SUBJECT- COMPILER DESIGN

SEMESTER- 5TH SEM

VISSION AND MISSION OF INSTITUTE

To become a renowned center of outcome based learning and work towards academic, professional, cultural and social enrichment of the lives of individuals and communities

M1: Focus on evaluation of learning outcomes and motivate students to inculcate research aptitude by project based learning.

M2: Identify, based on informed perception of Indian, regional and global needs, the areas of focus and provide platform to gain knowledge and solutions.

M3: Offer opportunities for interaction between academia and industry.

M4: Develop human potential to its fullest extent so that intellectually capable and imaginatively gifted leaders can emerge in a range of professions.

VISION OF THE DEPARTMENT

To become renowned Centre of excellence in computer science and engineering and make competent engineers & professionals with high ethical values prepared for lifelong learning.

MISION OF THE DEPARTMENT

M1: To impart outcome based education for emerging technologies in the field of computer science and engineering.

M2: To provide opportunities for interaction between academia and industry.

M3: To provide platform for lifelong learning by accepting the change in technologies.

M4: To develop aptitude of fulfilling social responsibilities

PROGRAM OUTCOMES

Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

Problem analysis: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

COURSE OUTCOME

CO1: Compare different phases of compiler and design lexical analyzer. CO2: Examine syntax and semantic analyzer by understanding grammars.

CO3: Illustrate storage allocation and its organization & analyze symbol table organization.

CO4: Analyze code optimization, code generation & compare various compilers.

CO-PO Mapping

Semester	Subject	Code	L/T/P	CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
V	COMPILER DESIGN	5CS4 - 02	L	1. Compare different phases of compiler and design lexical analyzer.	3	3	3	3	2	1	1	1	1	2	1	3
			L	2. Examine syntax and semantic analyzer and illustrate storage allocation and its organization	3	3	3	3	1	1	1	1	1	2	2	3
			L	3. Analyze symbol table organization, code optimization and code generator	3	3	3	3	2	1	1	1	1	2	2	3
			L	4.Compare and evaluate various compilers and analyzers	3	3	3	3	2	1	1	1	1	2	1	3

PROGRAM EDUCATIONAL OBJECTIVES:

1. To provide students with the fundamentals of Engineering Sciences with more emphasis in **Computer Science &Engineering** by way of analyzing and exploiting engineering challenges.
2. To train students with good scientific and engineering knowledge so as to comprehend, analyze, design, and create novel products and solutions for the real life problems.
3. To inculcate professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, entrepreneurial thinking and an ability to relate engineering issues with social issues.
4. To provide students with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the self motivated life-long learning needed for a successful professional career.
5. To prepare students to excel in Industry and Higher education by Educating Students along with High moral values and Knowledge

PSO

PSO1. Ability to interpret and analyze network specific and cyber security issues, automation in real word environment.

PSO2. Ability to Design and Develop Mobile and Web-based applications under realistic constraints.

SYLLABUS

RAJASTHAN TECHNICAL UNIVERSITY, KOTA

Syllabus

III Year-V Semester: B.Tech. Computer Science and Engineering

5CS4-02: Compiler Design

Credit: 3 | **Max. Marks: 150(IA:30, ETE:120)**

3L+0T+0P | **End Term Exam: 3 Hours**

SN	Contents	Hours
1	**Introduction:**Objective, scope and outcome of the course.	01
2	**Introduction:** Objective, scope and outcome of the course. Compiler, Translator, Interpreter definition, Phase of compiler, Bootstrapping, Review of Finite automata lexical analyzer, Input, Recognition of tokens, Idea about LEX: A lexical analyzer generator, Error handling.	06
3	**Review of CFG Ambiguity of grammars:** Introduction to parsing. Top down parsing, LL grammars & passers error handling of LL parser, Recursive descent parsing predictive parsers, Bottom up parsing, Shift reduce parsing, LR parsers, Construction of SLR, Conical LR & LALR parsing tables, parsing with ambiguous grammar. Operator precedence parsing, Introduction of automatic parser generator: YACC error handling in LR parsers.	10
4	**Syntax directed definitions;** Construction of syntax trees, S-Attributed Definition, L-attributed definitions, Top down translation. Intermediate code forms using postfix notation, DAG, Three address code, TAC for various control structures, Representing TAC using triples and quadruples, Boolean expression and control structures.	10
5	**Storage organization;** Storage allocation, Strategies, Activation records, Accessing local and non-local names in a block structured language, Parameters passing, Symbol table organization, Data structures used in symbol tables.	08
6	**Definition of basic block control flow graphs;** DAG representation of basic block, Advantages of DAG, Sources of optimization, Loop optimization, Idea about global data flow analysis, Loop invariant computation, Peephole optimization, Issues in design of code generator, A simple code generator, Code generation from DAG.	07

LECTURE PLAN:

Subject: Compiler Design (5CS4 – 02) **Year/Sem: III/V**

Unit No./ Total lec. Req.	Topics	Lect. Req.
Unit-1 (6)	Compiler, Translator, Interpreter definition, Phase of compiler	1
	Introduction to one pass & Multipass compilers, Bootstrapping	1
	Review of Finite automata lexical analyzer, Input, buffering,	2
	Recognition of tokens, Idea about LEX:, GATE Questions	1
	A lexical analyzer generator, Error Handling, Unit Test	1
Unit-2 (17)	Review of CFG Ambiguity of grammars, Introduction to parsing	2
	Bottom up parsing Top down Parsing Technique	5
	Shift reduce parsing, Operator Precedence Parsing	2
	Recursive descent parsing predictive parsers	1
	LL grammars & passers error handling of LL parser	1
	Conical LR & LALR parsing tables	3
	parsing with ambiguous grammar, GATE Questions	2
	Introduction of automatic parser generator: YACC error handling in LR parsers, Unit Test	1
Unit 3- (7)	Syntax directed definitions; Construction of syntax trees	1
	L-attributed definitions, Top down translation	1
	Specification of a type checker, GATE Questions	1
	Intermediate code forms using postfix notation and three address code,	2
	Representing TAC using triples and quadruples, Translation of assignment statement.	1
	Boolean expression and control structures, Unit Test	1
Unit 4- (4)	Storage organization, Storage allocation, Strategies, Activation records,	1
	Accessing local and non local names in a block structured language	1
	Parameters passing, Symbol table organization, GATE Questions	1
	Data structures used in symbol tables, Unit Test	1
Unit 5- (6)	Definition of basic block control flow graphs,	1
	DAG representation of basic block, Advantages of DAG,	1
	Sources of optimization, Loop optimization Idea about global data flow analysis, Loop invariant computation, Loop invariant computation, Tutorial	2
	Peephole optimization, GATE Questions, Tutorial	1
	Issues in design of code generator, A simple code generator, Code generation from DAG., UNIT TEST, Revision	1

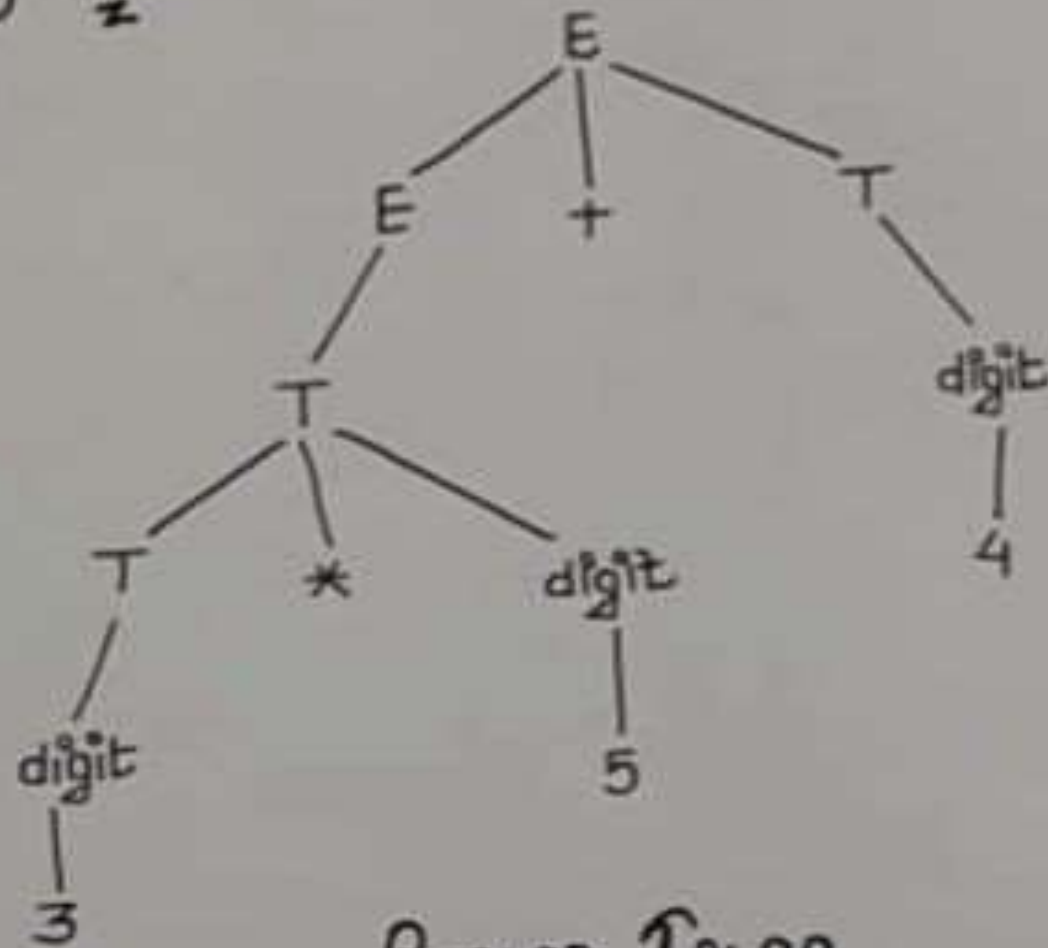
JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

Year & Sem – 3rd/ 5th sem
Subject – Compiler Design
Unit – 3

Definition

A syntax tree (also known as abstract syntax tree) is a condensed form of parse tree.

Example:



Parse Tree

Syntax Tree

Parse Tree Vs Syntax Tree

Parse Tree	Syntax Tree
A parse tree is a graphical representation of the replacement process in a derivation.	A syntax tree is a condensed form of parse tree.
In parse trees, → each interior node represents a grammar rule → each leaf node represents a terminal	In syntax trees, → each interior node represents an operator → each leaf node represents an operand

Parse trees represent every detail from the real syntax	Syntax trees does not represent every detail from the real syntax (that's why they are called abstract) For example- no rule nodes, no parentheses etc
Parse trees are less dense compared to syntax trees for the same language construct	Syntax trees are more dense compared to parse trees for the same language construct.

——— xxxxxxx ———

Problem-01 :

Consider the following grammar-

$$E \rightarrow E + T \,/\, T$$

$$T \rightarrow T * F \,/\, F$$

$$F \rightarrow (E) \,/\, id$$

For the string,

$$w = id + id * id$$

Generate-

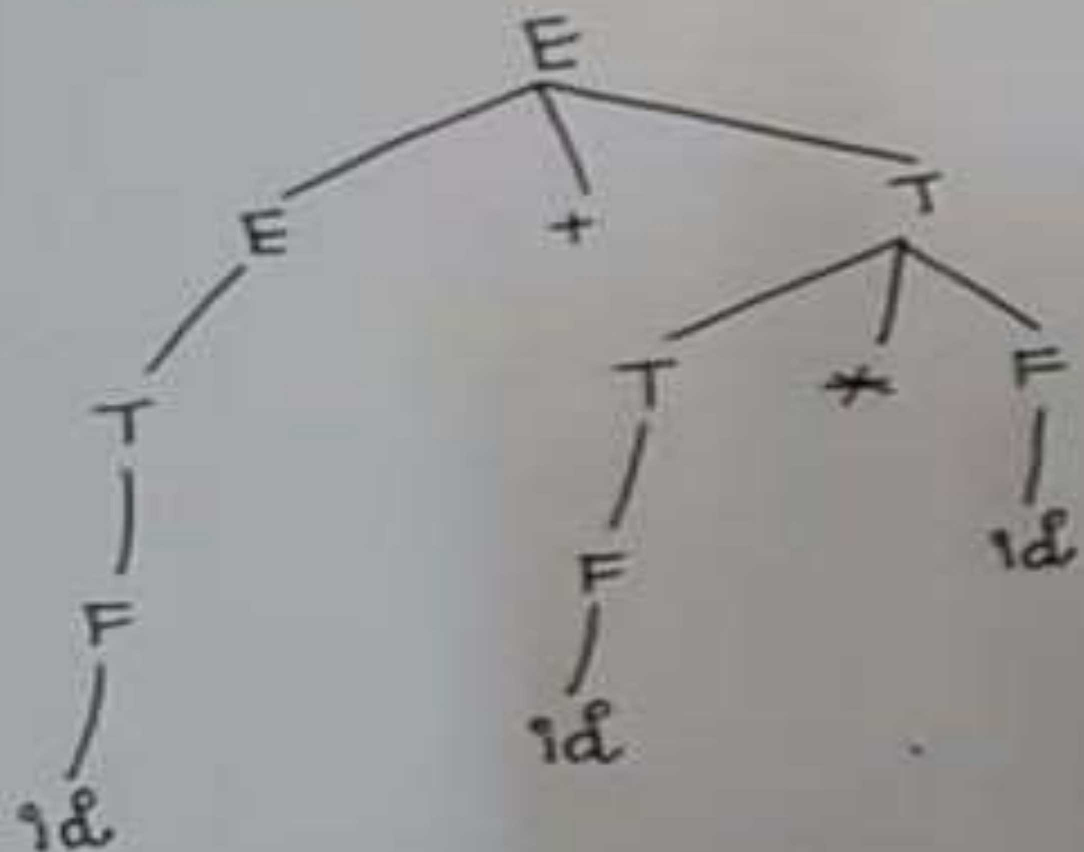
i) Parse tree

ii) Syntax tree

Solution:- The given grammar is-

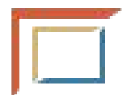
$$E \longrightarrow E+T \,/\, T$$
$$T \longrightarrow T*F \,/\, F$$
$$F \longrightarrow (E) \,/\, id$$

$w = id + id * id$



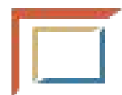
Syntax-directed translation

- 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

- There are two ways to represent the semantic rules associated with grammar symbols.
  - Syntax-Directed Definitions (SDD)
  - Syntax-Directed Translation Schemes (SDT)



Syntax-Directed Definitions

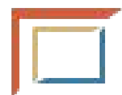
- A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules.
- Attributes are associated with grammar symbols and rules are associated with productions.

PRODUCTION

$E \rightarrow E1 + T$

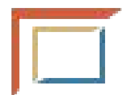
SEMANTIC RULE

E.code = E1.code || T.code || ‘+’



Syntax-Directed Translation

- 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.



Syntax directed Translation (SDT)

→ Grammar + semantic rules = SDT

Informal notation

→ Along with the grammar, we're giving semantic actions also s.t along with the parsing task, some other tasks can also be done in parallel like code generation, ICG, expression evaluation, etc
EE

eg:- SDT for EE :

$E \rightarrow E + T$ { E.val = E.val + T.val }
$\quad / T$ { E.val = T.val }
$T \rightarrow T * F$ { T.val = T.val * F.val }
$\quad / F$ { T.val = F.val }
$F \rightarrow num$ { F.val = num.value }

let exp : 2 + 3 * 4

= 14

Generate parse tree

S – attributed and L – attributed SDTs in Syntax directed translation

Types of attributes –

Attributes may be of two types – Synthesized or Inherited.

Synthesized attributes –

A Synthesized attribute is an attribute of the non-terminal on the left-hand side of a production. Synthesized attributes represent information that is being passed up the parse tree. The attribute can take value only from its children (Variables in the RHS of the production).

For eg. let's say A -> BC is a production of a grammar, and A's attribute is dependent on B's attributes or C's attributes then it will be synthesized attribute.

Inherited attributes –

An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute. The attribute can take value either from its parent or from its siblings (variables in the LHS or RHS of the production).

For example, let's say A -> BC is a production of a grammar and B's attribute is dependent on A's attributes or C's attributes then it will be inherited attribute.

1. **S-attributed SDT :**
    - If an SDT uses only synthesized attributes, it is 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.
    - Semantic actions are placed in rightmost place of RHS.

2. **L-attributed SDT:**

L-attributed SDT This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.
In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production
S → ABC S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

Grammar Symbol 'X' may be characterisized by two types of attributes/Values.

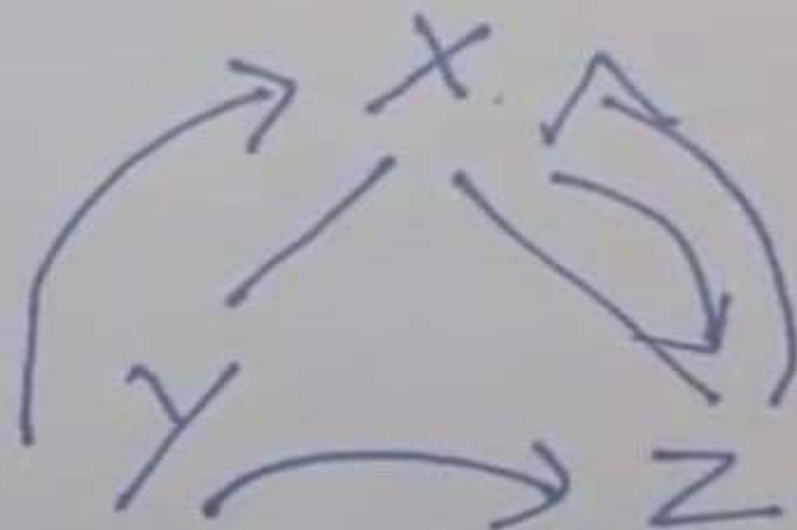
1. Synthesized attribute
2. Inherited attribute.

$X \rightarrow YZ$

1. Synthesized attribute

2. Inherited attribute.

$X \rightarrow YZ$



$X.\underline{a} = f(Y, Z)$ → Synthesis

$Z.a$ — inherited attribute

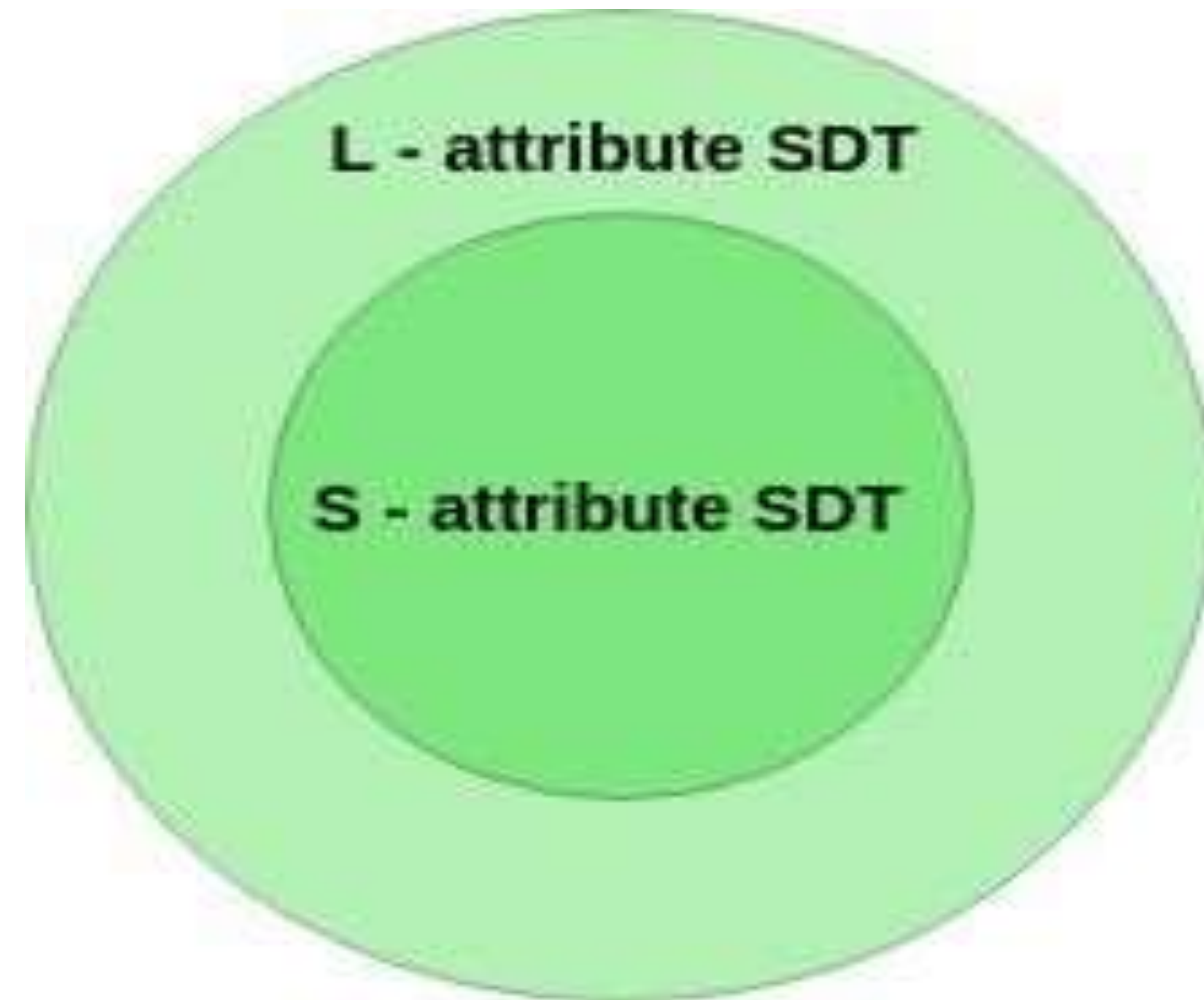
S-Attributed SDT vs. L-Attributed SDT

S-Attributed SDT	L-Attributed SDT
1. Uses only synthesized attributes	Uses both synthesized and inherited attributes.
2. Evaluated in bottom-up parsing (post order traversal)	Evaluated in depth-first and left to right parsing manner.
3. Semantic actions are placed in the rightmost of RHS of production rule. $A \rightarrow BCD$	Semantic actions are placed anywhere in RHS.

2. Evaluated in bottom-up parsing (post order traversal)	Evaluated in depth-first and left to right parsing manner.
3. Semantic actions are placed in the rightmost of RHS of production Rule. $A \to BCD \{ \ \}$	Semantic actions are placed anywhere in RHS. $A \to BCD$ $A \to \{ \ \} BCD$ $A \to B \{ \ \} CD$ $A \to BCD \{ \ \}$

S-Attributed SDT	L-Attributed SDT
(4) Post order traversal Order	Depth first Order and left to right.
(5) Eg:- $A \to XY$, $A.a = f(X, Y)$	$A \to XYZ$, $A.a = f(X, Y, Z)$, $X.b = f(X)$, $Z.c = f(X, Y)$
(6)	

If a definition is S-attributed, then it is also L-attributed but **NOT** vice-versa.



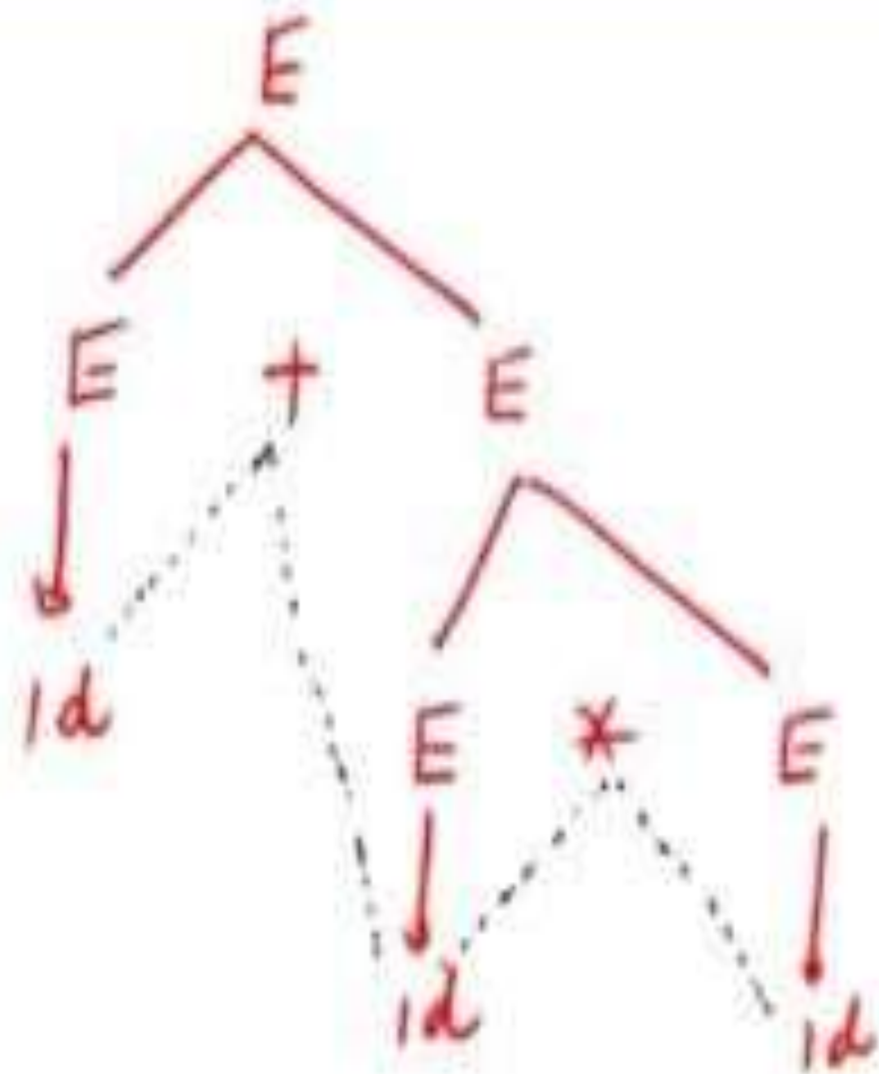
Syntax directed translation

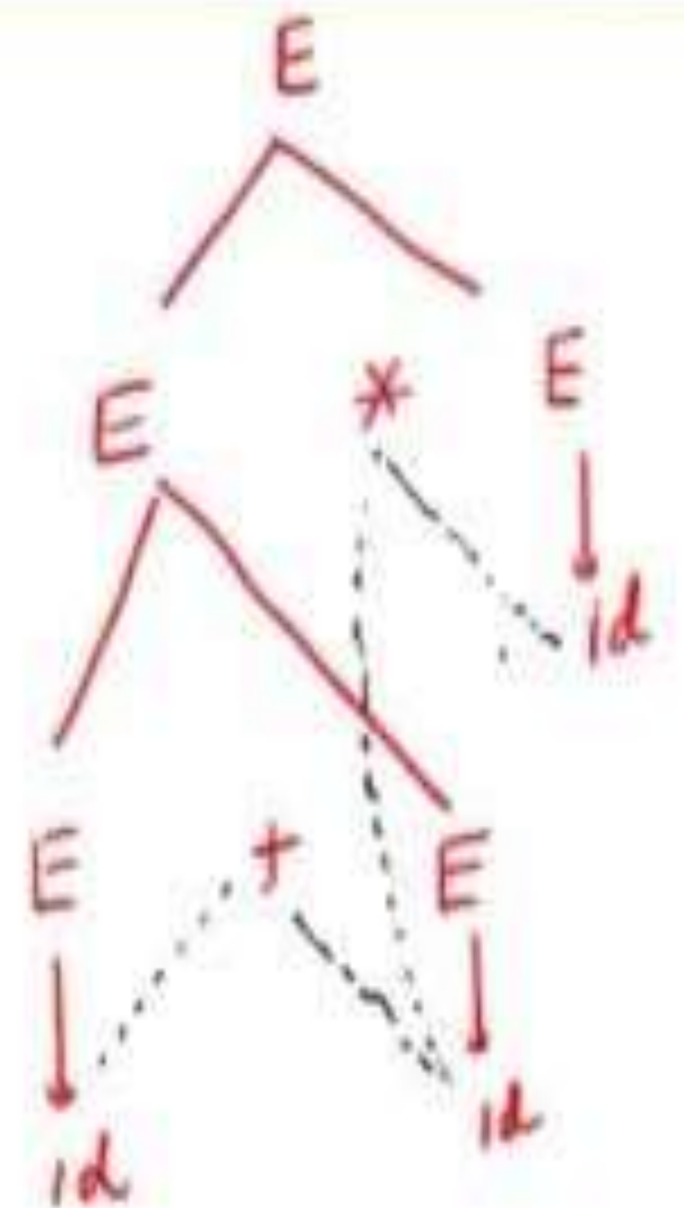
Rule to construct the SDT

- Define the grammar of looking at input string.
- Construct the parse tree
- Attach semantic rule by looking at expected output.

$E \rightarrow E+E \mid E*E \mid id$

I/P = id+id*id

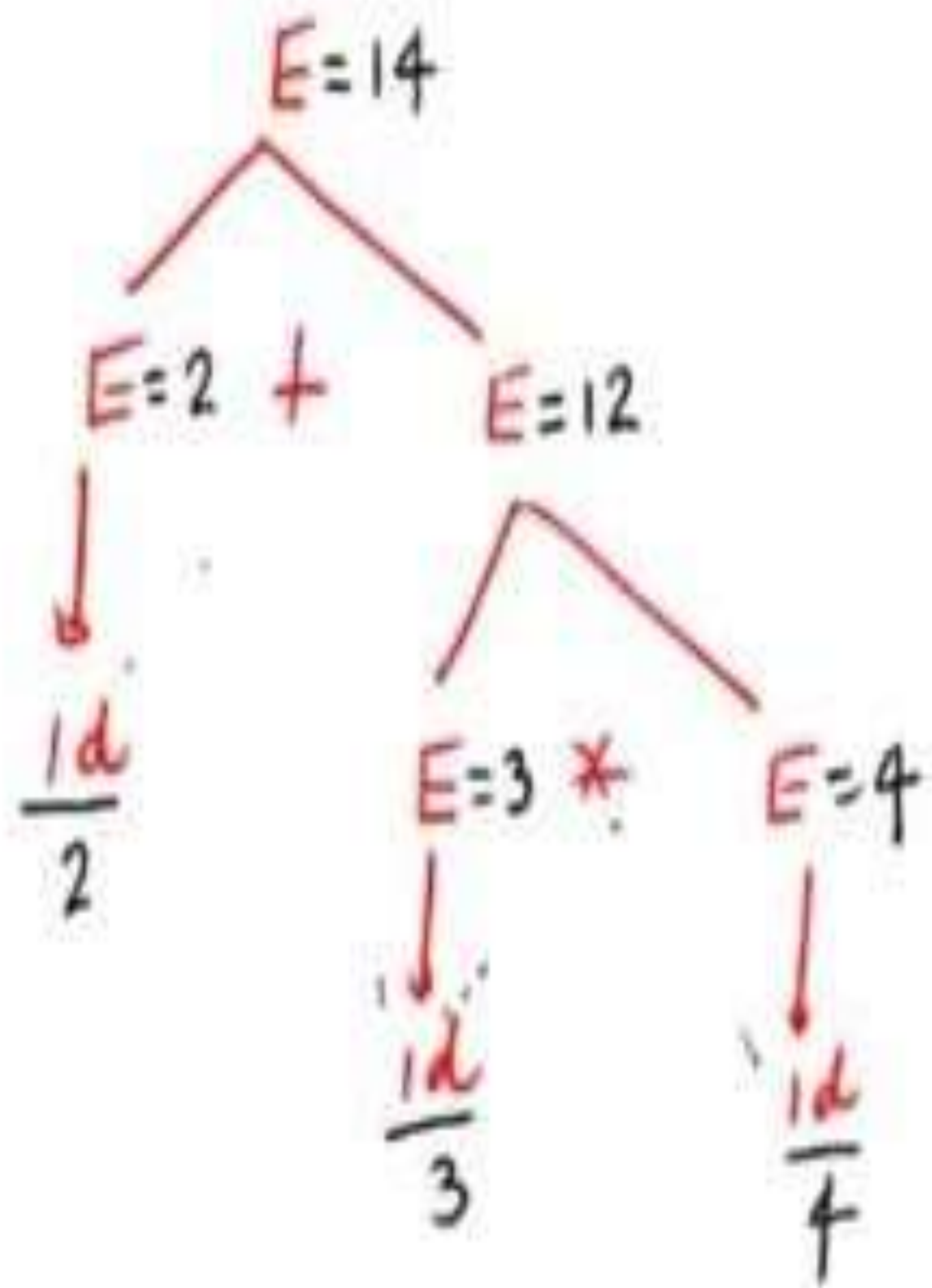


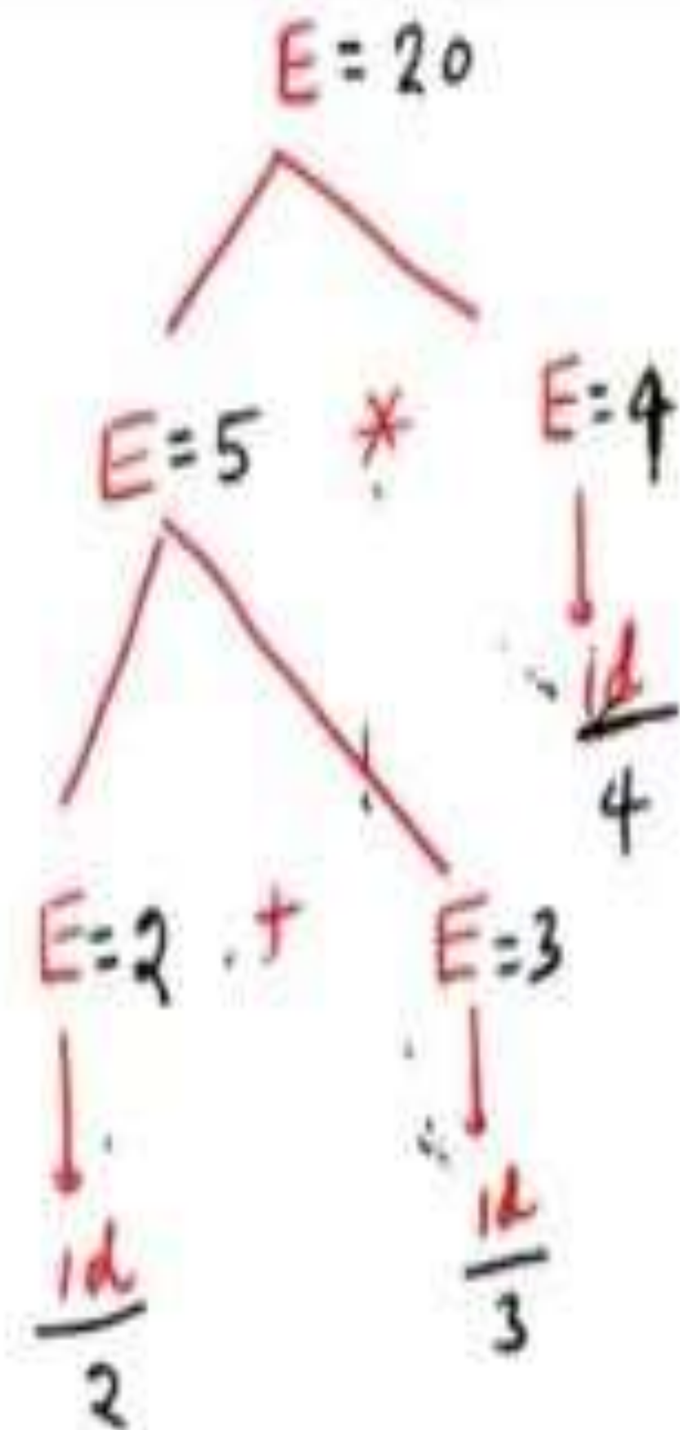


$E \rightarrow E+E \mid E*E \mid id$

I/P = id+id*id

$2+(3*4)=14$





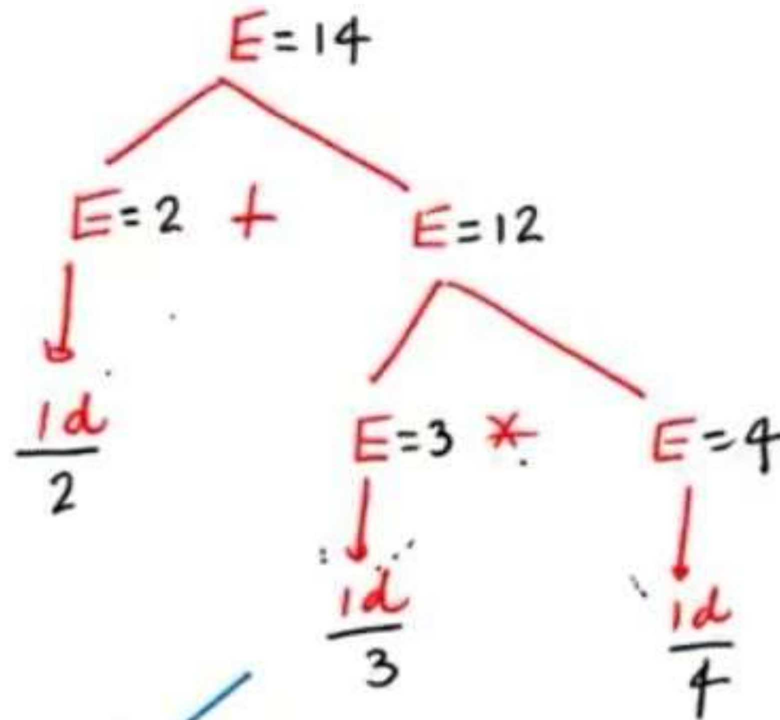
gradeup

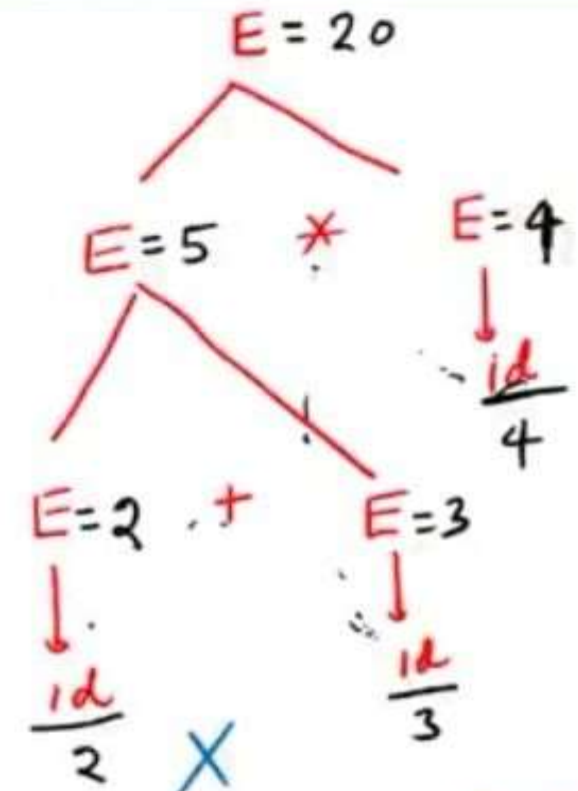
Syntax directed translation

$E \rightarrow E+E \mid E*E \mid id$

i/p = id+id*id

$2+(3*4)=14$

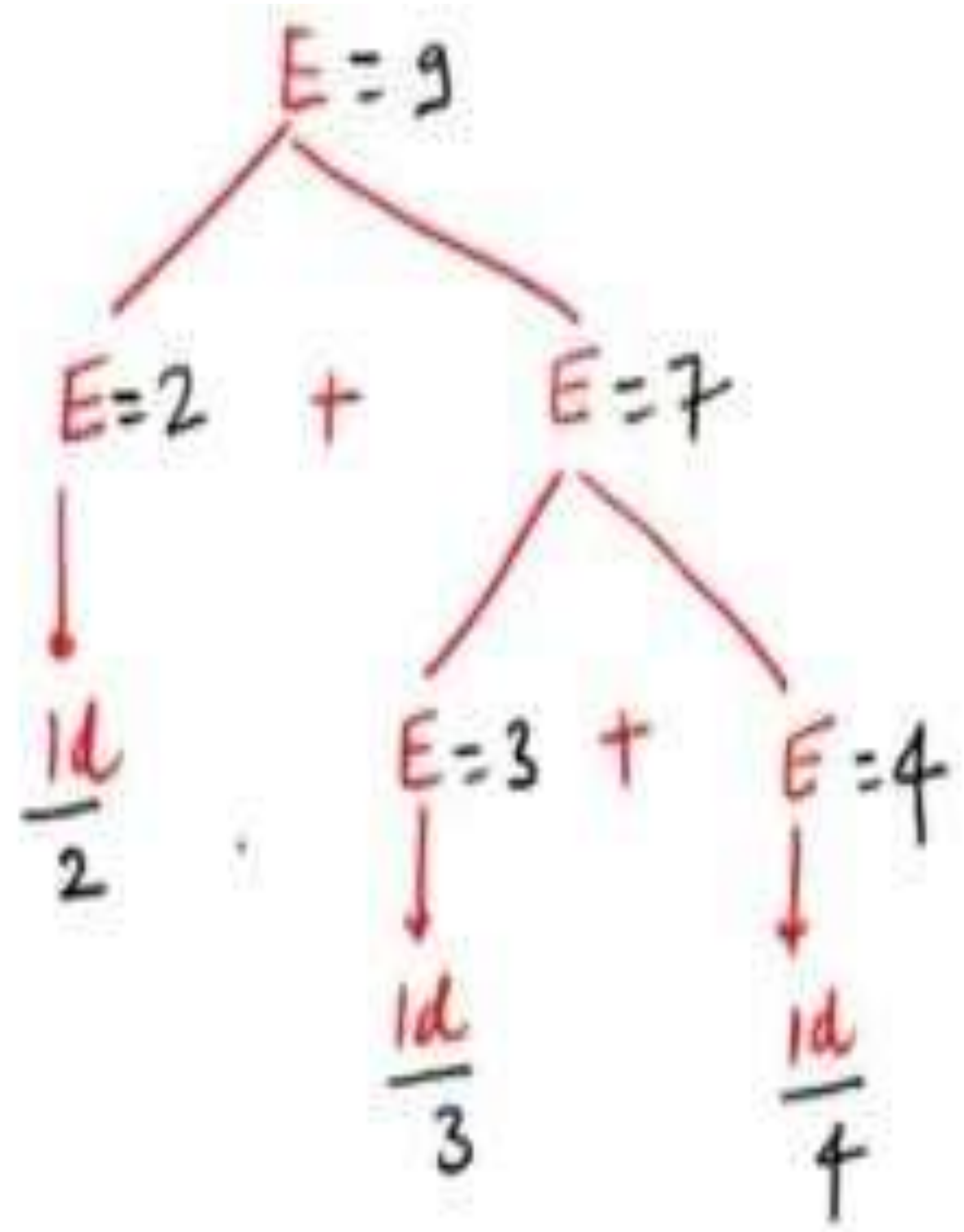


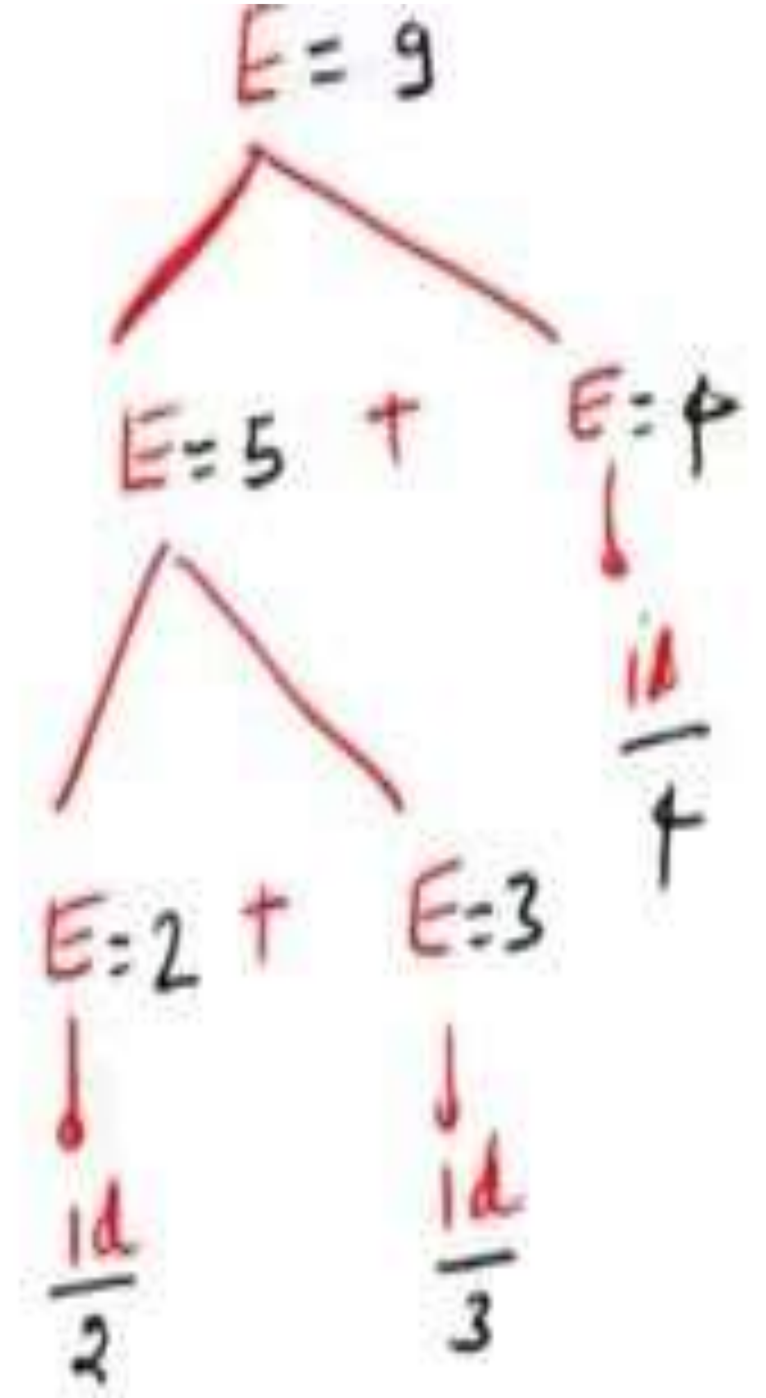


GATE

$E \rightarrow E+E \mid E*E \mid id$

i/p = $\left(\begin{array}{c} id+id+id \\ 2+3+4=9 \end{array} \right)$





gradeup

Syntax directed translation

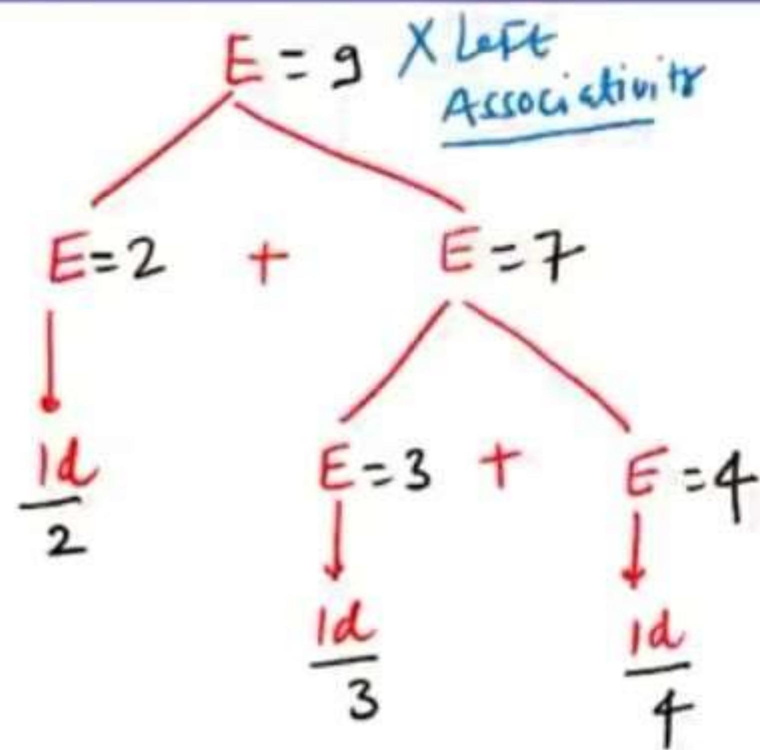
$E \rightarrow E+E \mid E*E \mid id$

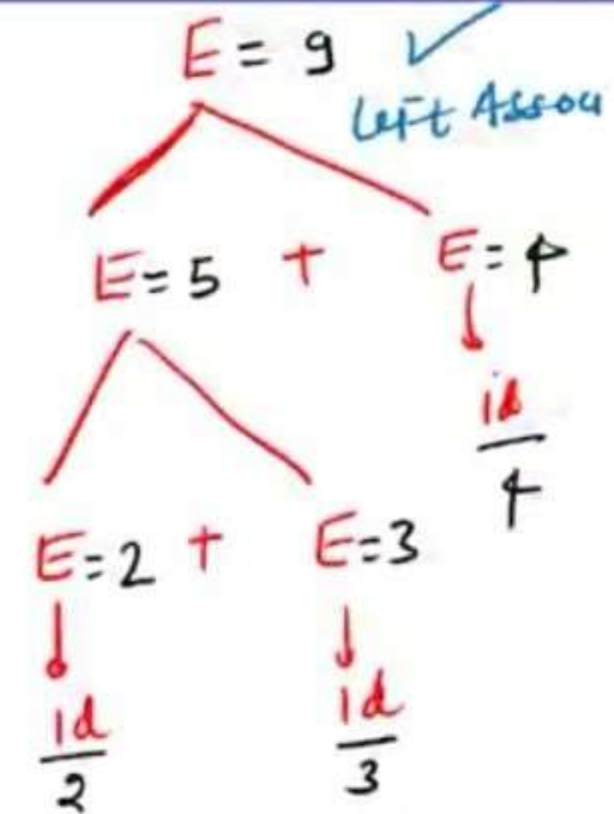
i/p = (id + id + id)

((2 + 3) + 4) = 9

((2+3)+4) = 9

(((2+3)+4)+5) = 14





+ → Left Associative
* → Left Assciative
∧ → Right Associative

((2*3)*4) = 24

(2∧(3∧4))

or

Associativity → Recursion

Left Associativity → Left Recursion

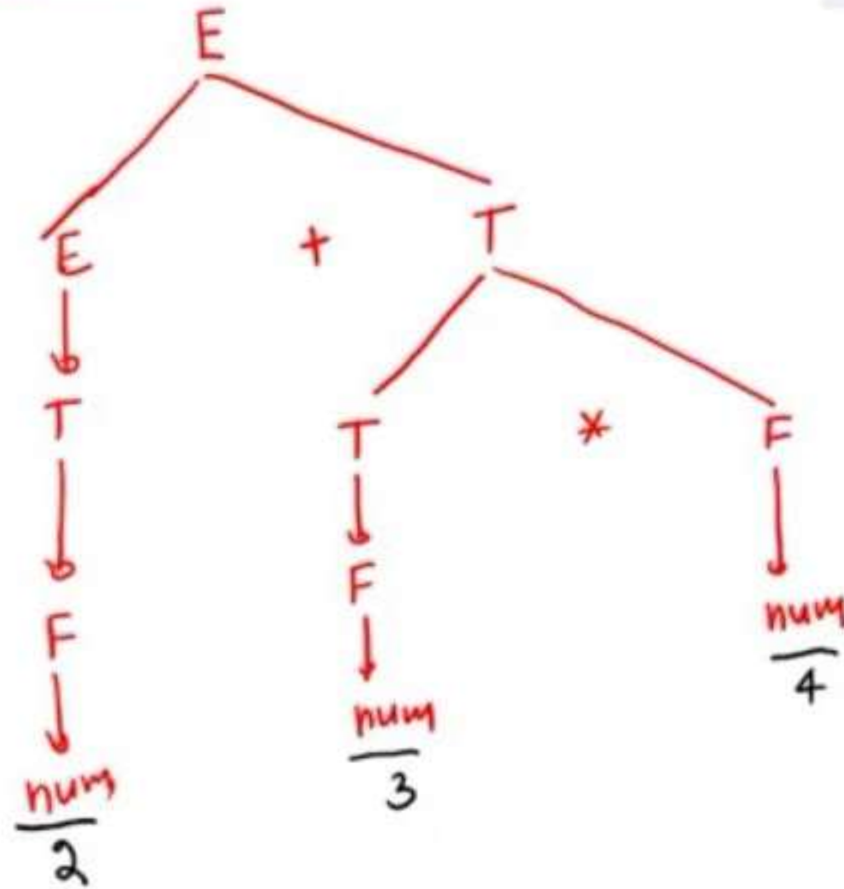
Right Associativity → Right Recursive

Precedence → Level

↑ (High) ↓ (Low)

↓ (Low) ↑ (High)

Syntax directed translation



SDT for evaluation of an Airthmatic express

E → E + T	{E. Val = E.Val + T. Val}
E→ T	{e.Val = T.Val}
T → T*F	{T. Value = T. value * F. Val}
T → F	{T. Value = F. Val}
F → num	{F. Val = num.Val}

GATE

SDT EVALUATION USING BOTTOM UP PARSER

Syntax directed translation

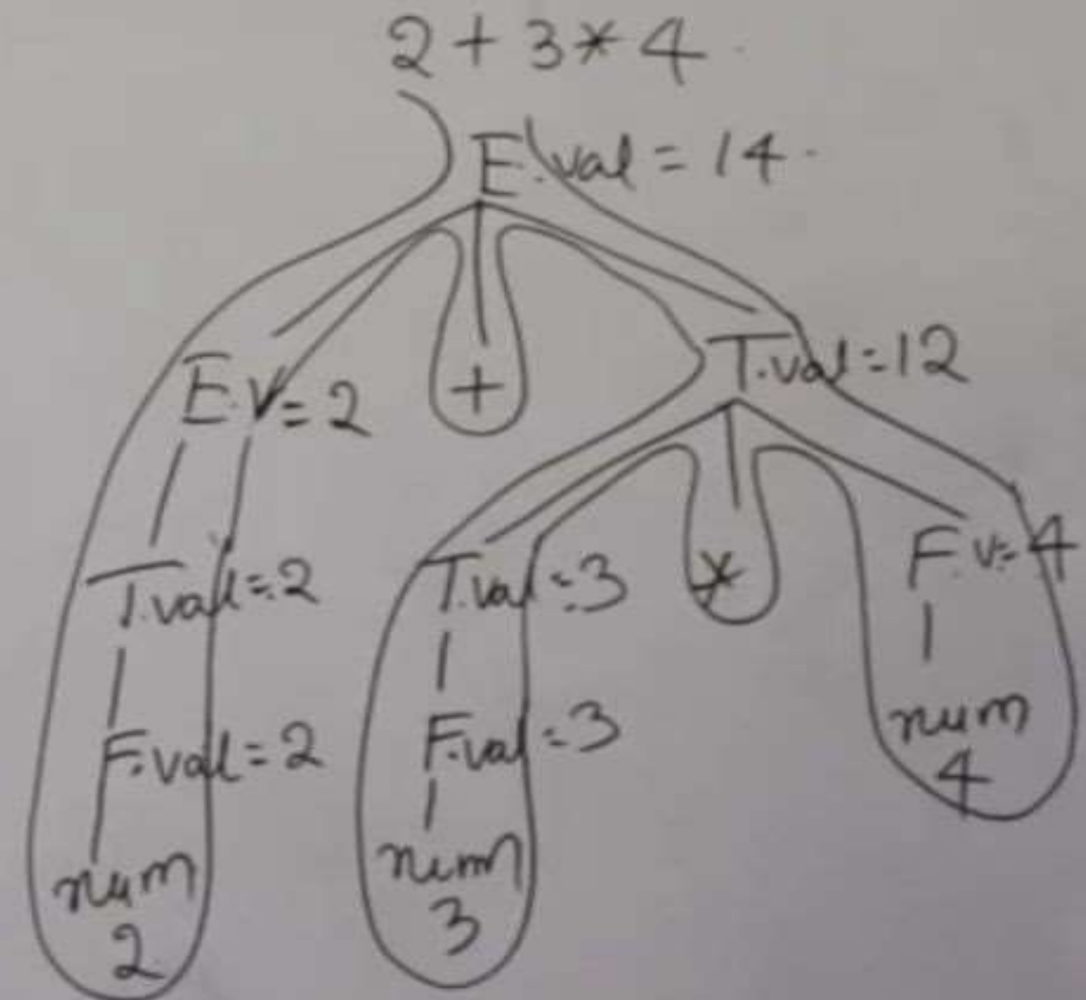
Grammar + Semantic rules = SDT

SDT for evaluation of expression

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

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

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

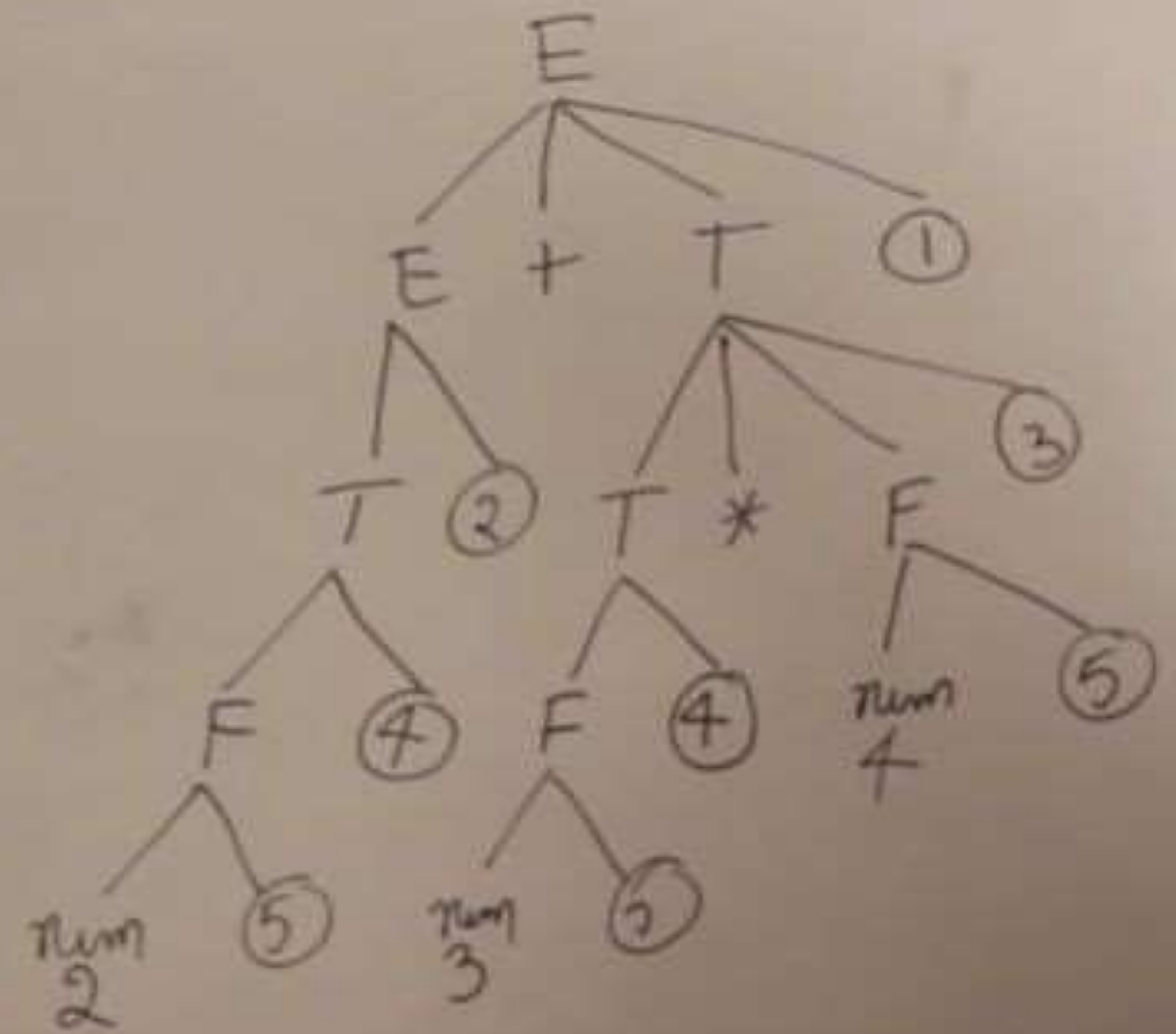


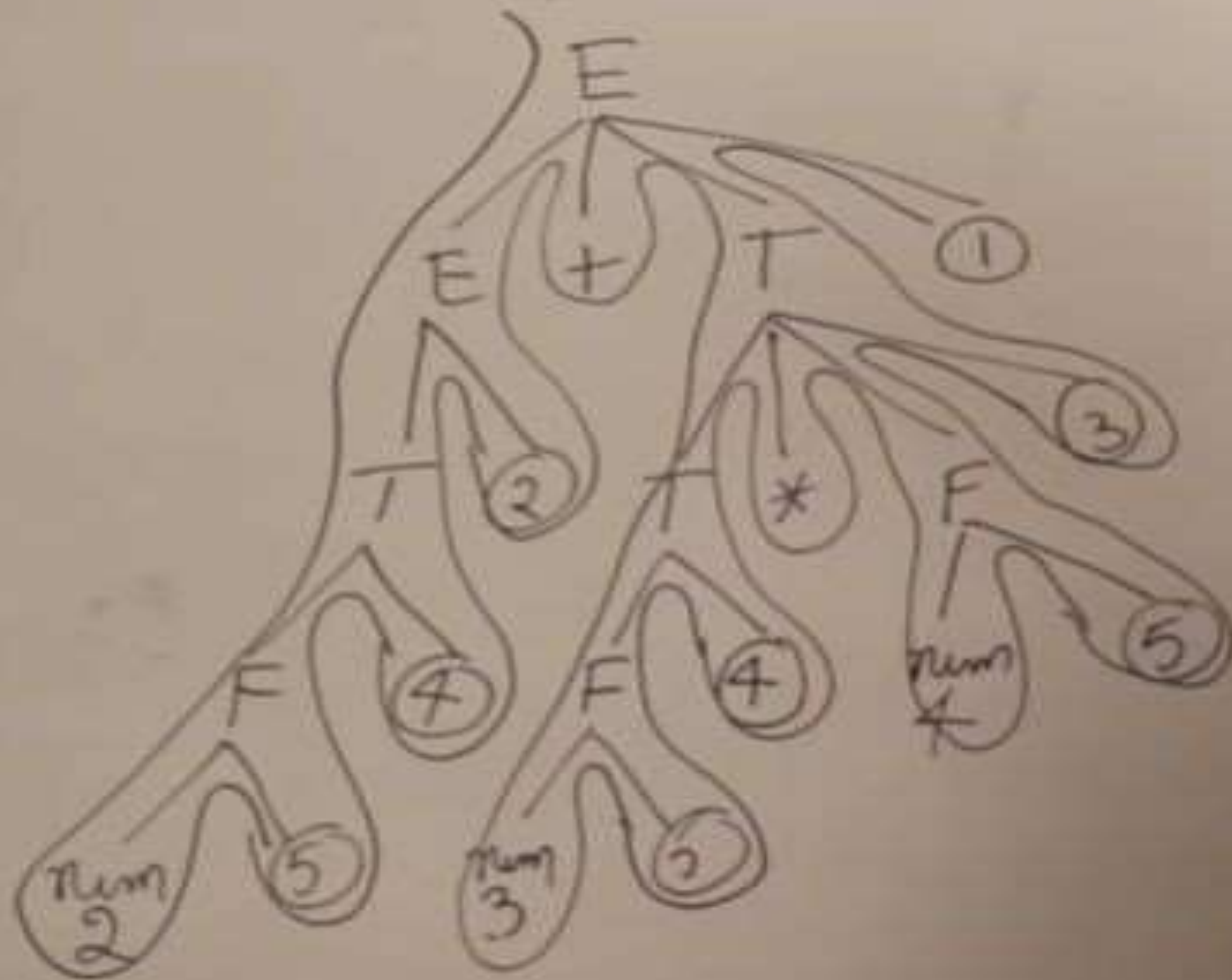
RAUDRA

SDT EVALUATION USING TOP DOWN PARSER

Convert the Infix Expression 2+3*4 to Postfix for the given SDT Using Top down Parser

```
E -> E+T  { Printf("+"); }        (1)
   / T    { }                     (2)
T -> T*F  { printf("*"); }        (3)
   / F    { }                     (4)
F -> num  { Printf(num.lval); }   (5)
```

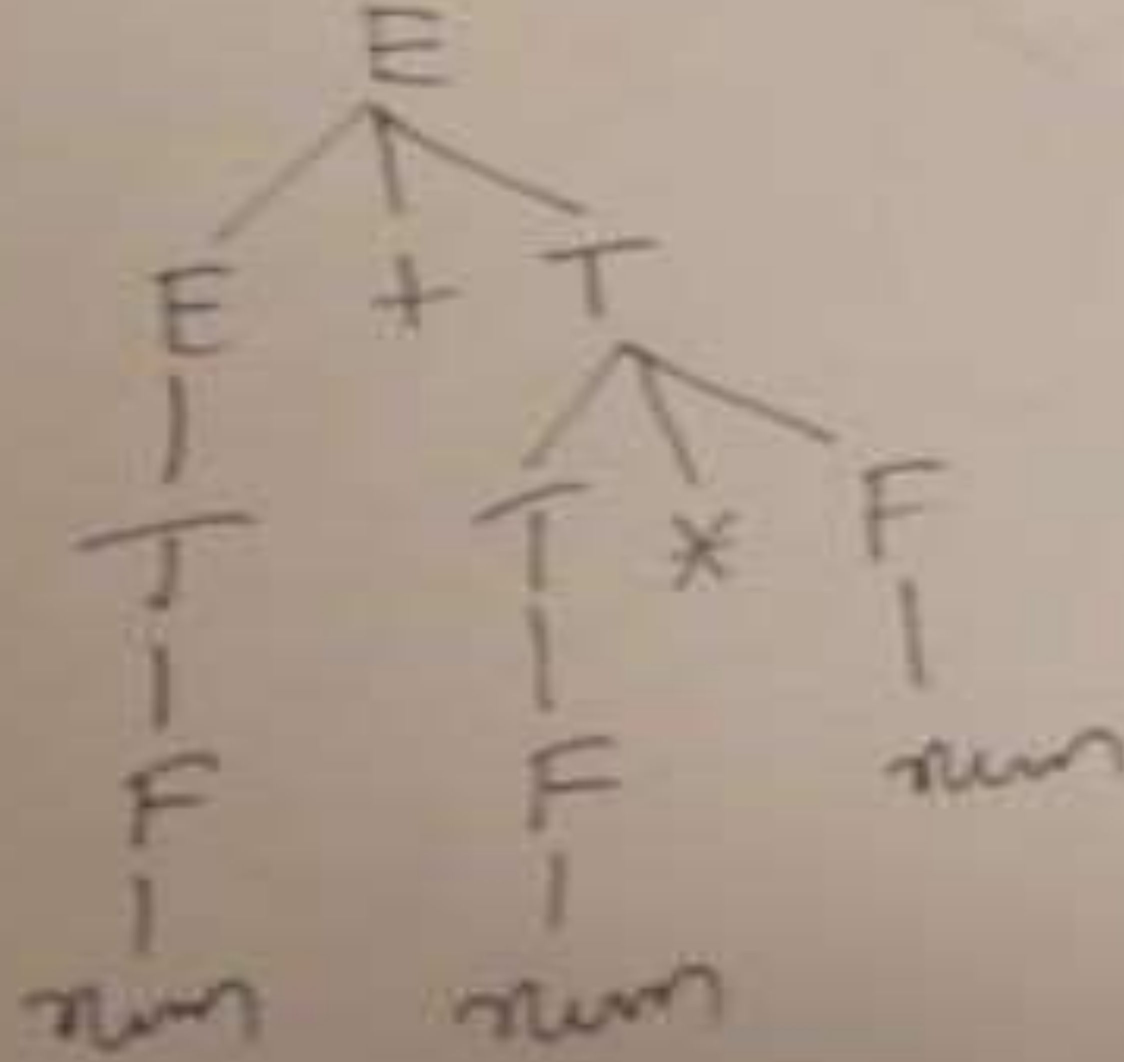

2+3*4.
E
E + T ①
T ② T * F ③
F ④ F ④ num 4 ⑤
num 2 ⑤ num 3



2+3*4.
234*+
E
E
+
T
T
F
F
F
num 2
num 3
num 4

Convert the Infix Expression 2+3*4 to Postfix for the given SDT Using Bottom Up Parser

```
E -> E+T   { printf("+"); }        (1)
   | T     { }                     (2)
T -> T*F   { printf("*"); }        (3)
   | F     { }                     (4)
F -> num   { printf(num.lval); }   (5)
```


E
E
+
T
T
T
*
F
F
F
num
num
num

2+3*4

234*+

E

E + T

T * F

T F num 4

F num 3

num 2

✓ 234*+

Intermediate Code generation

What is intermediate code?

Clip slide

During the translation of a source program into the object code for a target machine, a compiler may generate a middle-level language code, which is known as **intermediate code**

intermediate code

Clip slide

The following are commonly used intermediate code representation :

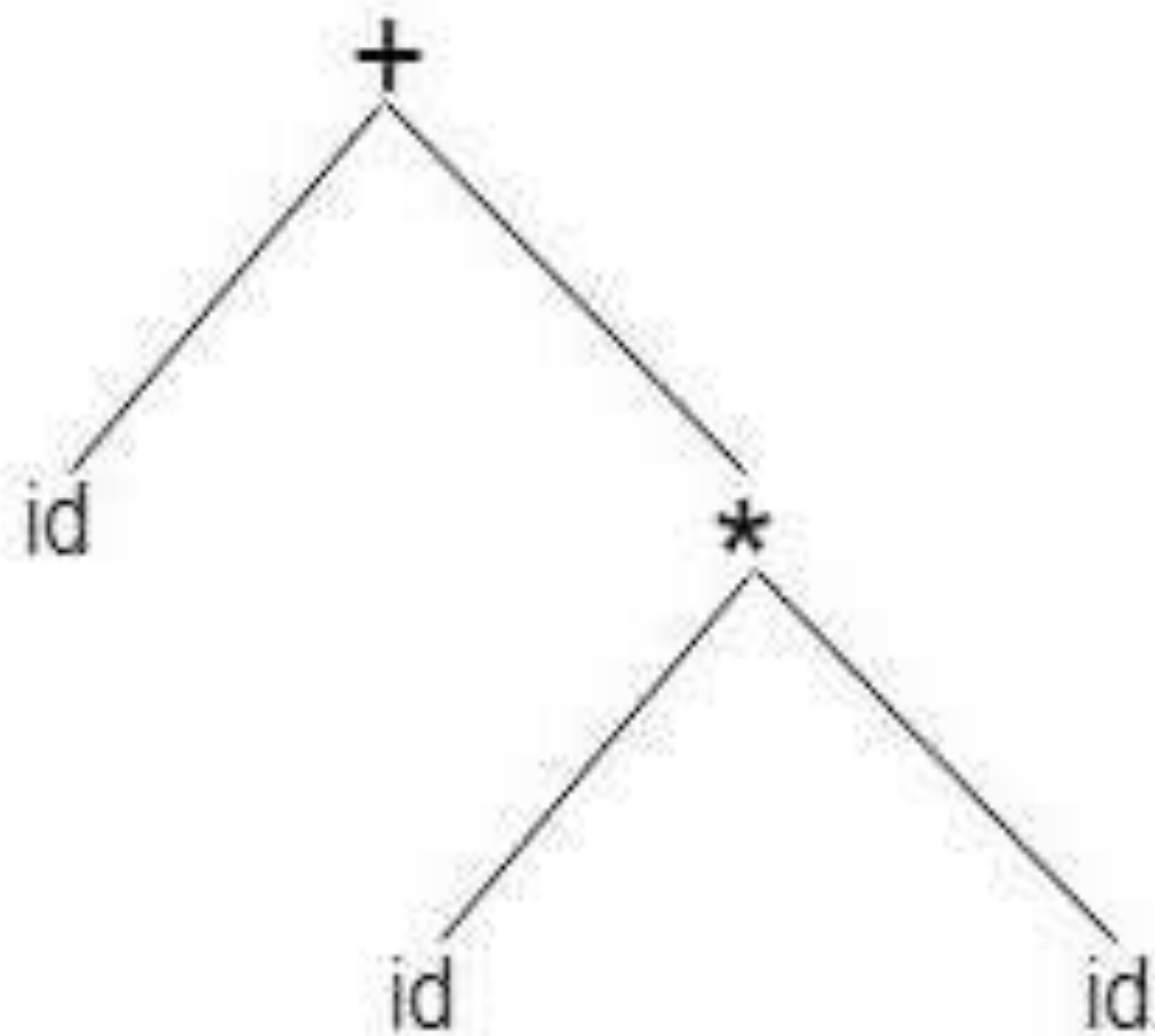
- Syntax tree
- Postfix Notation
- Three-Address Code

Clip slide

Syntax tree

Syntax tree is nothing more than condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

A sentence **id + id * id** would have the following syntax tree:



Postfix Notation

- The ordinary (infix) way of writing the sum of a and b is with operator in the middle : a + b
- The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation the operator follows the operand.

Postfix Notation

← prev
next →

- Postfix notation is the useful form of intermediate code if the given language is expressions.
- Postfix notation is also called as 'suffix notation' and 'reverse polish'.
- Postfix notation is a linear representation of a syntax tree.
- In the postfix notation, any expression can be written unambiguously without parentheses.
- The ordinary (infix) way of writing the sum of x and y is with operator in the middle: x * y. But in the postfix notation, we place the operator at the right end as xy *.
- In postfix notation, the operator follows the operand.

Three-Address Code

Clip slide

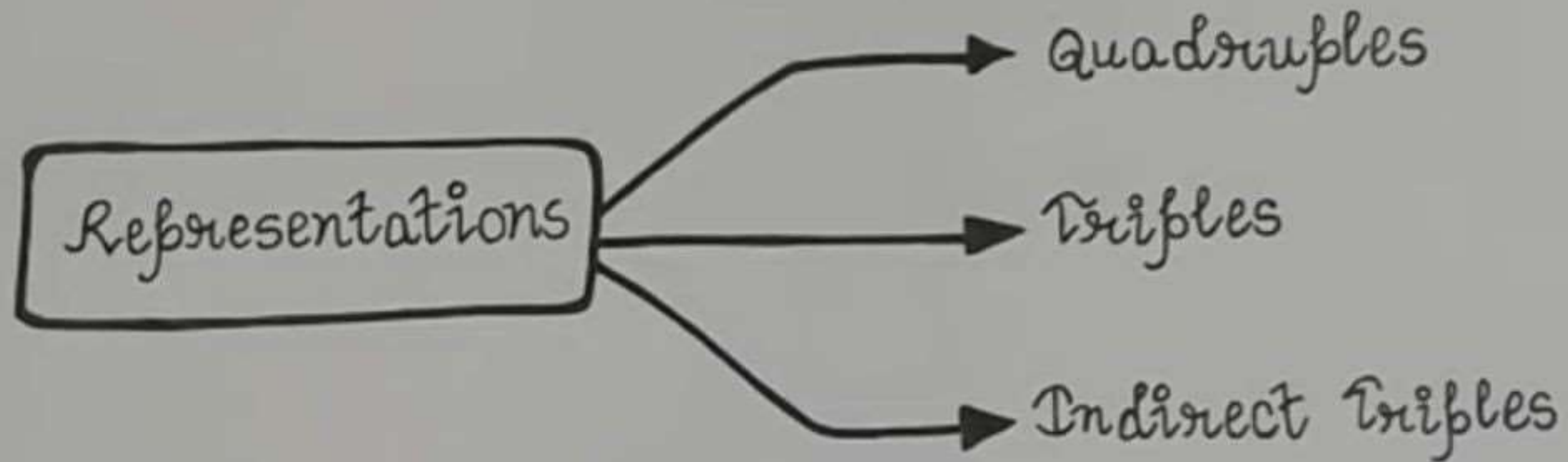
A statement involving no more than three references(two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z , here x, y, z will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statement.

For Example :a = b + c * d;

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;
r2 = b + r1;
a = r2
```

Three address code can be implemented as a record with the address fields. There are three representations used for three address code -

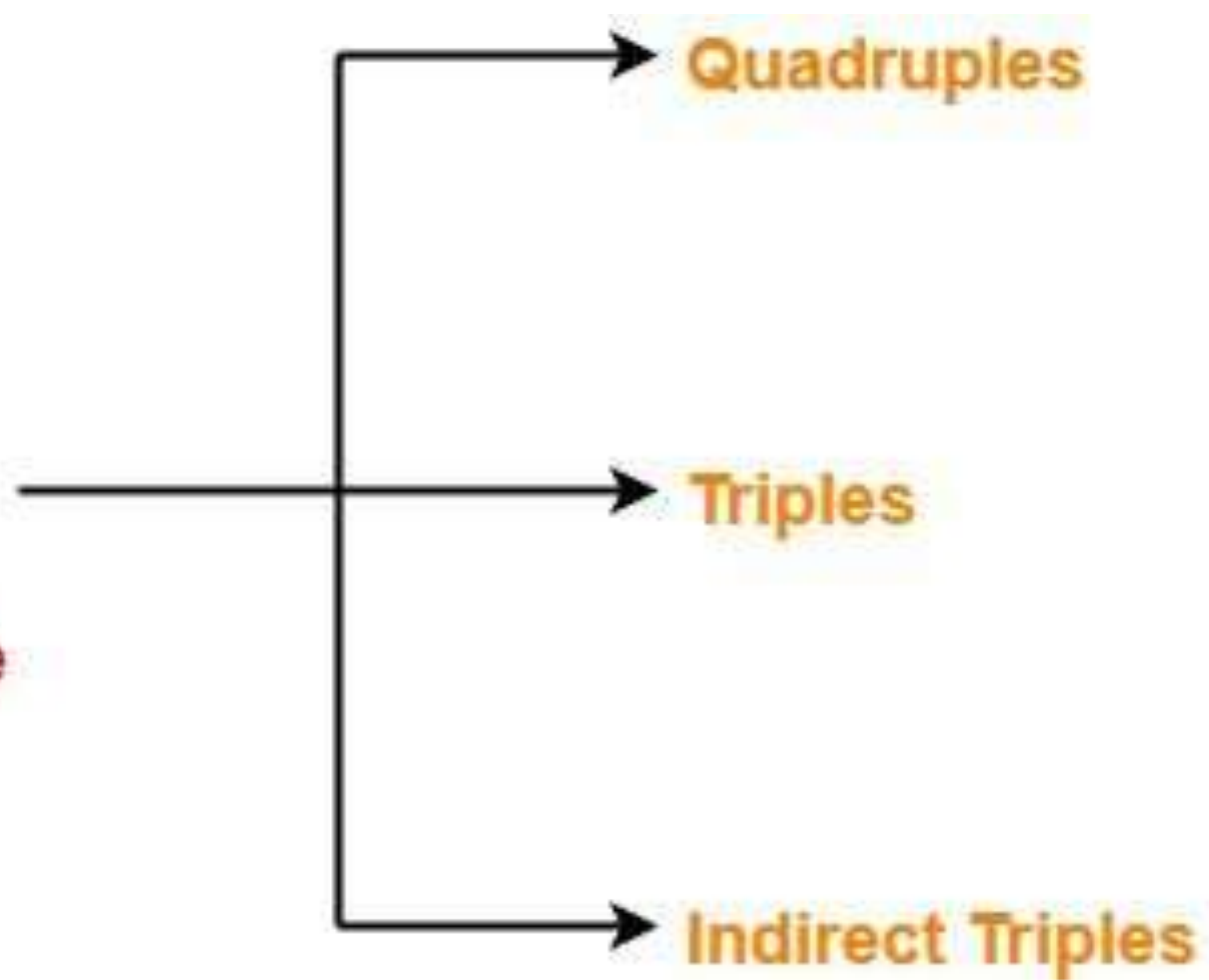


Learn Vid Fun!

Three-Address Code

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms :

- Quadruples
- Triples
- Indirect Triples


Representations
for
Three Address Code
Quadruples
Triples
Indirect Triples

Three address codes Implementations

Clip slide

- Quadruples
  - Has four fields: op, arg1, arg2 and result
- Triples
  - Temporaries are not used and instead references to instructions are made
- Indirect triples
  - In addition to triples we use a list of pointers to triples

1) Quadruples

In quadruple representation, each instruction is divided into four fields -

op , arg1 , arg2 , result

where -

→ The op field is used to represent the internal code for operator

→ The arg1 and arg2 fields represent the two operands used

→ The result field is used to store the result of an expression.

Learn Vid Fun!

2> Triples :-

In triple representation, the use of temporary variables is avoided and instead references to instructions are made.

3> Indirect Triples :-

This representation is an enhancement over triples representation. It uses an additional instruction array to list the pointers to the triples in the desired order. Thus, it uses pointers instead of position to store results which enables the optimizers to freely reposition the sub-expression to produce an optimized code.

Learn Vid Fun!

Three-address Code, Quadruples, Triples and Indirect Triples

Consider the following assignment statement like

$$A := -B * (C / D)$$

We can express this expression using

1. Three-address codes
2. Quadruples
3. Triples
4. Indirect Triples

as follows -

Three Address Code

$T_1 := -B$

$T_2 := C / D$

$T_3 := T_1 * T_2$

$A := T_3$

(1)

A= -B*(C/D)

QUADRAPLES

	OP	ARG1	ARG2	RESULT
(0)	uminus	B	-	T1
(1)	/	C	D	T2
(2)	*	T1	T2	T3
(3)	:=	T3	-	A

(2)

TRIPLES

	OP	ARG1	ARG2
(0)	uminus	B	-
(1)	/	C	D
(2)	*	(0)	(1)
(3)	:=	A	(2)

(3)

INDIRECT TRIPLES

	STATEMENT
(0)	(21)
(1)	(22)
(2)	(23)
(3)	(24)

	OP	ARG1	ARG2
(21)	uminus	B	–
(22)	/	C	D
(23)	*	(21)	(22)
(24)	:=	A	(23)

4

DAG

Definitions

- In compiler design, a directed acyclic graph (DAG) is on abstract syntax tree(AST) with a unique node for each value.

OR

- A directed acyclic graph(DAG) is a directed graph that contains no cycles.

Introduction:

- Directed Acyclic Graphs (dags) are useful data structures for implementing transformations on basic blocks.
- A dag gives a picture of how the value computed by each statement in a basic block is used in subsequent statements of the block.
- Constructing a dag from three- address statements is a good way of determining common sub expressions within a block, determining which names are used inside the block but evaluate outside the block, and determining which statements of the block have their computed value outside the block.

Use of DAG for optimizing basic blocks

Clip slide

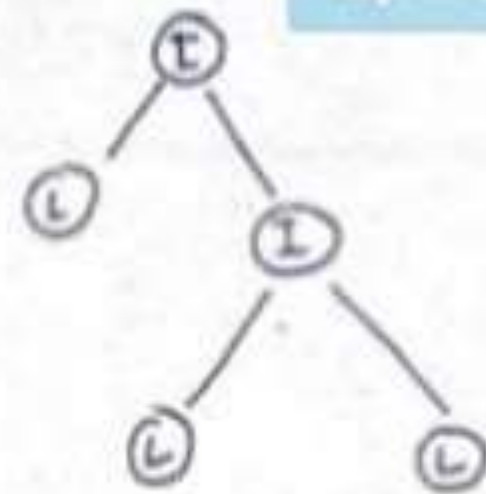
- DAG is useful data structure for implementing transformation on basic blocks.
- A basic block can optimized by the construction of DAG.
- A DAG can be constructed for a block and certain transformations such as common sub-expression elimination and dead code elimination applied for performance the local optimization.
- To apply the transformation on basic block, a DAG is constructed from three address code.

Applications of dags:

1. Automatically detect common sub expressions.
2. Can determine which identifiers have their values used in the block.
3. Can determine which statements compute value that could be used outside the block.

Rules of the constructing DAG

Clip slide



Rule 1: In a DAG

- Leaf node represent identifiers, names or constants.
- Interior node represent operators.

Rule 2:

- While constructing DAG, there is a check made to find if there is on existing node with the same children. A new node is created only when such a node doesn't exist This action allow us to detect common sub expression and eliminate the re computation of the same.

Problems

Clip slide

- **Problem 1:**

Construct DAG for the given expression

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

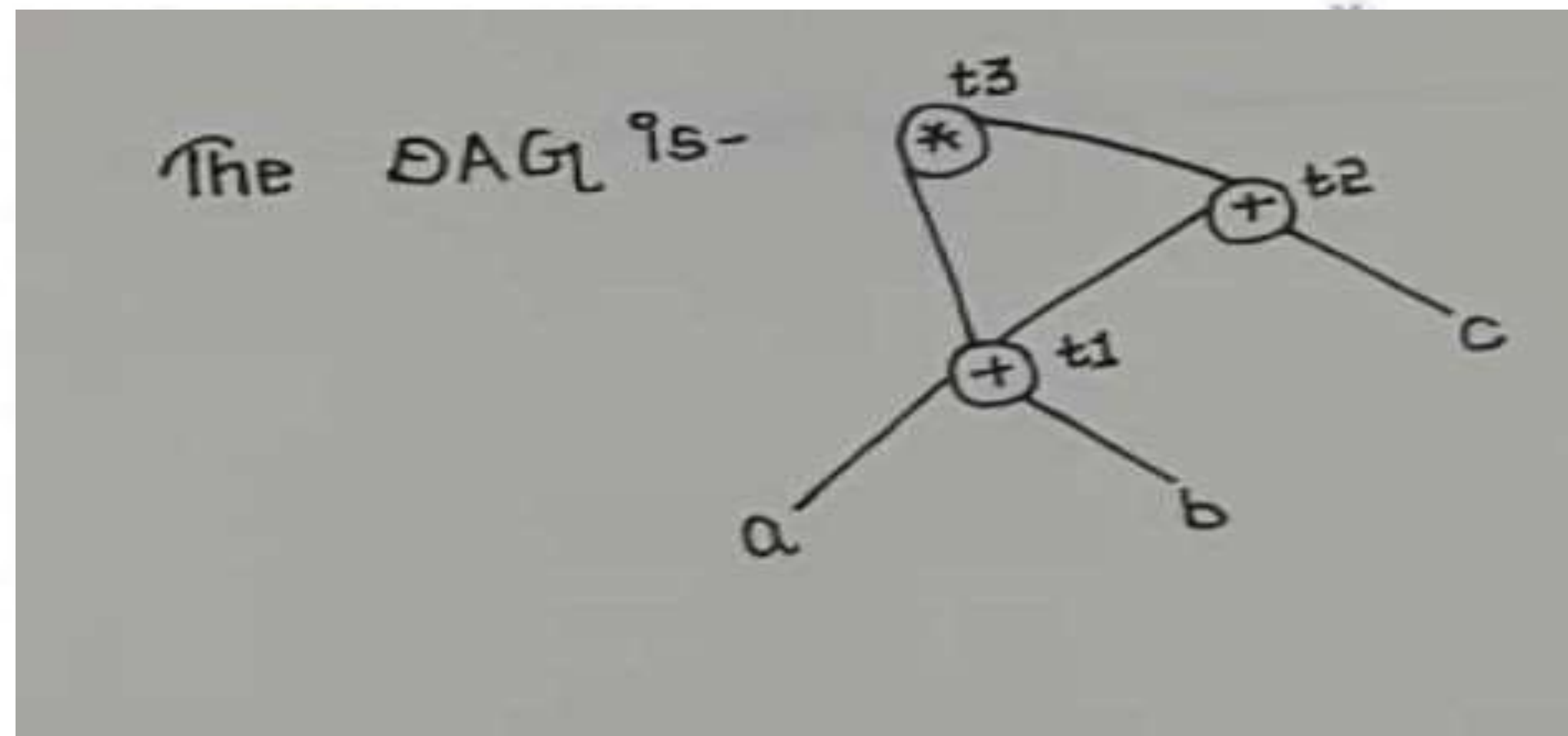
Solution: Three address code for the given expression.

t1=a+b

t2=t1+c

t3=t1*t2

The DAG is:



Explanation

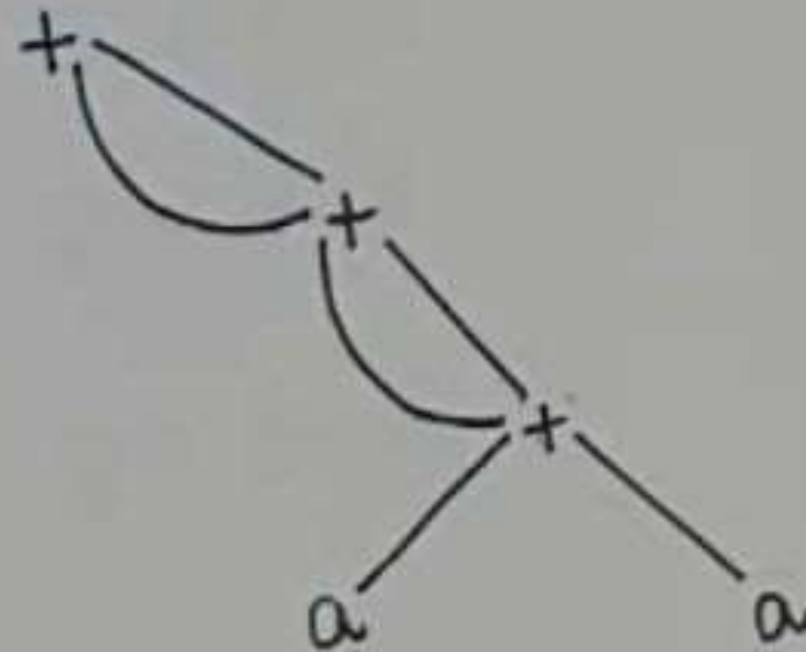
- From the constructed DAG, we observed that the common sub expression (a+b) is translated into a single node in the DAG. The computation is carried out only once and stored in the identifier t1 and reused later.
- This illustrates how the DAG construction scheme identifies the common sub expression and helps in elimination its re-computation later.

- **Problem 2:**

Construct DAG for the given expression

$$(((a+a) + (a+a)) + ((a+a) + (a+a)))$$

Solution: DAG for this Expression is-



Clip slide

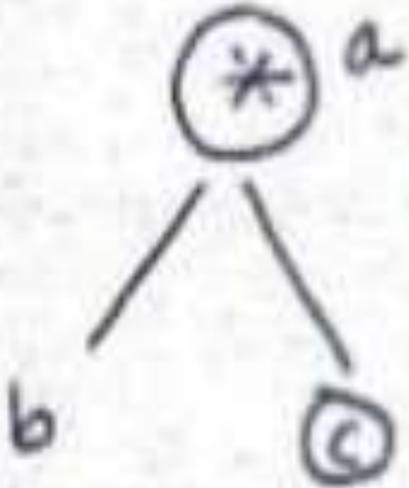
Problem 3:

Construct DAG for the given expression

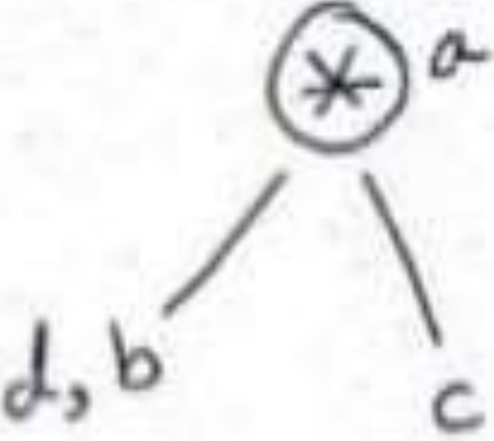
a = b * c
d = b
e = d * c
b = e
f = b + c
g = f + d

Solution:

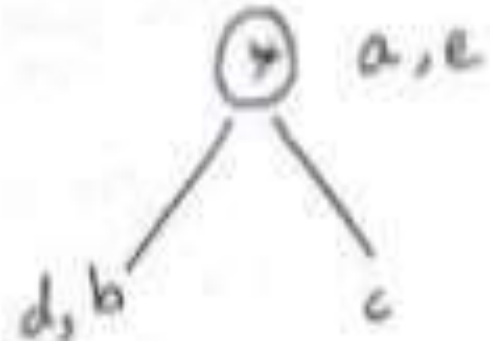
Step 1:



Step 2:

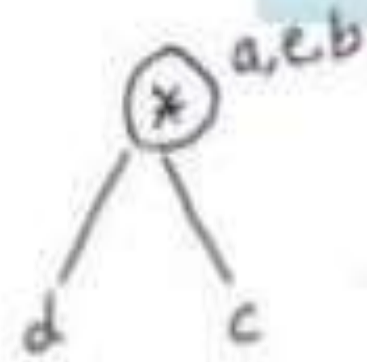


Step 3:

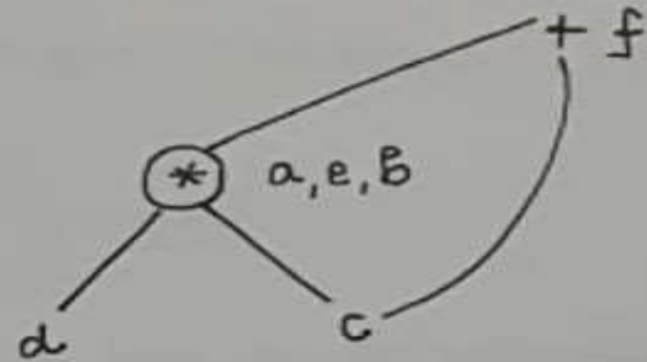


Step 4:

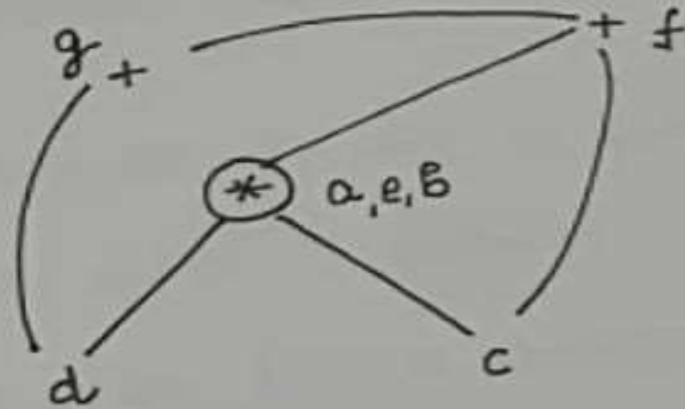
Clip slide



Step-5 :-



Step-6 :-



Boolean Expressions

- Can be used for Computing the logical values.
- Can also be used as conditional expression for
  - if-then-else statement
  - while-do loops

Consider a grammar

E → E or E
E → E and E
E → not E
E → (E)
E → id relop id
E → true
E → false

- The relop is denoted by **<, <=, >, >=, =, <>**
- Here **or** and **and** are left associative
- **not** has the higher precedence then **and** and lastly **or**

Methods of Translating Boolean Expressions

- **First Method: Numerical Representation**
  - To encode true and false values numerically
  - Evaluate Boolean expression like arithmetic expression
- **Second Method: Flow of control**
  - Represent Boolean expression by a position reached
  - Convenient for if-then and loop statements

Numerical Representation

- **Example:**

  a or b and not c

- **Three address code**

  t1 := not c

  t2 := b and t1

  t3 := a or t2

- **Example:**

A relational expression a<b is considered as:

if a<b then 1 else 0

```
100: if a<b goto 103
101: t:=0
102: goto 104
103: t:=1
104:
```

Translation scheme for Booleans (Numerical Representation)

Production rule	Semantic actions
E → E1 or E2	{E.place = newtemp(); emit (E.place ':=' E1.place 'or' E2.place)}
E → E1 and E2	{E.place = newtemp(); emit (E.place ':=' E1.place 'and' E2.place) }
E → not E1	{E.place = newtemp(); emit (E.place ':=' 'not' E1.place)}
E → (E1)	{E.place = E1.place}
E → id1 relop id2	{E.place = newtemp(); emit ('if' id1.place relop.op id2.place 'goto' nextstat +3); emit (E.place ':=' '0'); emit ('goto' nextstat + 2); emit (E.place ':=' '1') }
E → true	{E.place := newtemp(); emit (E.place ':=' '1') }
E → false	{E.place := newtemp(); emit (E.place ':=' '0') }

Syntax-Directed Definition for Assignment Statement

- **emit()** - function is used to generate the three address code
- **newtemp()** - function is used to generate the temporary variables
- **The E → id relop id2**
  - contains the nextstate
  - it gives the index of next three address statements in the output sequence

E → id relop id2	{E.place = newtemp(); emit ('if' id1.place relop.op id2.place 'goto' nextstat +3); emit (E.place ':=' '0'); emit ('goto' nextstat + 2); emit (E.place ':=' '1') }

- if a<b then 1 else 0

```
100: if a<b goto 103
101: t:=0
102: goto 104
103: t:=1
104:
```

E → id relop id2	{E.place = newtemp(); emit ('if' id1.place relop.op id2.place 'goto' nextstat +3); emit (E.place ':=' '0'); emit ('goto' nextstat + 2); emit (E.place ':=' '1') }

- if a<b then 1 else 0

  100: if a<b goto 103

  101: t:=0

  102: goto 104

  103: t:=1

  104:

Consider nextstat=100

E.Place = t

temporary variable

E → id relop id2 {E.place = newtemp();
emit ('if' id1.place relop.op id2.place 'goto' nextstat +3);
emit (E.place ':=' '0'); emit ('goto' nextstat + 2); emit (E.place ':=' '1') }

- if a<b then 1 else 0

  100: if a<b goto 103

  101: t:=0

  102: goto 104

  103: t:=1

  104:

Consider nextstat=100

id1.place = a

id2.place = b

relop.op = <

nextstat =100+3=103

E → id relop id2	{E.place = newtemp(); emit ('if' id1.place relop.op id2.place 'goto' nextstat +3); emit (E.place ':=' '0'); emit ('goto' nextstat + 2); emit (E.place ':=' '1') }

- if a<b then 1 else 0

  100: if a<b goto 103

  101: t:=0

  102: goto 104

  103: t:=1

  104:

E.Place := 0

t := 0

E → id relop id2	{E.place = newtemp(); emit ('if' id1.place relop.op id2.place 'goto' nextstat +3); emit (E.place ':=' '0'); emit ('goto' nextstat + 2); emit (E.place ':=' '1') }

- if a<b then 1 else 0

  100: if a<b goto 103

  101: t:=0

  102: goto 104

  103: t:=1

  104:

nextstat=102

nextstat+2=102+2=104

goto 104

E → id relop id2	{E.place = newtemp(); emit ('if' id1.place relop.op id2.place 'goto' nextstat +3); emit (E.place ':=' '0'); emit ('goto' nextstat + 2); emit (E.place ':=' '1') }

- if a<b then 1 else 0

  100: if a<b goto 103

  101: t:=0

  102: goto 104

  103: t:=1

  104:

E.Place = t

t := 1

Three Address Code generation for Boolean Expressions (for Flow of control statements)

Boolean Expressions in flow of control statements

- Boolean expressions are used as conditional expressions in statements that alter the flow of control
- The value of such Boolean expressions is implicit in a position reached in a program.
- For example:

in the statement if (E) S

the expression E must be true if statement S is reached.

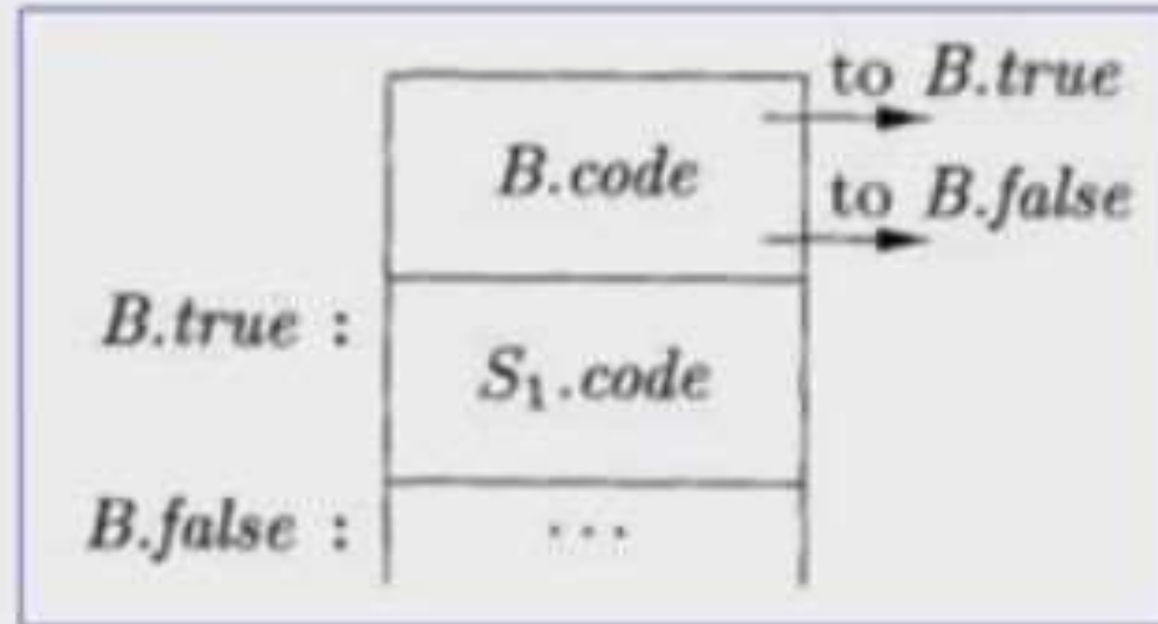
Grammar for flow of control statements

If-then
If-then-else
While-do

$$
\begin{aligned}
S &\rightarrow \text{if } (B)\ S1 \\
&\ \mid \text{if } (B)\ S1 \text{ else } S2 \\
&\ \mid \text{while } (B) \text{ do } S1
\end{aligned}
$$

- Here B is the Boolean Expression
- S1,S2 are the programming Statements

S→ if (B) S1



Code for the control flow statements

Syntax Directed Definition

$S \rightarrow \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$

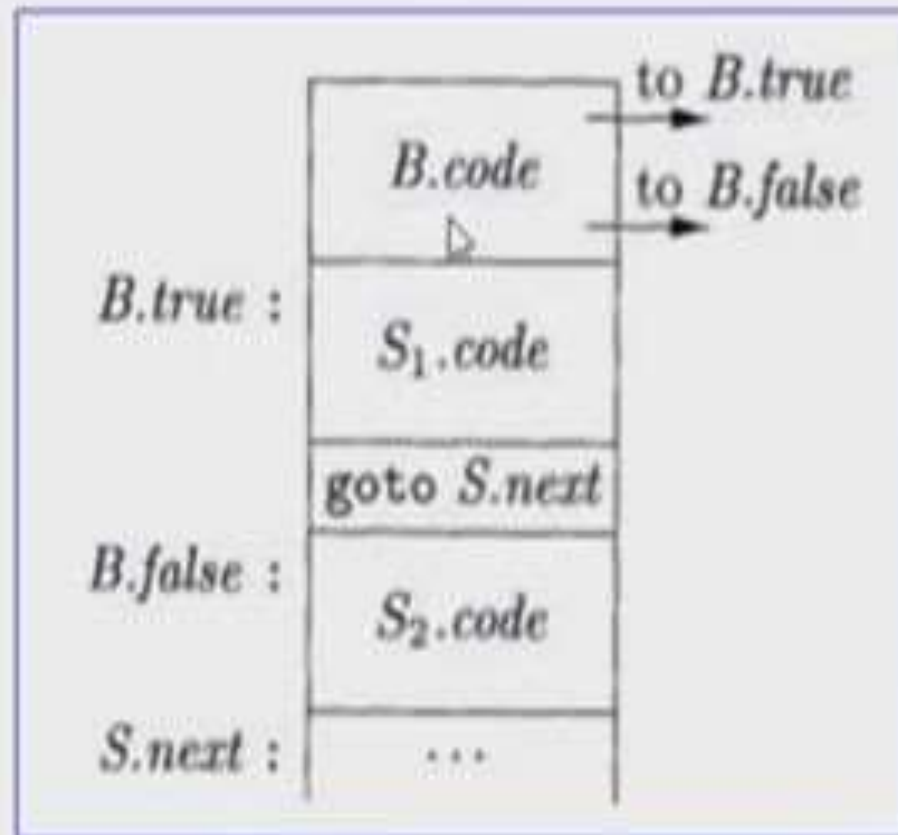
Example

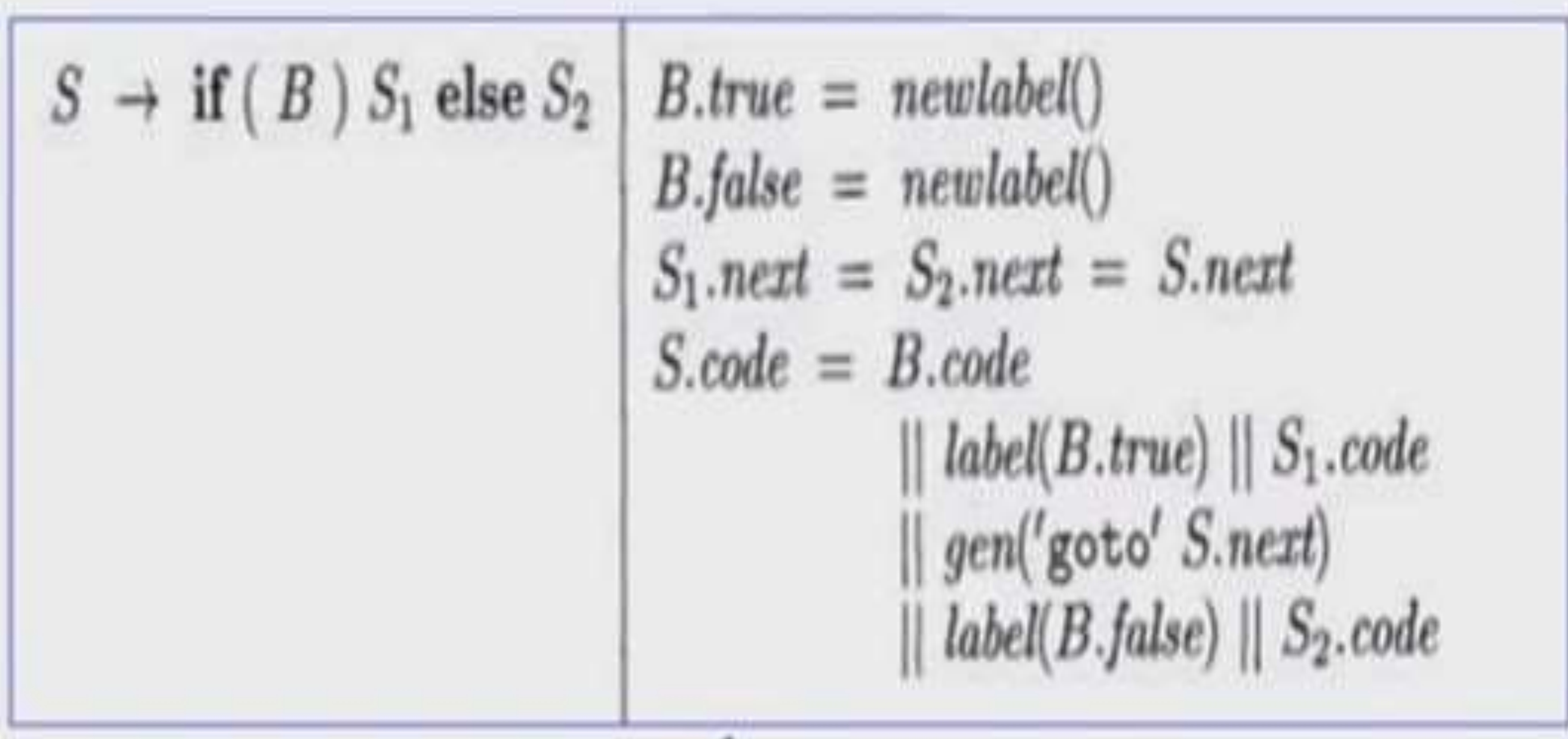
1. If (a < b) S1

- Assume that the attributes true and false exist for the entire expression as labels Ltrue and Lfalse respectively.
- Then the translation is

```
          if a < b goto Ltrue
          goto Lfalse
Ltrue:  S1.code
Lfalse:
```

S→ if (B) S1 else S2





Code for the control flow statements

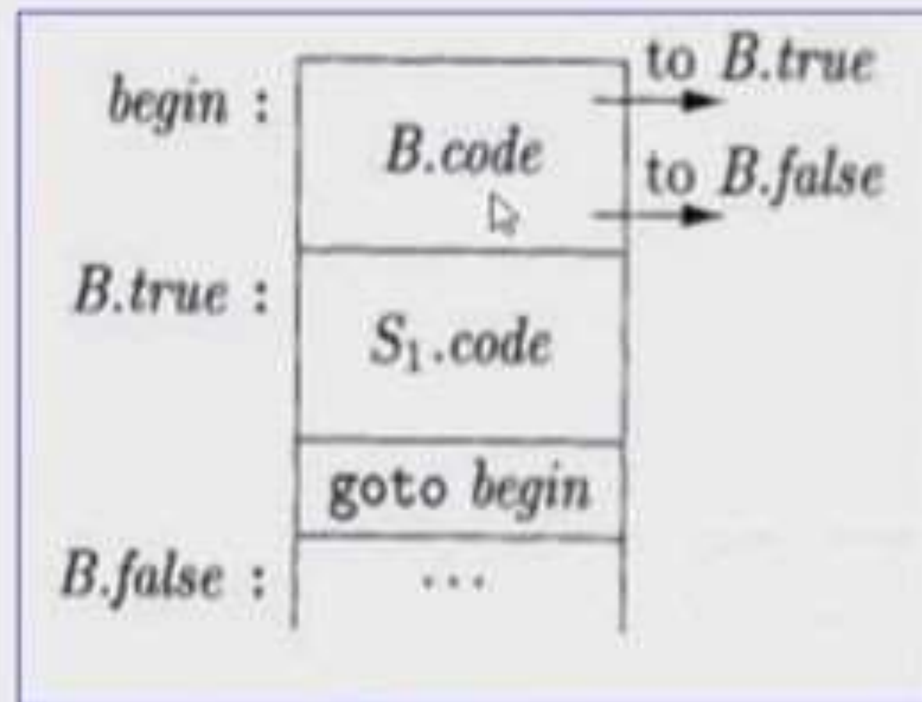
Syntax Directed Definition

Example

2. If (a < b or c < d) S1 else S2

```
        if a < b goto Ltrue
        goto L1
    L1: if c < d goto Ltrue
        goto Lfalse
Ltrue:  S1.code
        goto Snext
Lfalse: S2.code
Snext:
```

S→ while (B) do S1



Production	Semantic Rules
$S \rightarrow \textbf{while} (B) S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \parallel B.code \parallel label(B.true) \parallel S_1.code \parallel gen('goto'\ begin)$

Code for the control flow statements

Syntax Directed Definition

Example

```
while a < b do
    if c < d then
        x := y + z
    else
        x := y – z
```

```
L1: if a < b goto L2
    goto Lnext
L2: if c < d goto L3
    goto L4
L3: t1 := y + z
    x := t1
    goto L1
L4: t2:= y – z
    x:=t2
    gotoL1
Lnext:
```

JECRC Foundation

JAIPUR ENGINEERING COLLEGE
AND RESEARCH CENTRE

Thank you!