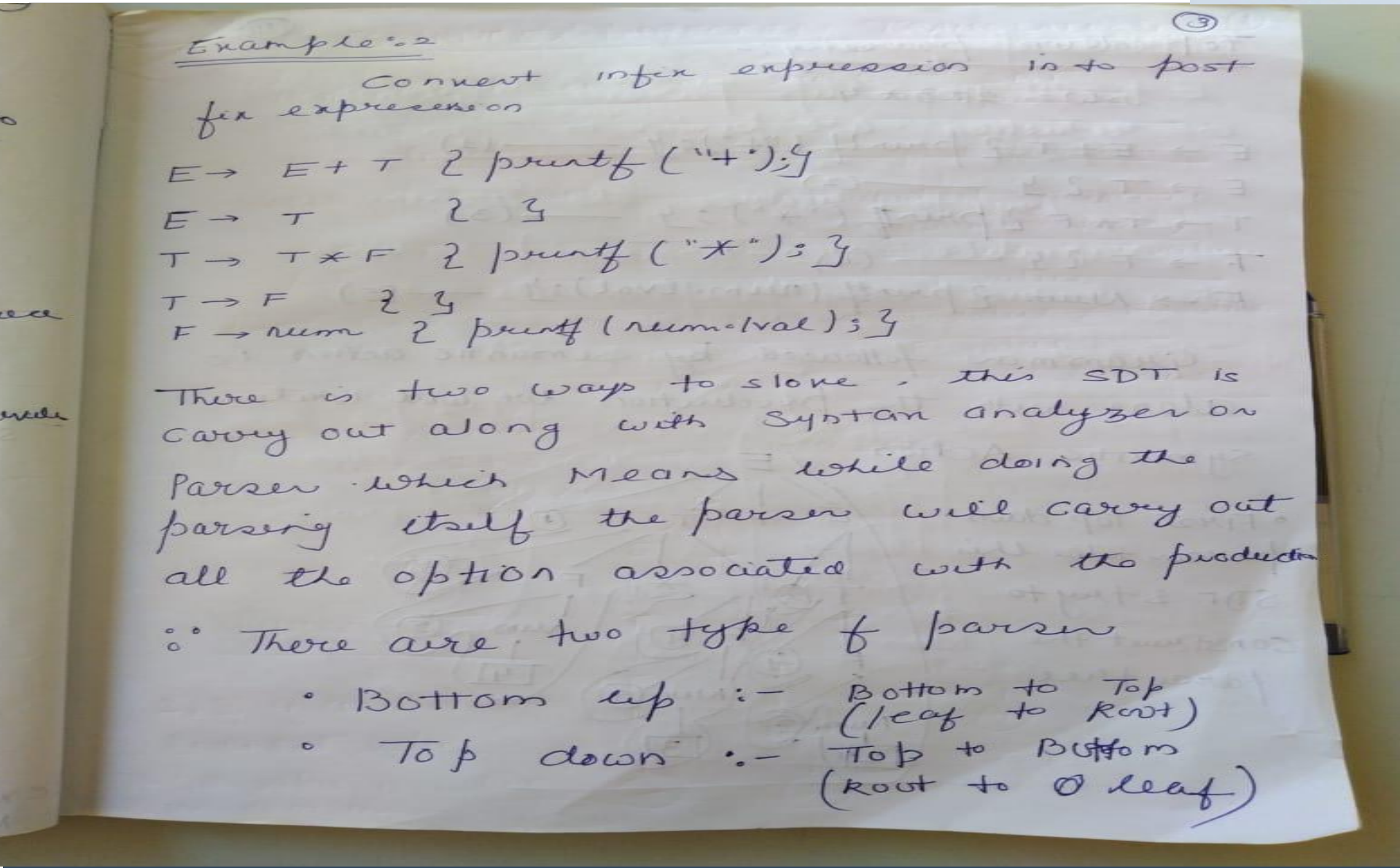
$T \rightarrow T_1 * F \quad \{T.value = T_1.value * F.value\}$

$/F \quad \{T.value = F.value\}$

$F \rightarrow num \quad \{F.val = num.lvalue\}$



# Syntax Directed Translation for converting Infix Expression into Postfix Expression :Example 2



Example :-2

Convert infix expression is to post fix expression

E → E + T { printf("+"); }

E → T { }

T → T * F { printf("*"); }

T → F { }

F → num { printf(num.lval); }

There is two ways to slove, this SDT is carry out along with Syntax analyzer or Parser which Means while doing the parsing itself the parser will carry out all the option associated with the production

∴ There are two type of parser

- Bottom up :- Bottom to Top (leaf to Root)
- Top down :- Top to Bottom (Root to leaf)

# Syntax Directed Translation for converting Infix Expression into Postfix Expression

## Top down parsing :-

$w = 2+3*4$

$E \rightarrow E + T$ { printf("+"); } ——— (1)

$E \rightarrow T$ { } ——— (2)

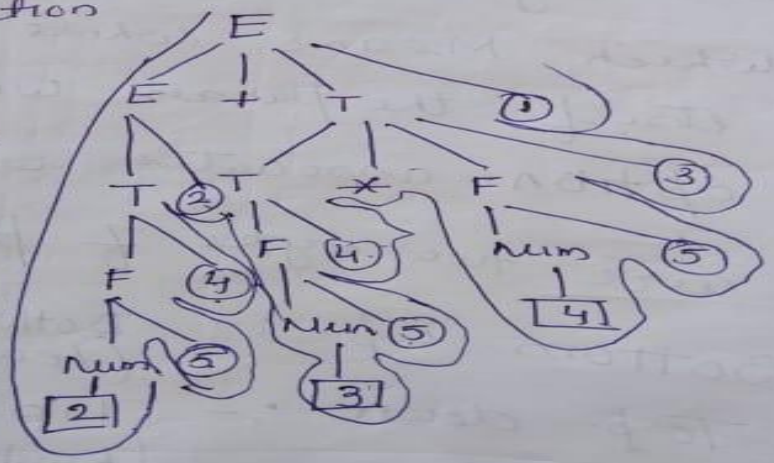
$T \rightarrow T * F$ { printf("*"); } ——— (3)

$T \rightarrow F$ { } ——— (4)

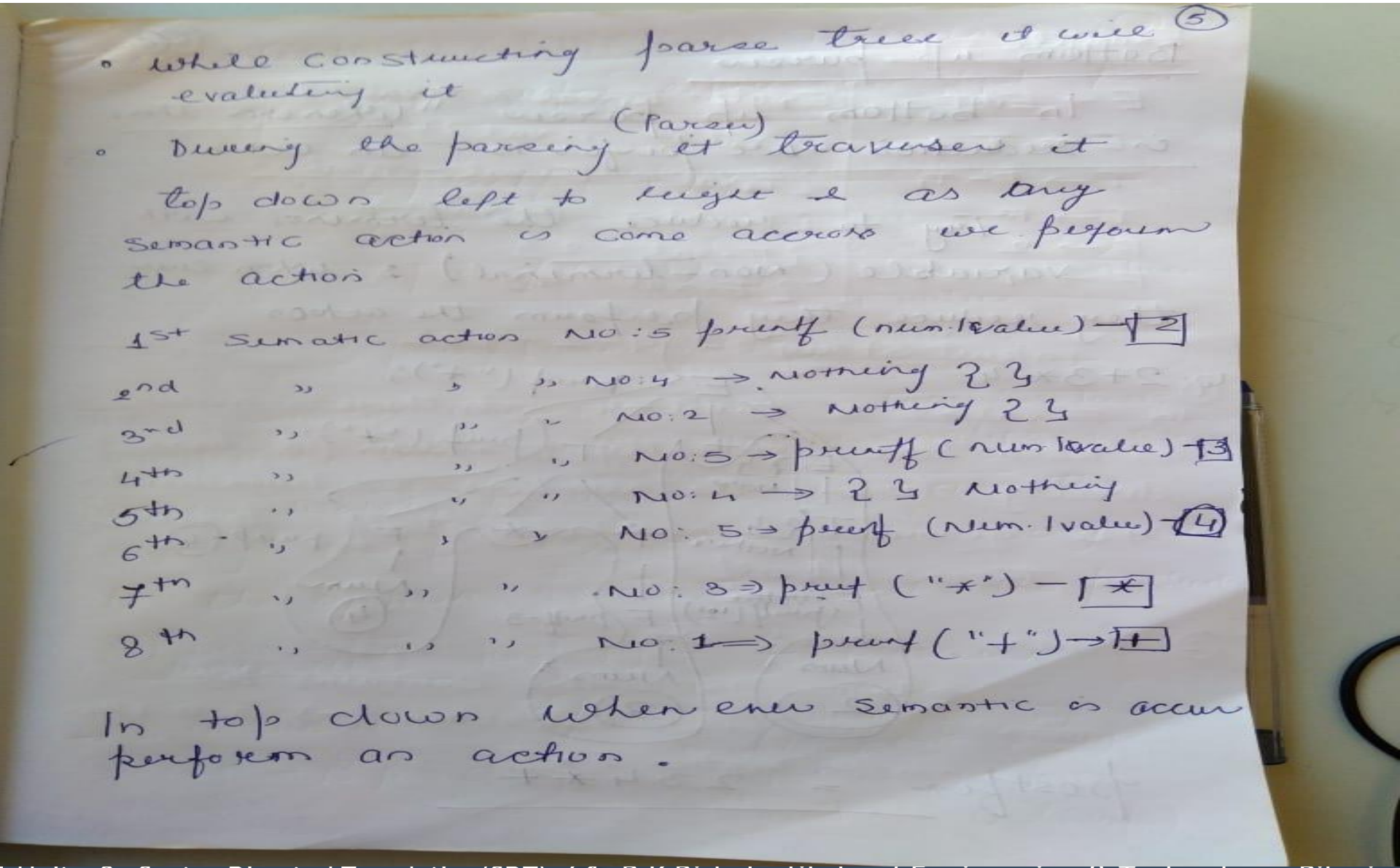
$F \rightarrow Num$ { printf(num.lval); } ——— (5)

Grammar followed by semantic action i.e along with the production we will write Symantic Action

- First Top down parser take this SDT & try to construct the parse tree



# Syntax Directed Translation for converting Infix Expression into Postfix Expression



- While constructing parse tree it will evaluating it (Parse)
- During the parsing it traverses it top down left to right & as any semantic action is come across we perform the action

1st Semantic action NO:5 printf (num.lvalue) → 2

2nd " " " NO:4 → Nothing { }

3rd " " " NO:2 → Nothing { }

4th " " " NO:5 → printf (num.lvalue) → 3

5th " " " NO:4 → { } Nothing

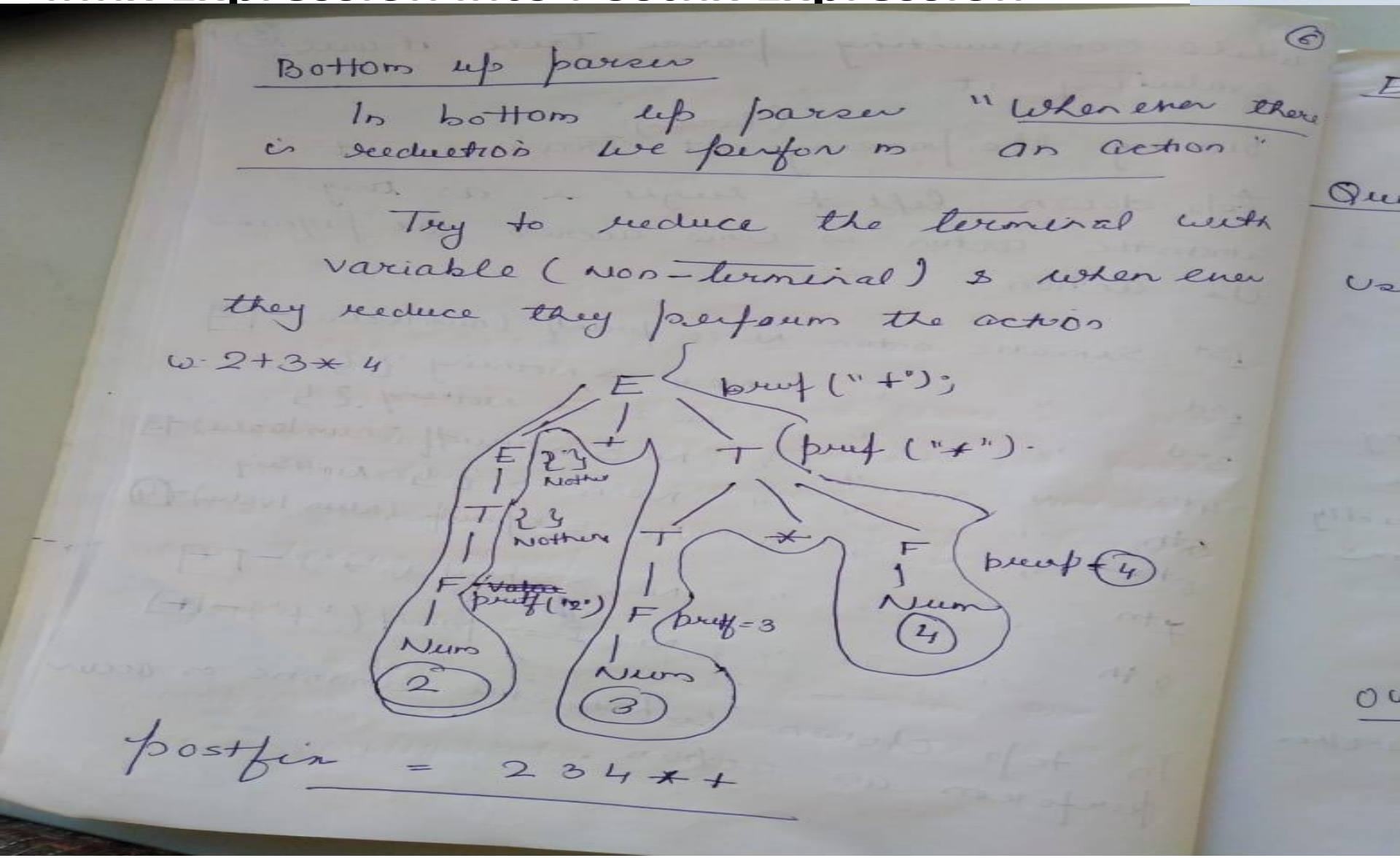
6th " " " NO: 5 → printf (Num.lvalue) → 4

7th " " " NO: 3 → print ("*") → *

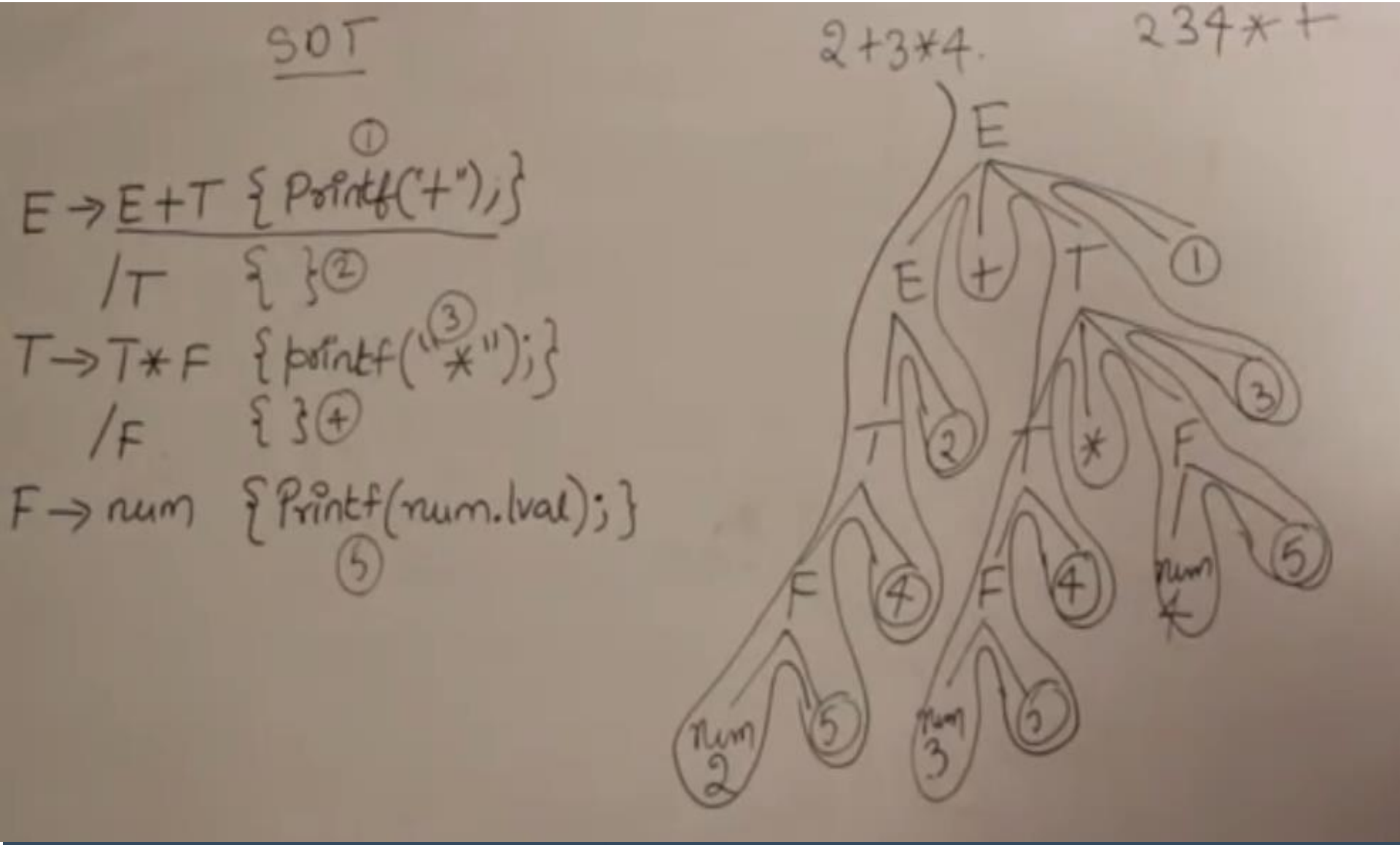
8th " " " NO: 1 → print ("+") → +

In top down when ever Semantic is occur perform an action.

# Syntax Directed Translation for converting Infix Expression into Postfix Expression



# Top Down Parser - Infix Expression into Post fix Expression



# Bottom Up Parser -Infix Expression into Post fix Expression

SDT

$E \rightarrow E+T$ { Printf("+"); } ①

/T { } ②

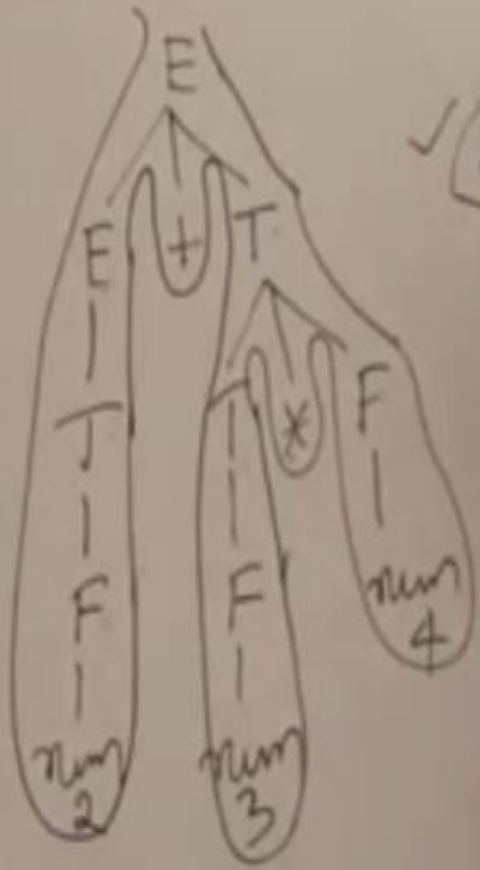
$T \rightarrow T*F$ { printf("*"); } ③

/F { } ④

$F \rightarrow num$ { Printf(num.lval); } ⑤

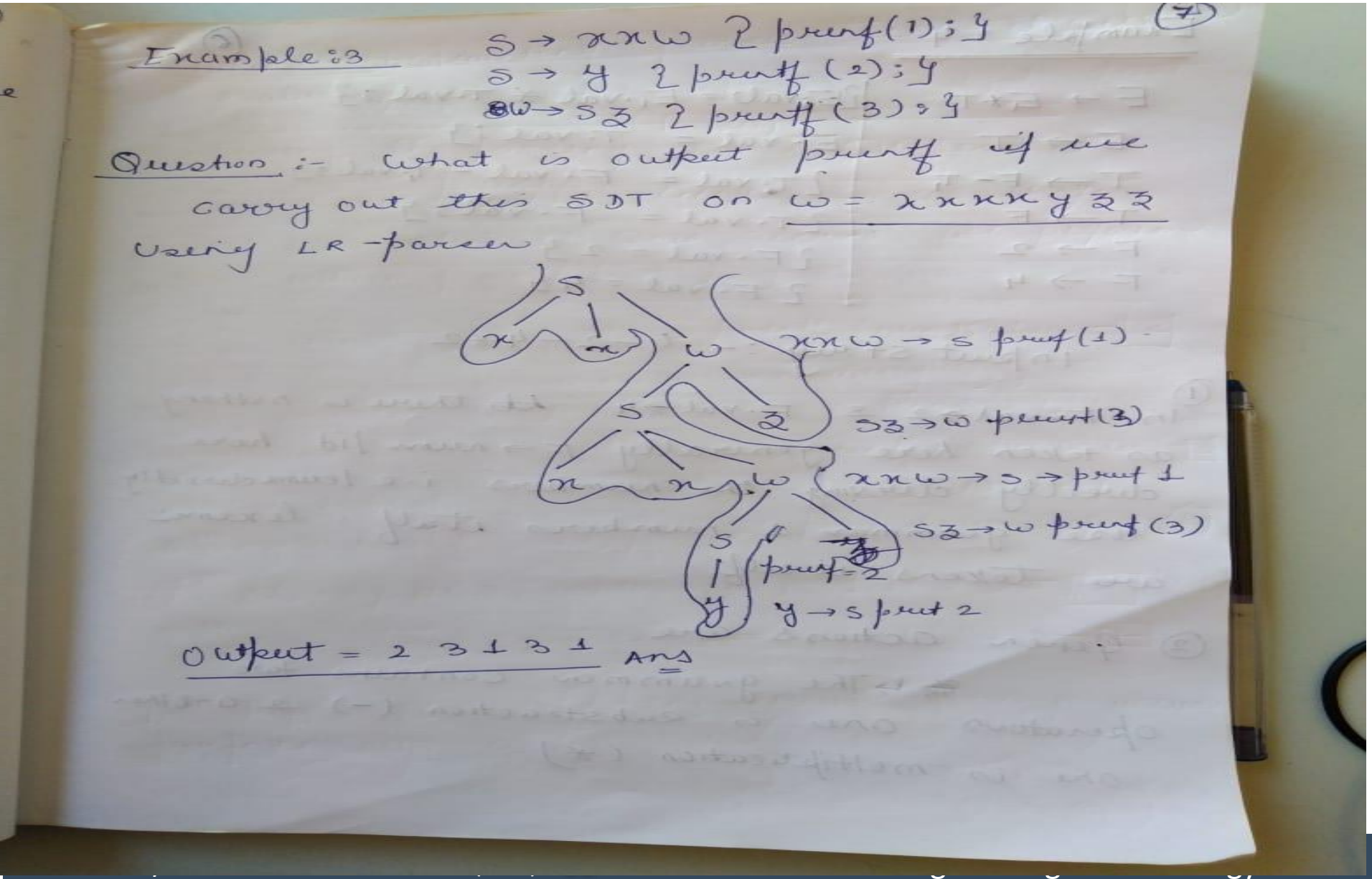
2+3*4.

234*+




✓ 234*+

# Syntax Directed Translation -Example :3



# Syntax Directed Translation -Example :4

Example : 4

| | |
|---|---|
| $E \rightarrow E_1 * T$ | { E.val = $E_1$.val * T.val ; } |
| $E \rightarrow T$ | { E.val = T.val ; } |
| $T \rightarrow F - T_1$ | { T.val = F.val - $T_1$.val ; } |
| $T \rightarrow F$ | { T.val = F.val ; } |
| $F \rightarrow 2$ | { F.val = 2 } |
| $F \rightarrow 4$ | { F.val = 4 } |

Input string :- 4 - 2 - 4 * 2

① In F.val → 2 & F.val → 4 there is nothing as token here generally F → num / id here directly dealing the numbers i.e lexeme directly use grammar & numbers itself lexems are tokens itself

② Given actions are

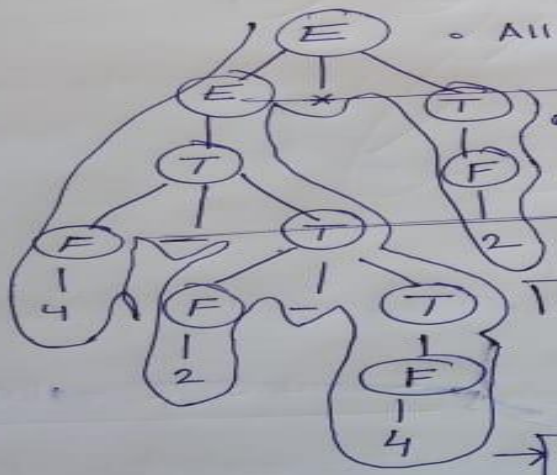
↳ The grammar contains two operators one is substraction (-) & another one is multiplication (*)

# Syntax Directed Translation -Example :4

↳ * is defined as left recursive therefore * is left associative

↳ − is defined as right recursive therefore − is right associative

↳ More over * is defined at higher level to the − in the grammar which means − has higher precedance to *




- All Non leaf are Reduction here
- Total No of reductions = 10

[6 * 2 = 12]

[4 − (−2)] ⑥

[2 − (4 ...)]

→ [2 − 4 (−2)]

w = [4 ?]

[[4 − (2 − 4)] * 2] = (12) Ans

# SDT to build Syntax Tree

STD to build Syntax tree

$E \rightarrow E_1 + T$ { E.nptr = mknode (E1.nptr, '+', T.nptr); }

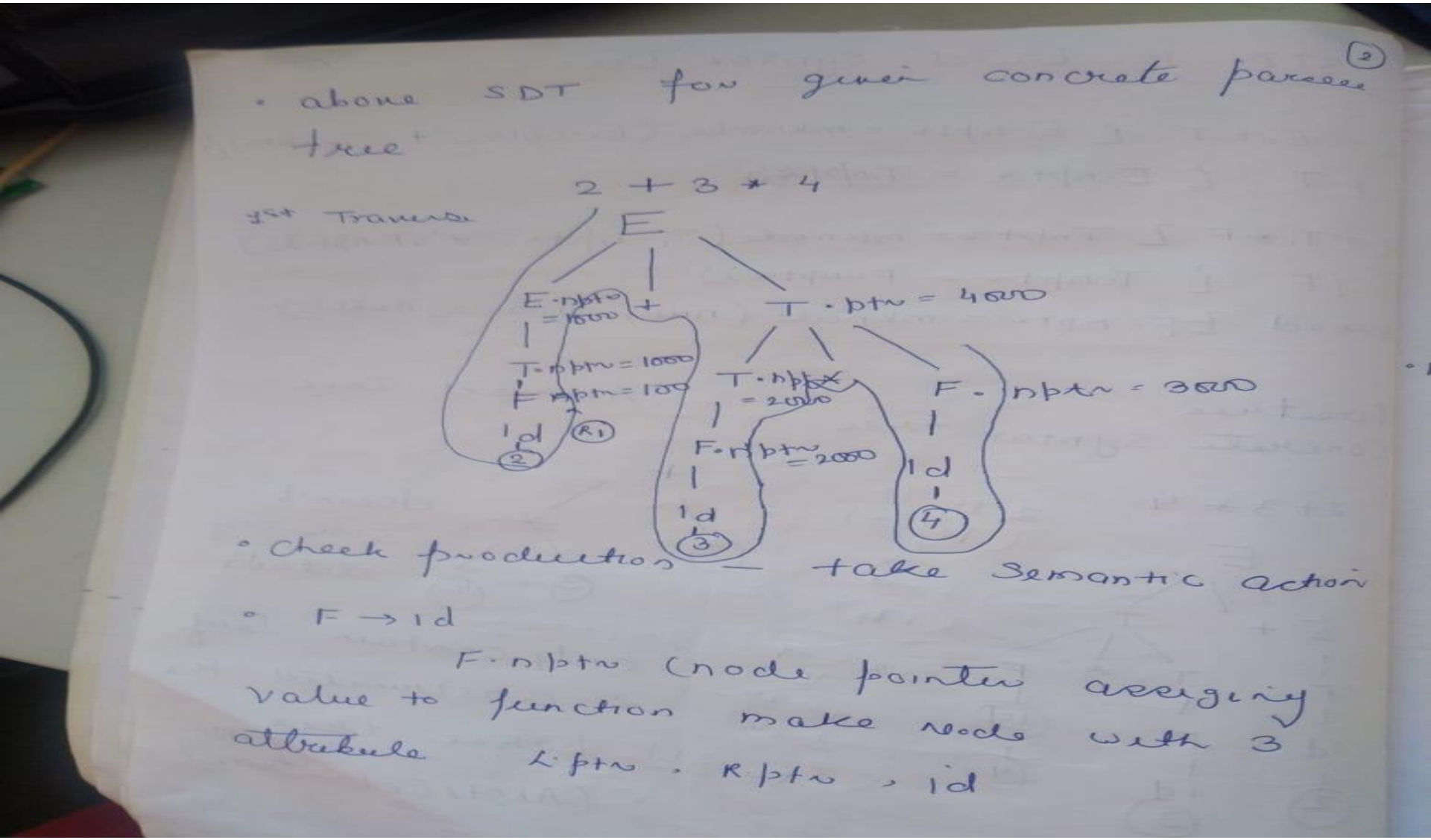
| T { E.nptr = T.nptr; }

$T \rightarrow T_1 * F$ { T.nptr = mknode (T1.nptr, '*', F.nptr); }

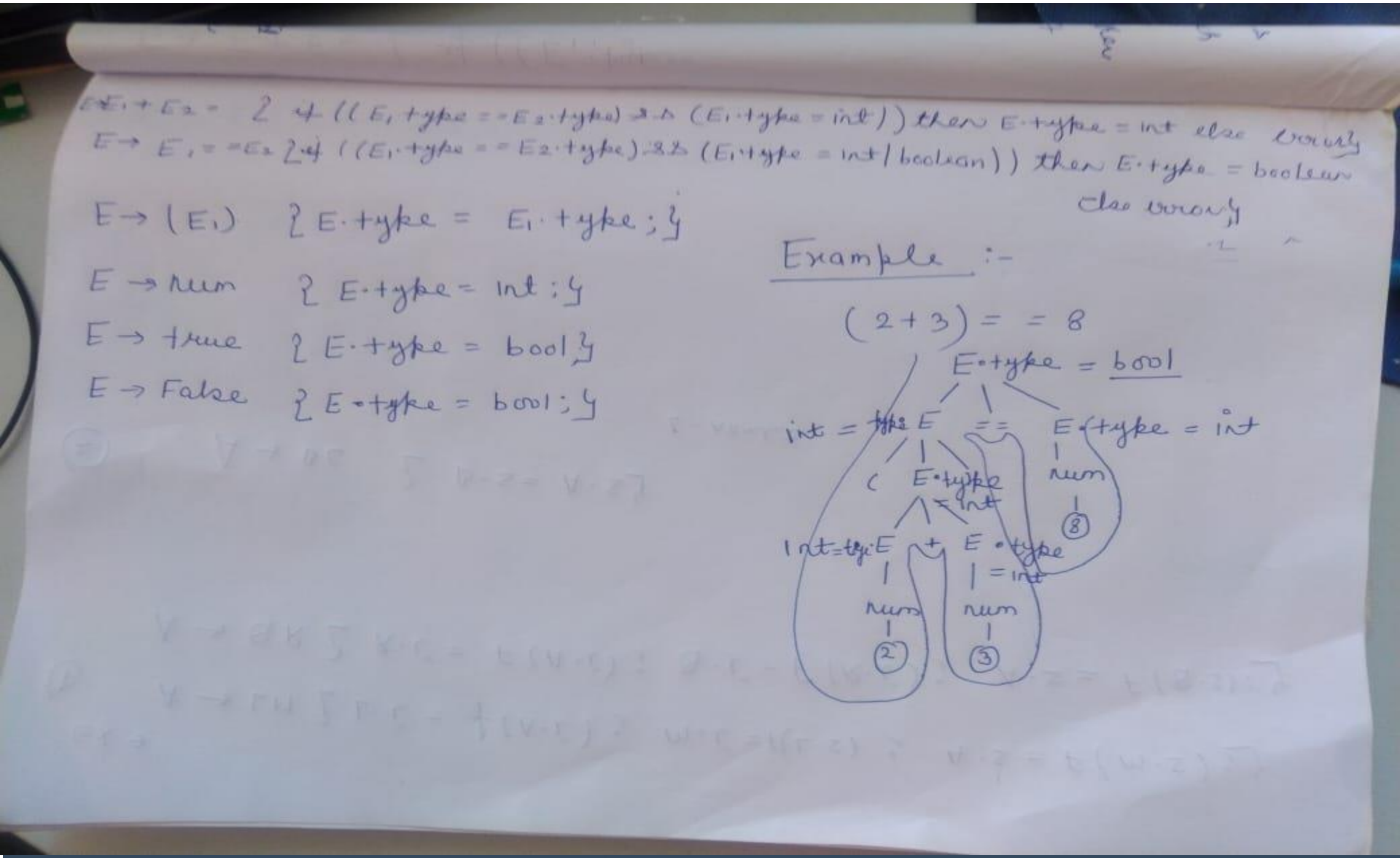
| F { T.nptr = F.nptr; }

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

# SDT to build Syntax Tree



# STD for Type Checking



# Boolean Expressions

- compute logical values
- change the flow of control
- boolean operators are: and or not

$$
\begin{aligned}
E &\rightarrow E \text{ or } E \\
&\mid E \text{ and } E \\
&\mid \text{not } E \\
&\mid (E) \\
&\mid \text{id relop id} \\
&\mid \text{true} \\
&\mid \text{false}
\end{aligned}
$$

# Methods of translation

- ***Evaluate similar to arithmetic expressions***
  - **Normally use 1 for true and 0 for false**

- **Implement by flow of control**
  - **given expression $E_1$ or $E_2$**

    **if $E_1$ evaluates to true**

    **then $E_1$ or $E_2$ evaluates to true**

    **without evaluating $E_2$**

# Numerical representation

- a or b and not c

$t_1$ = not c
$t_2$ = b and $t_1$
$t_3$ = a or $t_2$

- relational expression a < b is equivalent to if a < b then 1 else 0

1. if a < b goto 4.
2. t = 0
3. goto 5
4. t = 1
5.

# Syntax directed translation of boolean expressions

$E \rightarrow E_1$ or $E_2$

E.place := newtmp
emit(E.place ':=' $E_1$.place 'or' $E_2$.place)

$E \rightarrow E_1$ and $E_2$

E.place:= newtmp
emit(E.place ':=' $E_1$.place 'and' $E_2$.place)

$E \rightarrow$ not $E_1$

E.place := newtmp
emit(E.place ':=' 'not' $E_1$.place)

$E \rightarrow (E_1)$ E.place = $E_1$.place

# Syntax directed translation of boolean expressions

***E → id1 relop id2***

**E.place := newtmp**
**emit(if id1.place relop id2.place goto nextstat+3)**
**emit(E.place = 0)**
**emit(goto nextstat+2)**
**emit(E.place = 1)**

***E → true***

**E.place := newtmp**
**emit(E.place = '1')**

***E → false***

**E.place := newtmp**
**emit(E.place = '0')**

# Example:

## Code for a < b or c < d and e < f

## TAC

100: if a < b goto 103
101: $t_1 = 0$
102: goto 104
103: $t_1 = 1$
104:
if c < d goto 107
105: $t_2 = 0$
106: goto 108
107: $t_2 = 1$
108:
if e < f goto 111
109: $t_3 = 0$
110: goto 112
111: $t_3 = 1$
112:
$t_4 = t_2$ and $t_3$
113: $t_5 = t_1$ or $t_4$

# Intermediate Code generation

# Topics to be covered

- Different intermediate forms
- Different Representation of Three Address Code

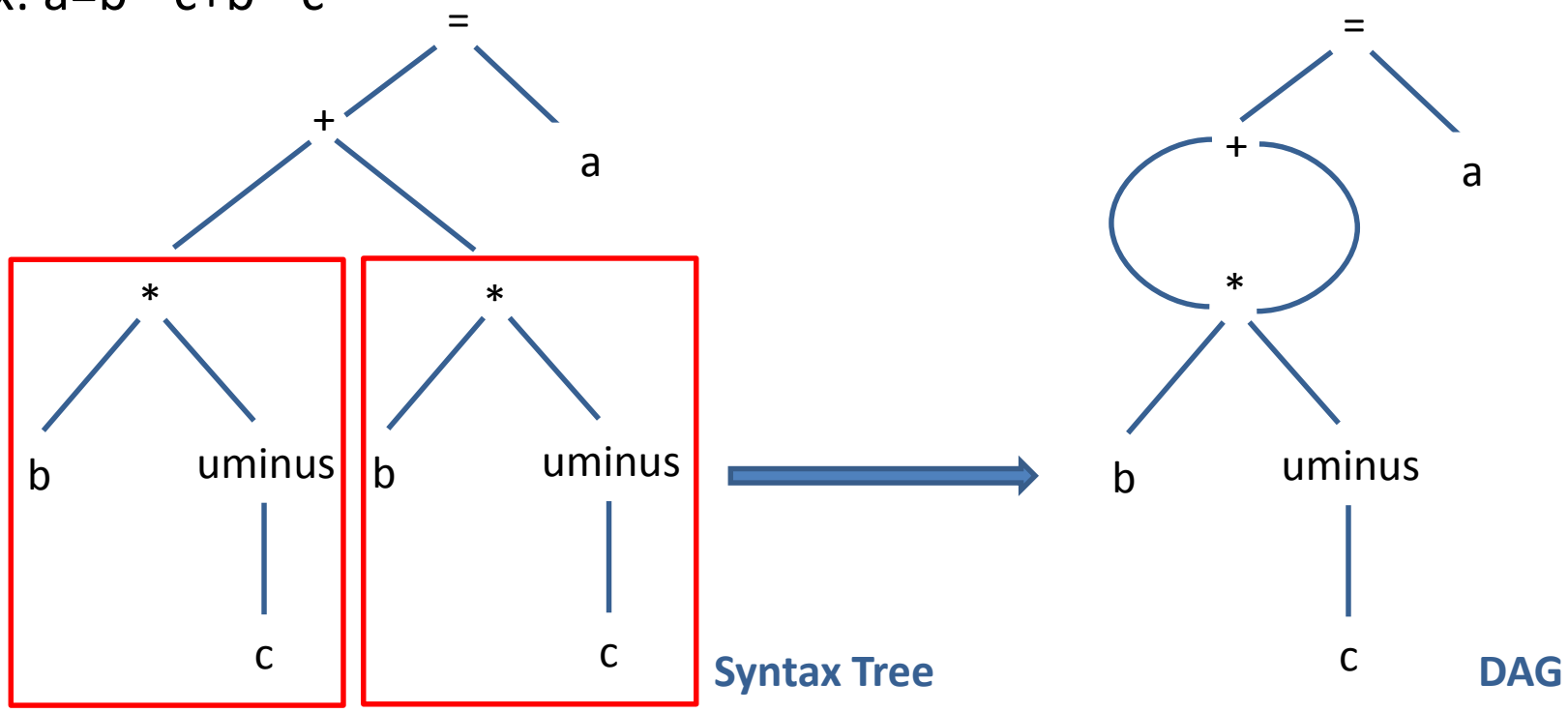
# Different Intermediate Forms

# Types of Intermediate Forms

- types of intermediate forms are:
  1. Abstract syntax tree
  2. Postfix notation
  3. Three address code

# Abstract syntax tree & DAG

- A syntax tree depicts the natural hierarchical structure of a source program.
- A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common sub-expressions are identified.
- Ex: a=b*-c+b*-c



# Postfix Notation

- Postfix notation is a linearization of a syntax tree.
- In postfix notation the operands occurs first and then operators are arranged.
- Ex: **(A + B) * (C + D)**

**Postfix notation: A B + C D + ***

# Three address code

- Three address code is a sequence of statements of the general form,

**a:= b op c**

- Where a, b or c are the operands that can be names or constants and op stands for any operator.
- Example: a = b + c + d

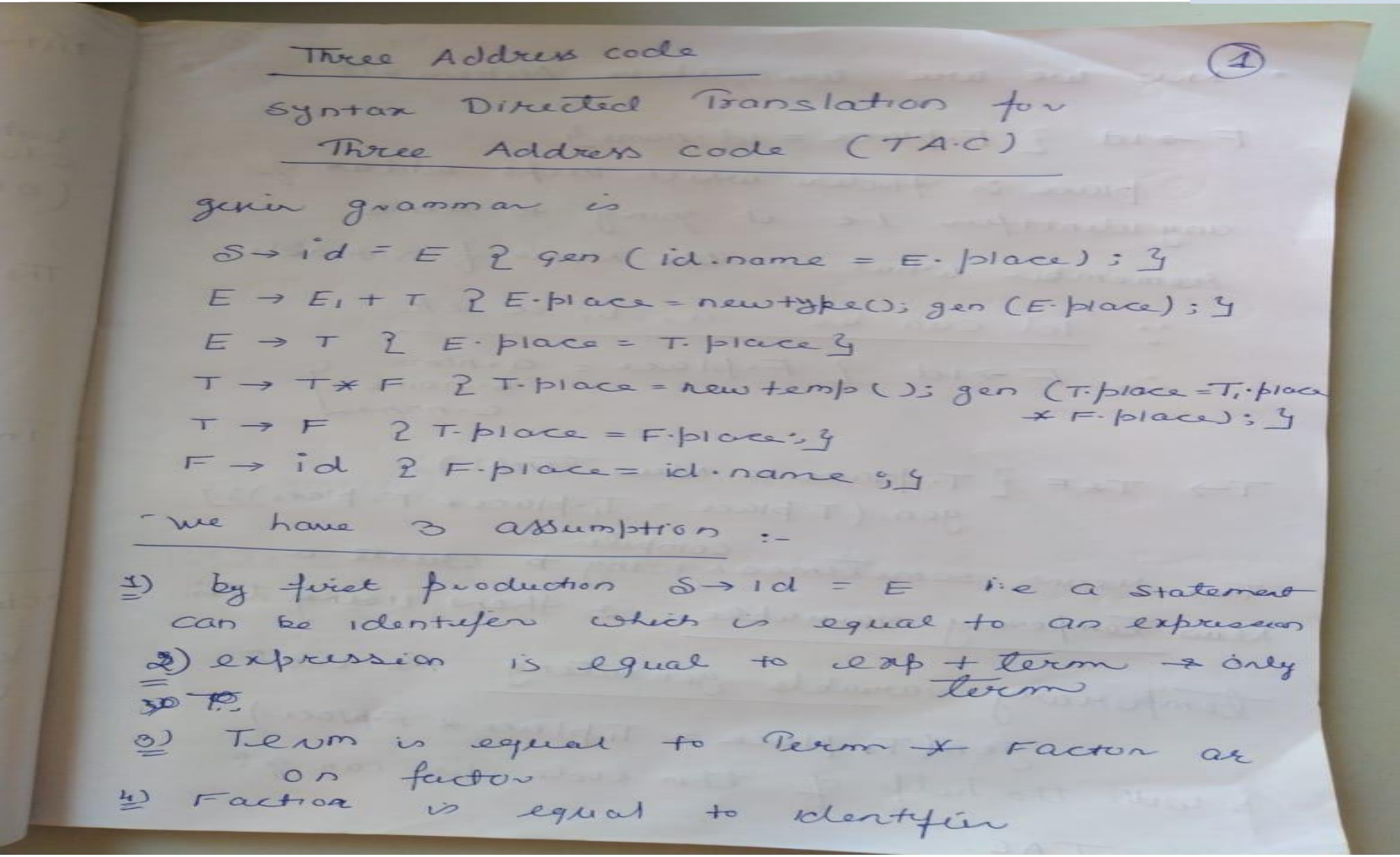
$t_1$=b+c

$t_2$=$t_1$+d

a= $t_2$

- Here $t_1$ and $t_2$ are the temporary names generated by the compiler.
- There are at most three addresses allowed (two for operands and one for result). Hence, this representation is called three-address code.

# Syntax Directed Translation for TAC



Three Address code (1)

Syntax Directed Translation for Three Address code (TAC)

genin grammar is

$S \rightarrow id = E$ { gen (id.name = E.place) ; }

$E \rightarrow E_1 + T$ { E.place = newtype(); gen (E.place) ; }

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

$T \rightarrow T * F$ { T.place = newtemp (); gen (T.place = $T_1$.place * F.place) ; }

$T \rightarrow F$ { T.place = F.place; }

$F \rightarrow id$ { F.place = id.name ; }

we have 3 assumption :-

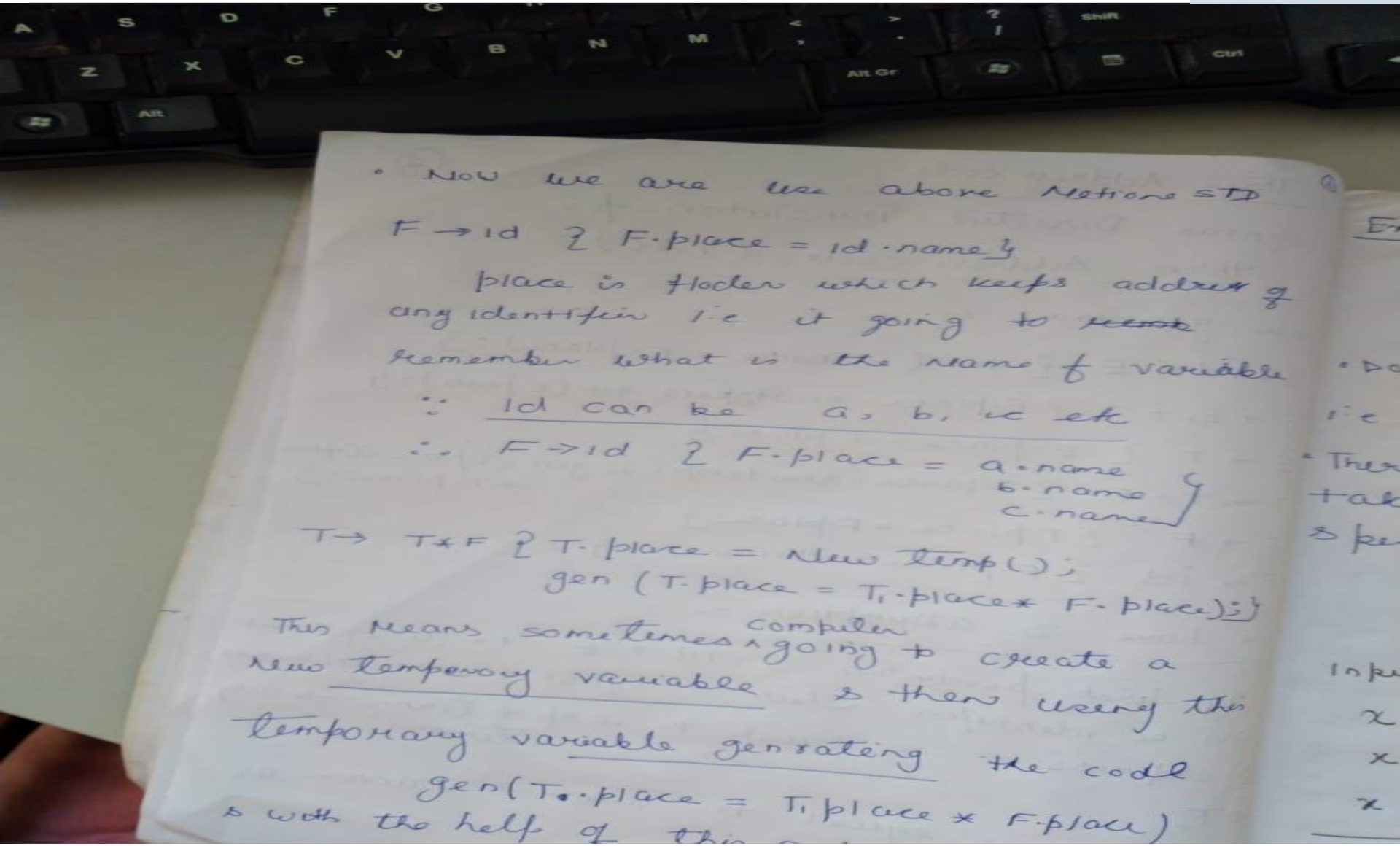
1) by first production $S \rightarrow id = E$ i.e a statement can be identifier which is equal to an expression

2) expression is equal to exp + term ~~+~~ only term

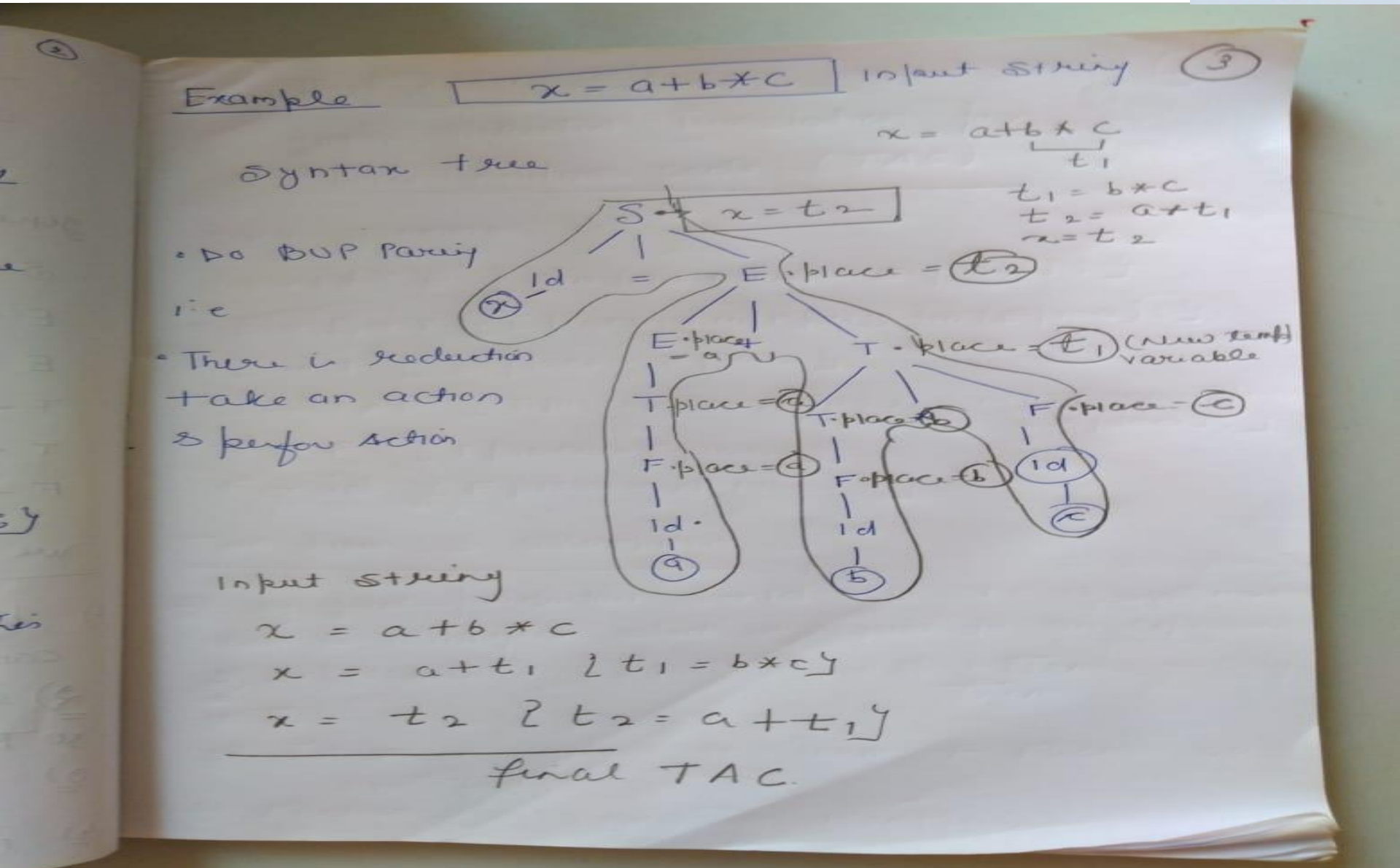
3) Term is equal to Term * Factor or factor

4) Factor is equal to identifier

# Syntax Directed Translation for TAC



# Syntax Directed Translation for TAC



# Common Three Address Instruction For

The common forms of Three Address instructions are

1) **Assignment Statement**
2) **Copy Statement**
3) **Conditional Jump**
4) **Unconditional Jump**
5) **Procedure Call**

# 1. Assignment Statement-

**x = y op z and x = op y**

**X=y*z && x= ++y**

**Here,**

**x, y and z are the operands.**

**op represents the operator.**

It assigns the result obtained after solving the right side expression of the assignment operator to the left side operand.

# 2. Copy Statement

**x = y**

Here,

x and y are the operands.

**= is an assignment operator.**

It copies and assigns the value of operand y to operand x.

# 3. Conditional Jump

**If x relop y goto X**

Here,

x & y are the operands.

X is the tag or label of the target statement.

relop is a relational operator.

If the condition “x relop y” gets satisfied, then-

The control is sent directly to the location specified by label X.

All the statements in between are skipped.

If the condition “x relop y” fails, then-

The control is not sent to the location specified by label X.

The next statement appearing in the usual sequence is executed.

# Unconditional Jump & Procedure Call

## 4. Unconditional Jump

**goto X**

Here, X is the tag or label of the target statement. On executing the statement, The control is sent directly to the location specified by label X.

## 5. Procedure Call

All the statements in between are skipped.

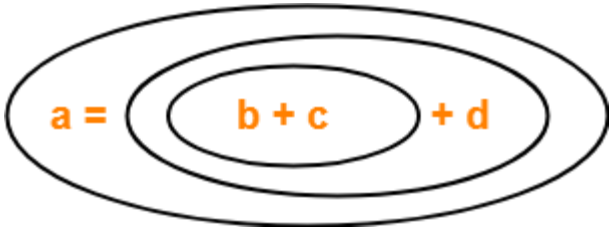
**param x call p return y**

Here, p is a function which takes x as a parameter and returns y.

# Problem-01:

**Write Three Address Code for the following expression-**

**a = b + c + d**




Three Address Code for the given expression is-

(1) T1 = b + c

(2) T2 = T1 + d

(3) a = T2

# Problem-02:

**Write Three Address Code for the following expression-**

**-(a x b) + (c + d) – (a + b + c + d)**

**Solution-**

Three Address Code for the given expression is-

(1) T1 = a x b

(2) T2 = uminus T1

(3) T3 = c + d

(4) T4 = T2 + T3

(5) T5 = a + b

(6) T6 = T3 + T5

(7) T7 = T4 – T6

# Problem-03:

**Write Three Address Code for the following expression-**

**If A < B then 1 else 0**

**Solution-**

Three Address Code for the given expression is-

(1) If (A < B) goto (4)

(2) T1 = 0

(3) goto (5)

(4) T1 = 1

(5) exit

# Problem-04:

**Write Three Address Code for the following expression-**

**If A < B and C < D then t = 1 else t = 0**

**Solution-**

Three Address Code for the given expression is-

(1) If (A < B) goto (3)

(2) goto (4)

(3) If (C < D) goto (6)

(4) t = 0

(5) goto (7)

(6) t = 1

(7) exit

# Different Representation of Three Address Code

# Different Representation of Three Address C

- There are three types of representation used for three address code,
  1. Quadruples
  2. Triples
  3. Indirect triples
- Ex: x= -a*b + -a*b

$t_1 = -a$

$t_2 = t_1 * b$

$t_3 = -a$

$t_4 = t_3 * b$

$t_5 = t_2 + t_4$

$x = t_5$

**Three Address Code**

# Quadruple

- The quadruple is a structure with at the most four fields such as op, arg1, arg2 and result.
- The op field is used to represent the internal code for operator.
- The arg1 and arg2 represent the two operands.
- And result field is used to store the result of an expression.

x= -a*b + -a*b

$t_1$= - a

$t_2$ = $t_1$ * b

$t_3$= - a

$t_4$ = $t_3$ * b

$t_5$ = $t_2$ + $t_4$

x= $t_5$

**Quadruple**

| No. | Operator | Arg1 | Arg2 | Result |
|---|---|---|---|---|
| (0) | | | | |
| (1) | | | | |
| (2) | | | | |
| (3) | | | | |
| (4) | | | | |
| (5) | | | | |

# Triple

- To avoid entering temporary names into the symbol table, we might refer a temporary value by the position of the statement that computes it.
- If we do so, three address statements can be represented by records with only three fields: op, arg1 and arg2.

Quadruple

| No. | Operator | Arg1 | Arg2 | Result |
|---|---|---|---|---|
| (0) | uminus | a | | $t_1$ |
| (1) | * | $t_1$ | b | $t_2$ |
| (2) | uminus | a | | $t_3$ |
| (3) | * | $t_3$ | b | $t_4$ |
| (4) | + | $t_2$ | $t_4$ | $t_5$ |
| (5) | = | $t_5$ | | x |

Triple

| No. | Operator | Arg1 | Arg2 |
|---|---|---|---|
| (0) | | | |
| (1) | | | |
| (2) | | | |
| (3) | | | |
| (4) | | | |
| (5) | | | |

# Indirect Triple

- In the indirect triple representation the listing of triples has been done. And listing pointers are used instead of using statement.
- This implementation is called indirect triples.

Triple

| No. | Operator | Arg1 | Arg2 |
|---|---|---|---|
| (0) | uminus | a | |
| (1) | * | (0) | b |
| (2) | uminus | a | |
| (3) | * | (2) | b |
| (4) | + | (1) | (3) |
| (5) | = | x | (4) |

| | Statement |
|---|---|
| (0) | (14) |
| (1) | (15) |
| (2) | (16) |
| (3) | (17) |
| (4) | (18) |
| (5) | (19) |

Indirect Triple

| No. | Operator | Arg1 | Arg2 |
|---|---|---|---|
| (0) | uminus | a | |
| (1) | * | | b |
| (2) | uminus | a | |
| (3) | * | | b |
| (4) | + | | |
| (5) | = | x | |

# PROBLEMS BASED ON QUADRUPLES, TRIPLES & INDIRECT TRIPLES-

***Translate the following expression to quadruple, triple and indirect triple-***

***a + b x c / e ↑ f + b x c***

**Solution-**

Three Address Code for the given expression is-

T1 = e ↑ f
T2 = b x c
T3 = T2 / T1
T4 = b x a
T5 = a + T3
T6 = T5 + T4

Now, we write the required representations-

# Quadruple Representation-

| Location | Op | Arg1 | Arg2 | Result |
|---|---|---|---|---|
| (0) | ↑ | e | f | T1 |
| (1) | x | b | c | T2 |
| (2) | / | T2 | T1 | T3 |
| (3) | x | b | a | T4 |
| (4) | + | a | T3 | T5 |
| (5) | + | T5 | T4 | T6 |

# Triple Representation-

| Location | Op | Arg1 | Arg2 |
|---|---|---|---|
| (0) | ↑ | e | f |
| (1) | x | b | c |
| (2) | / | (1) | (0) |
| (3) | x | b | a |
| (4) | + | a | (2) |
| (5) | + | (4) | (3) |

# Indirect Triple Representation-

| Statement | |
|---|---|
| 35 | (0) |
| 36 | (1) |
| 37 | (2) |
| 38 | (3) |
| 39 | (4) |
| 40 | (5) |

| Location | Op | Arg1 | Arg2 |
|---|---|---|---|
| (0) | ↑ | e | f |
| (1) | x | b | e |
| (2) | / | (1) | (0) |
| (3) | x | b | a |
| (4) | + | a | (2) |
| (5) | + | (4) | (3) |

# Problem-1:

***Translate the following expression to quadruple, triple and indirect triple-***

***a = b x – c + b x – c***

**Solution-**

Three Address Code for the given expression is-

T1 = uminus c

T2 = b x T1

T3 = uminus c

T4 = b x T3

T5 = T2 + T4

a = T5

# Quadruple Representation-

| Location | Op | Arg1 | Arg2 | Result |
|---|---|---|---|---|
| (1) | uminus | c | | T1 |
| (2) | x | b | T1 | T2 |
| (3) | uminus | c | | T3 |
| (4) | x | b | T3 | T4 |
| (5) | + | T2 | T4 | T5 |
| (6) | = | T5 | | a |

# Triple Representation-

| Location | Op | Arg1 | Arg2 |
|---|---|---|---|
| (1) | uminus | c | |
| (2) | x | b | (1) |
| (3) | uminus | c | |
| (4) | x | b | (3) |
| (5) | + | (2) | (4) |
| (6) | = | a | (5) |

# Indirect Triple Representation-

| Statement | |
|---|---|
| 35 | (1) |
| 36 | (2) |
| 37 | (3) |
| 38 | (4) |
| 39 | (5) |
| 40 | (6) |

| Location | Op | Arg1 | Arg2 |
|---|---|---|---|
| (1) | uminus | c | |
| (2) | x | b | (1) |
| (3) | uminus | c | |
| (4) | x | b | (3) |
| (5) | + | (2) | (4) |
| (6) | = | a | (5) |

# SDT for Three Address Code

SDT to generate three address code :

$S \rightarrow id = E \quad \{gen(id.name = E.place);\}$

$E \rightarrow E_1 + T \quad \{E.place = newTemp(); \; gen(E.place = E_1.place + T.place);\}$

$/T \quad \{E.place = T.place;\}$

$T \rightarrow T_1 * F \quad \{T.place = newtemp(); \; gen(T.place = T_1.place * F.place);\}$

$/F \quad \{T.place = F.place;\}$

$F \rightarrow id \quad \{F.place = id.name;\}$

# SDT for TAC for input string : X=a+b*C

SDT to generate three address code :-

$S \rightarrow id = E$ { gen(id.name = E.place); }

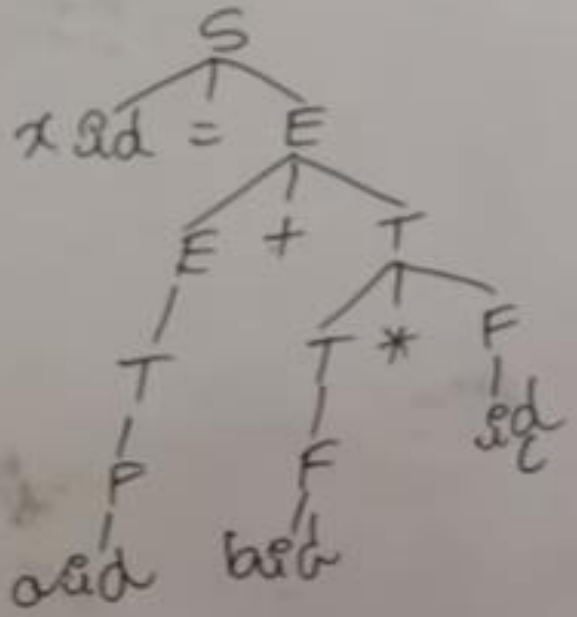
$E \rightarrow E_1 + T$ { E.place = newTemp(); gen(E.place = $E_1$.place + T.place); }

/T { E.place = T.place; }

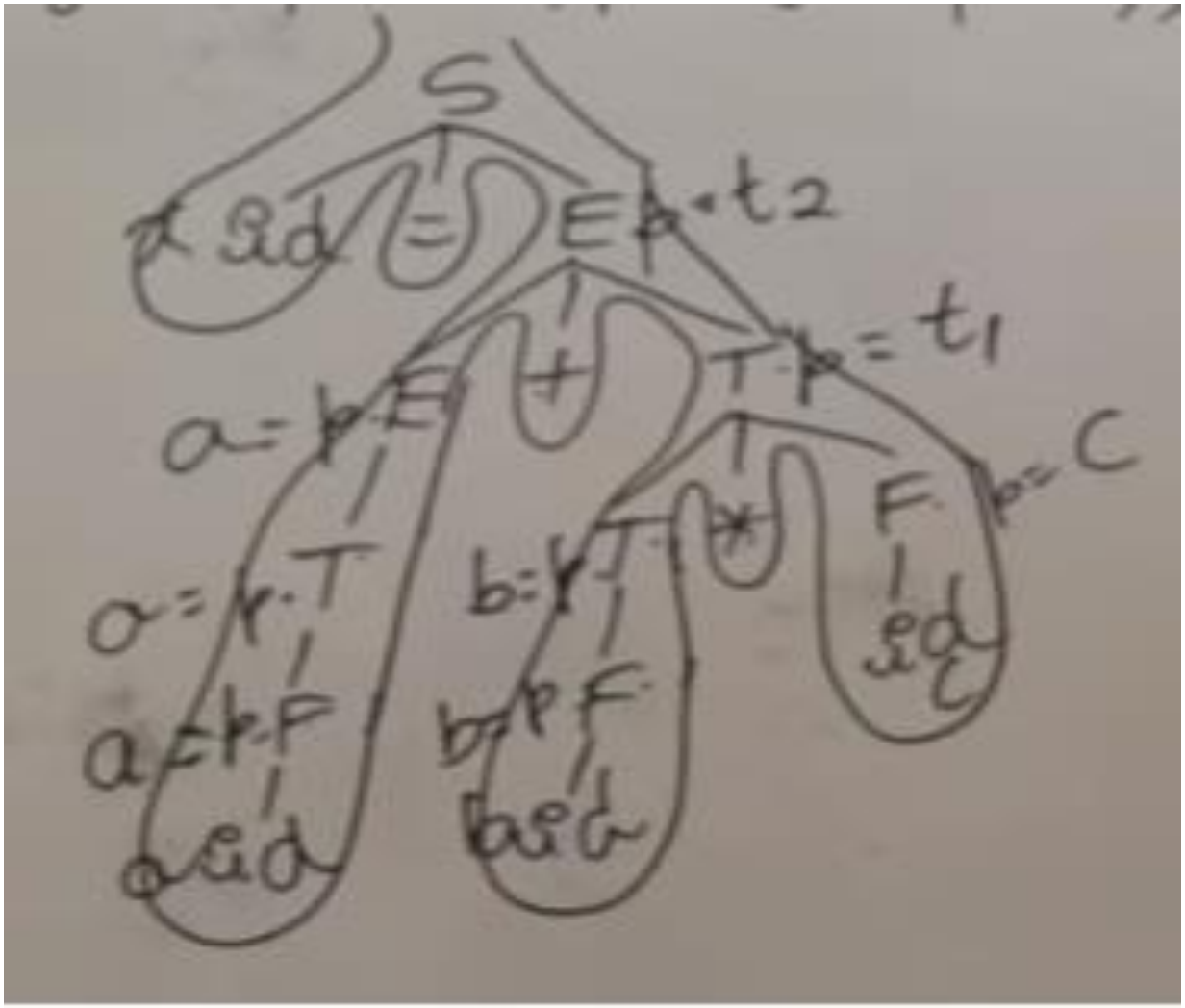
$T \rightarrow T_1 * F$ { T.place = newtemp(); gen(T.place = $T_1$.place * F.place); }

/F { T.place = F.place; }

$F \rightarrow id$ { F.place = id.name; }



# SDT for TAC for input string : X=a+b*C




**X=a+b*c**

**T1=b*c**

**T2 =a+T1**

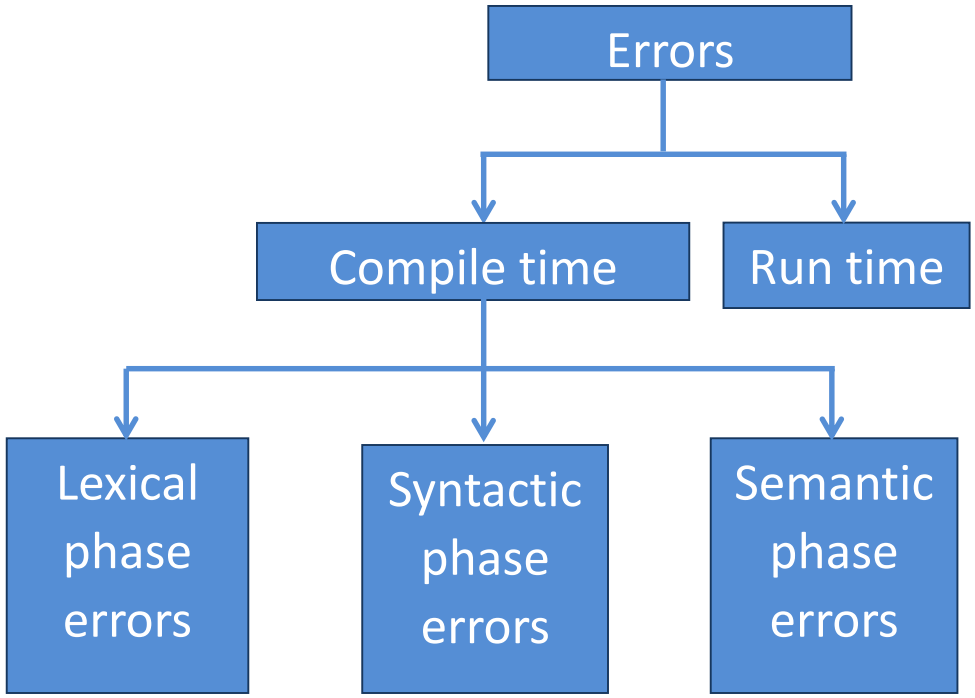
**X=T2**

# Topics to be covered

- Types of errors
- Error recovery strategies

# Types of errors

# Basic types of errors



# Lexical error

- Lexical errors can be detected during lexical analysis phase.
- Typical lexical phase errors are:
  1. Spelling errors
  2. Exceeding length of identifier or numeric constants
  3. Appearance of illegal characters
- Example:

```
fi ( )
{
}
```

- In above code 'fi' cannot be recognized as a misspelling of keyword ***if*** rather lexical analyzer will understand that it is an identifier and will return it as valid identifier.
- Thus misspelling causes errors in token formation.

# Syntax error

- Syntax error appear during syntax analysis phase of compiler.
- Typical syntax phase errors are:
  1. Errors in structure
  2. Missing operators
  3. Unbalanced parenthesis
- The parser demands for tokens from lexical analyzer and if the tokens do not satisfy the grammatical rules of programming language then the syntactical errors get raised.
- Example:

**printf("Hello World !!!")** ← **Error: Semicolon missing**

# Semantic error

- Semantic error detected during semantic analysis phase.
- Typical semantic phase errors are:
  1. Incompatible types of operands
  2. Undeclared variable
  3. Not matching of actual argument with formal argument
- Example:

  id1=id2+id3*60 (Note: id1, id2, id3 are real)

  (Directly we can not perform multiplication due to incompatible types of variables)

# Error recovery strategies (Ad-Hoc & systematic methods)

# Error recovery strategies (Ad-Hoc & systematic methods)

- There are mainly four error recovery strategies:
  1. Panic mode
  2. Phrase level recovery
  3. Error production
  4. Global generation

# Panic mode

- In this method on discovering error, the parser discards input symbol one at a time. This process is continued until one of a designated set of synchronizing tokens is found.
- Synchronizing tokens are delimiters such as semicolon or end.
- These tokens indicate an end of the statement.
- If there is less number of errors in the same statement then this strategy is best choice.

```
fi ( ) ⟵ Scan entire line otherwise scanner will return fi as valid identifier
{
}
```

# Phrase level recovery

- In this method, on discovering an error parser performs local correction on remaining input.
- The local correction can be replacing comma by semicolon, deletion of semicolons or inserting missing semicolon.
- This type of local correction is decided by compiler designer.
- This method is used in many error-repairing compilers.

# Error production

- If we have good knowledge of common errors that might be encountered, then we can augment the grammar for the corresponding language with error productions that generate the erroneous constructs.
- Then we use the grammar augmented by these error production to construct a parser.
- If error production is used then, during parsing we can generate appropriate error message and parsing can be continued.

# Global correction

- Given an incorrect input string x and grammar G, the algorithm will find a parse tree for a related string y, such that number of insertions, deletions and changes of token require to transform x into y is as small as possible.
- Such methods increase time and space requirements at parsing time.
- Global correction is thus simply a theoretical concept.

# *Give the syntax directed definition for if-else statement.*

Ans:

| Production | Semantic rule |
| --- | --- |
| $S \rightarrow$ if E then $S_1$ else $S_2$ | E.true := newlabel;<br>E.false := newlabel;<br>$S_1$.next := S.next<br>$S_2$.next := S.next<br>S.code := E.code \|\| gen(E.true ':') \|\| $S_1$.code \|\| gen('goto' S.next) \|\| gen(E.false ':') \|\| $S_2$.code |

# Exercise

Write quadruple, triple and indirect triple for following:

1. -(a*b)+(c+d)
2. a*-(b+c)
3. x=(a+b*c)^(d*e)+f*g^h
4. g+a*(b-c)+(x-y)*d