Semantic Analysis

C.Naga Raju
B.Tech(CSE),M.Tech(CSE),PhD(CSE),MIEEE,MCSI,MISTE
Professor
Department of CSE
YSR Engineering College of YVU
Proddatur

Contents

Semantic analyzer

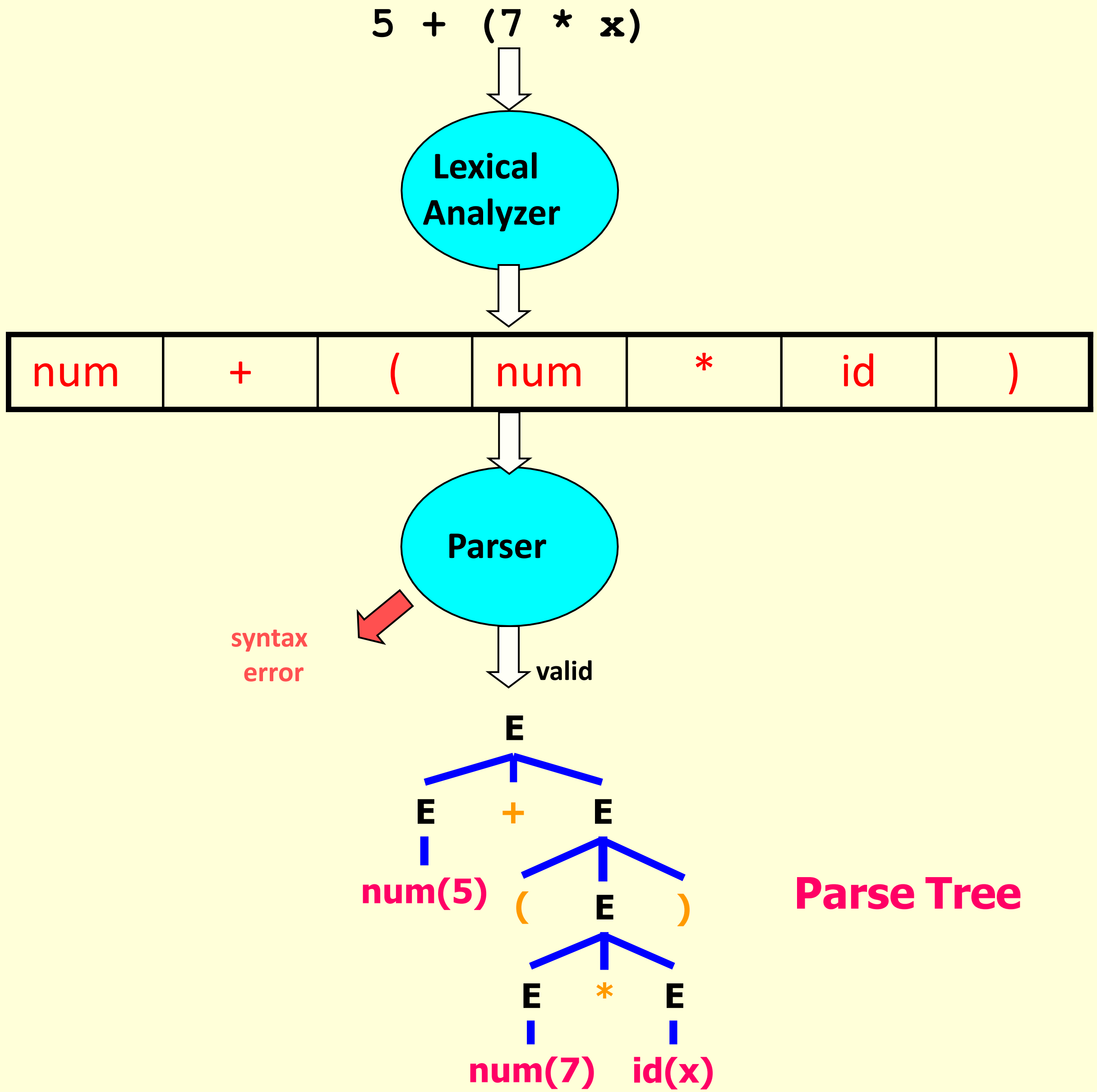
- It uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition or not.
- It gathers type information and stores it in either syntax tree or symbol table.
- Functions of Semantic Analysis:
- 1)Type Checking 2)Label Checking 3)Flow Control Check

Semantic errors

- Type mismatch
- Undeclared variables
- Reserved identifier misuse

Program Text

Token Stream



Parse Tree

Grammar:

E → id

E → num

E → E + E

E → E * E E

→ (E)

Grammar:

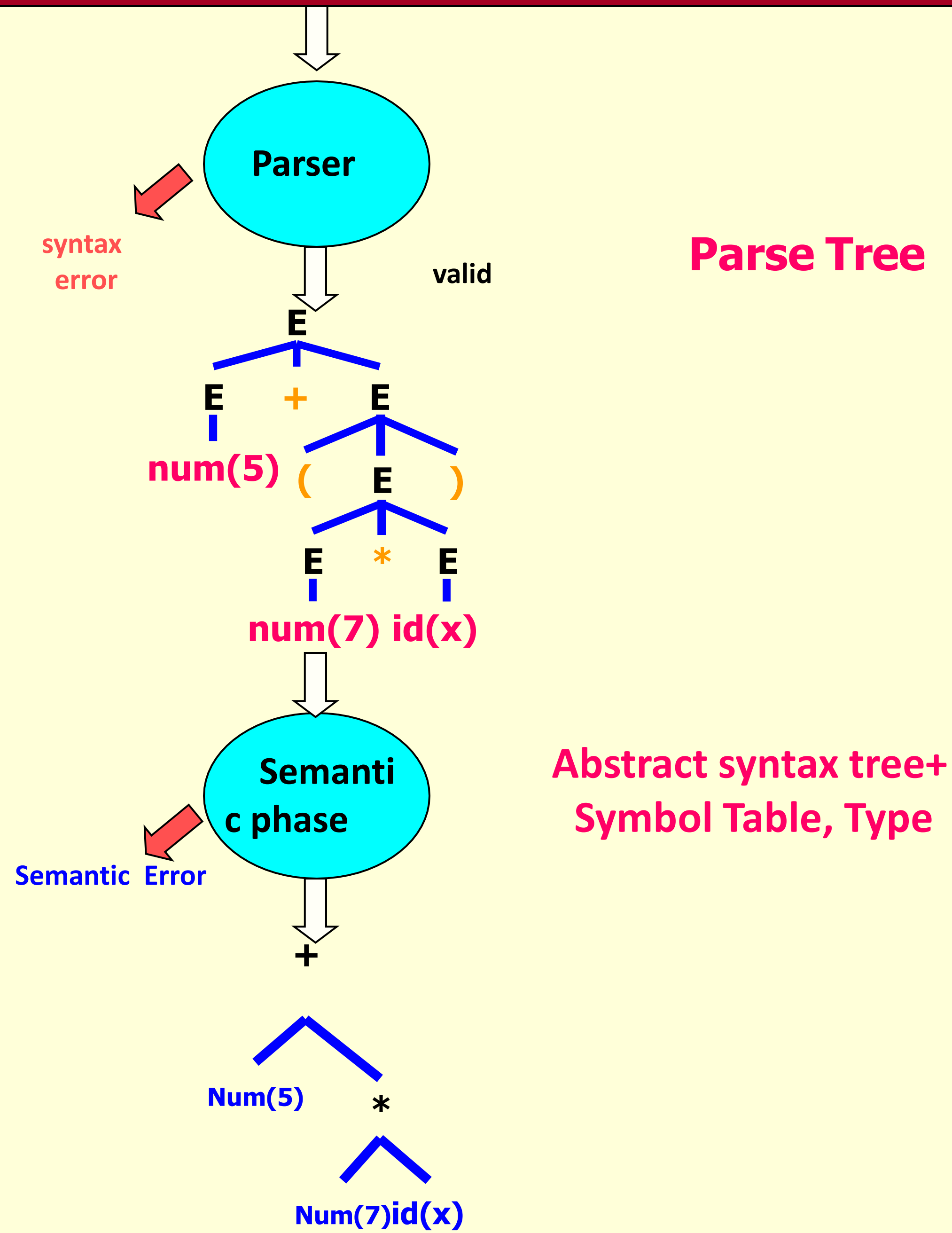
$E \rightarrow id$

$E \rightarrow num$

$E \rightarrow E + E$

$E \rightarrow E * E \quad E$

$\rightarrow (E)$



Syntax Directed Definitions

- Syntax Tree= Parse Tree +additional information.
- Additional information may be attributes ,rules actions etc
- A SDD is a context free grammar with attributes and rules
- Attributes are associated with grammar symbols and rules with productions
- Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions..
- Production Semantic Rule
- E->E1+T E.code=E1.code||T.code||'+'
- We may also insert the semantic actions inside the grammar
- E -> E1+T {print '+'}

Syntax Tree

- Syntax trees are abstract or compact representation of parse trees.
- Syntax trees are called as **Abstract Syntax Trees** because-
- They do not provide every characteristic information from the real syntax.
- For example- no rule nodes, no parenthesis etc.

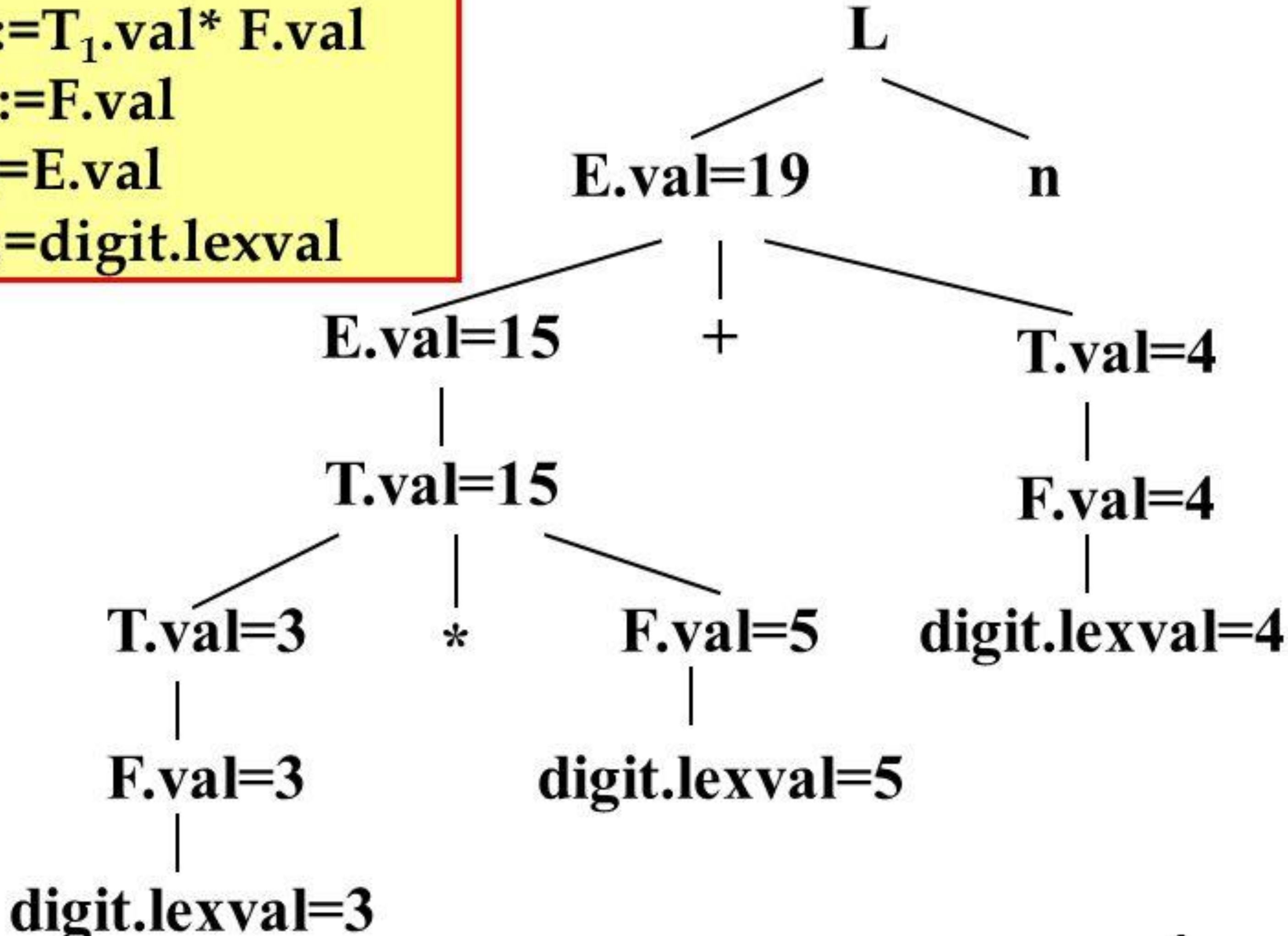
Parse Trees Vs Syntax Trees-

Parse Tree	Syntax Tree
Parse tree is a graphical representation of the replacement process in a derivation.	Syntax tree is the compact form of a parse tree.
Each interior node represents a grammar rule. Each leaf node represents a terminal.	Each interior node represents an operator. Each leaf node represents an operand.
Parse trees provide every characteristic information from the real syntax.	Syntax trees do not provide every characteristic information from the real syntax.
Parse trees are comparatively less dense than syntax trees.	Syntax trees are comparatively more dense than parse trees.

- "**Dependency graphs**" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree.
- While an annotated parse tree shows the values of attributes
  - A dependency graph helps us determine how those values can be computed.

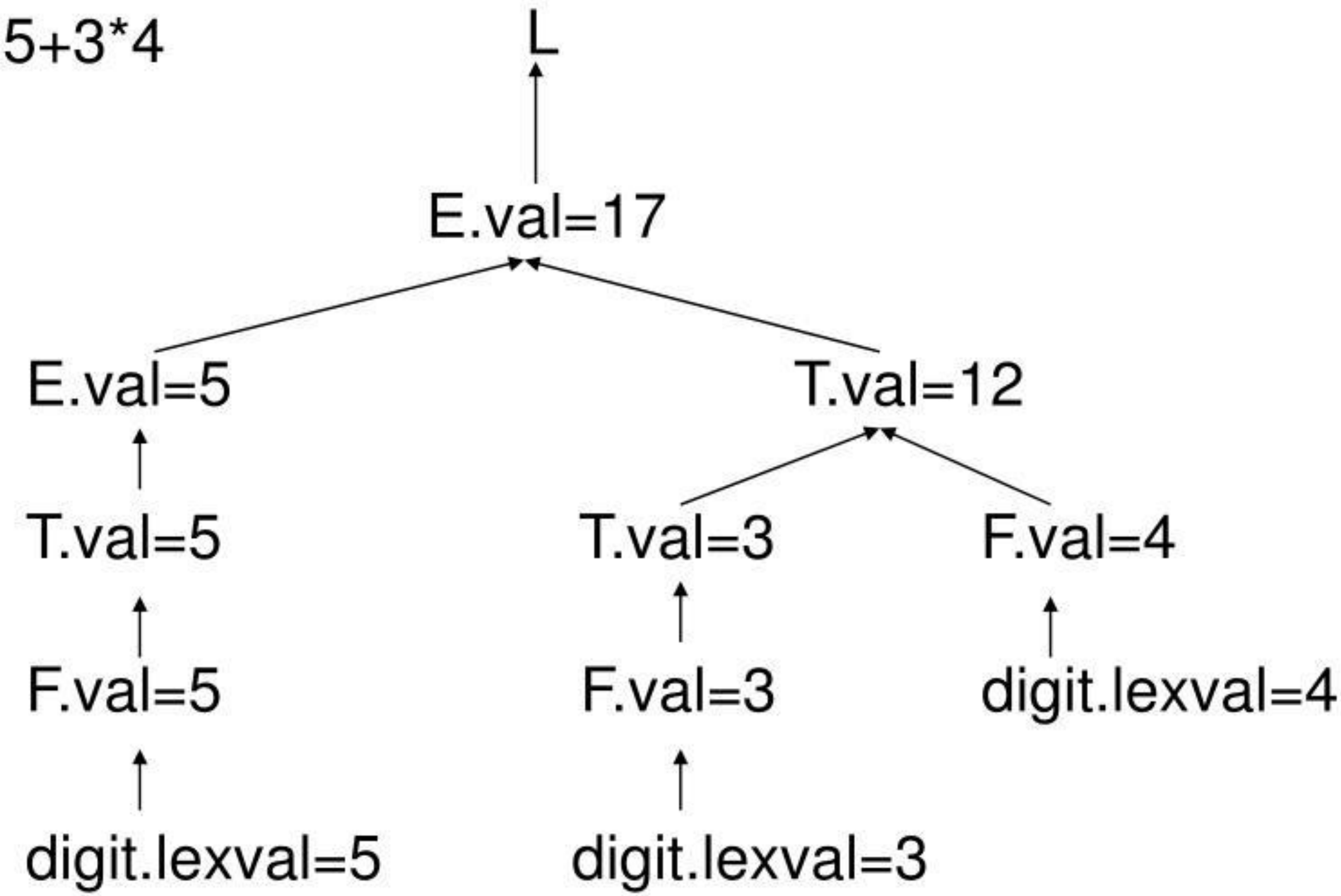
$L \rightarrow En$	print(E.val)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow digit$	$F.val := digit.lexval$

Annotated Parse Tree for 3*5+4 **n**

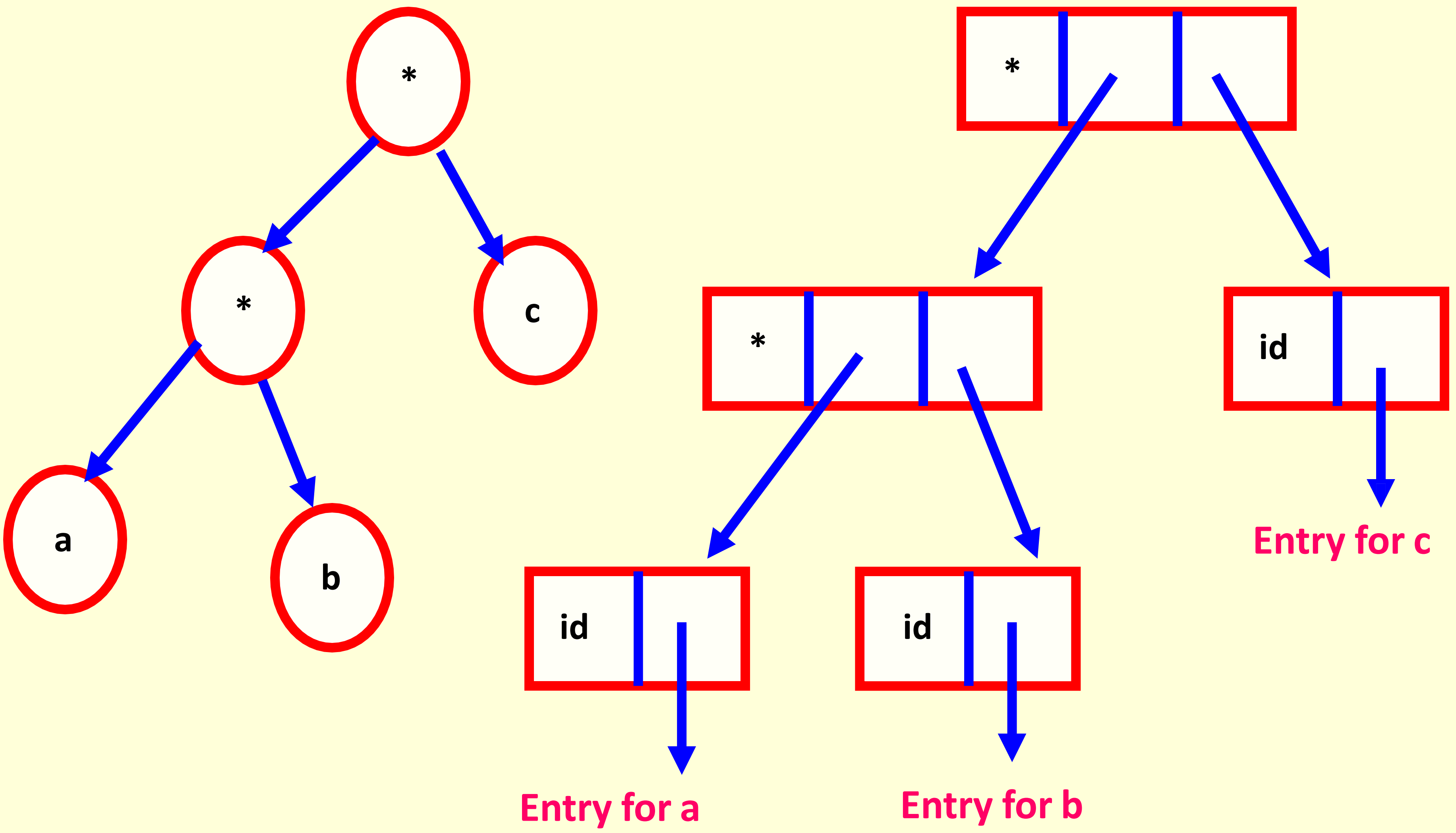


Dependency Graph Example

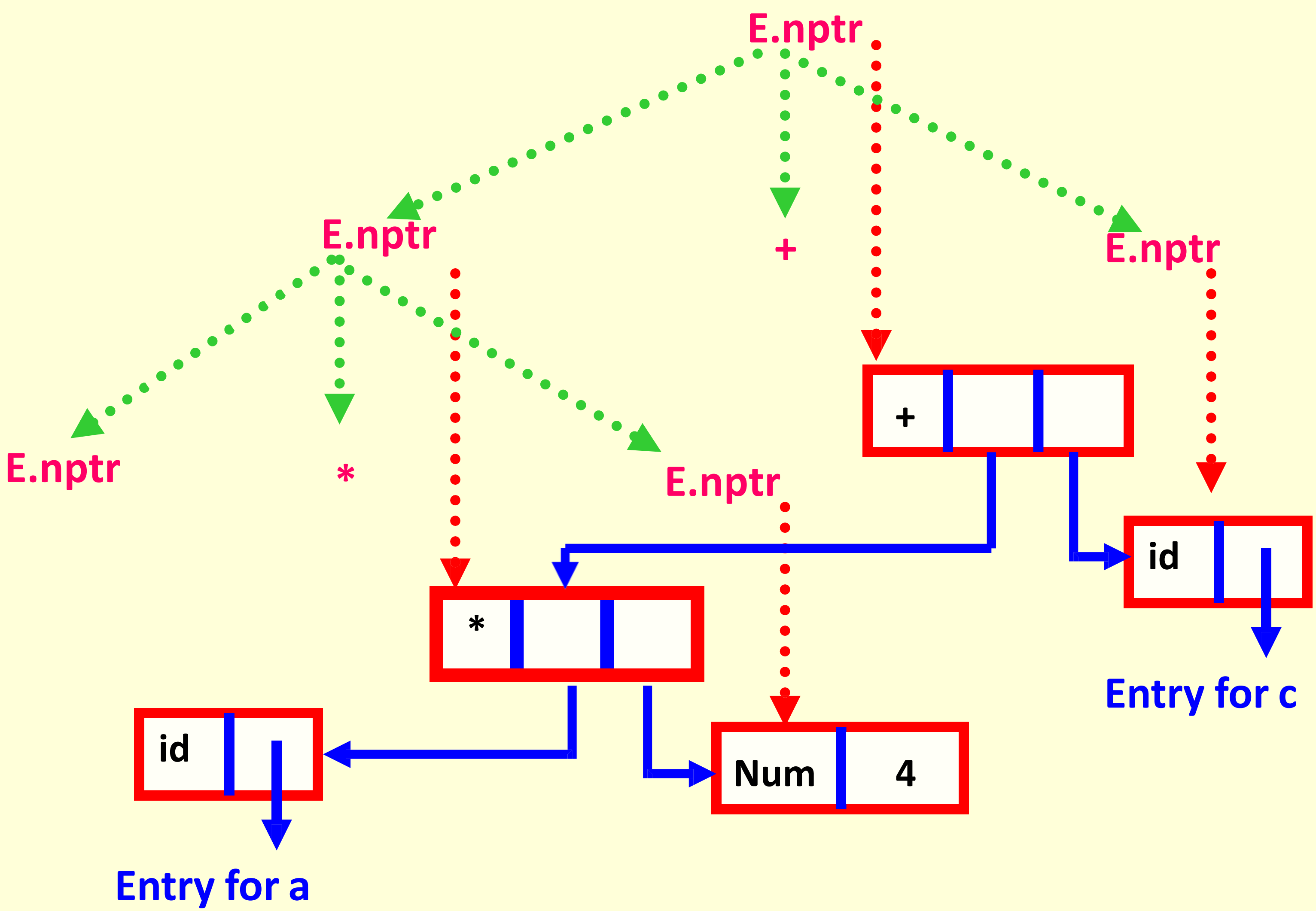
Input: 5+3*4



Constructing the Syntax Tree for Expression of Nodes (a*b*c)



Constructing the Syntax Tree for (a*4+c)

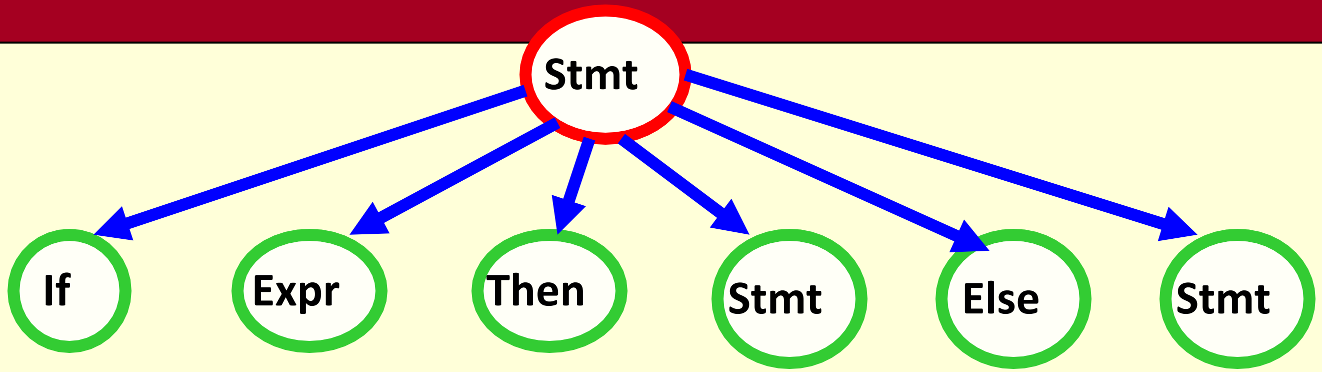


Example: Syntax Tree

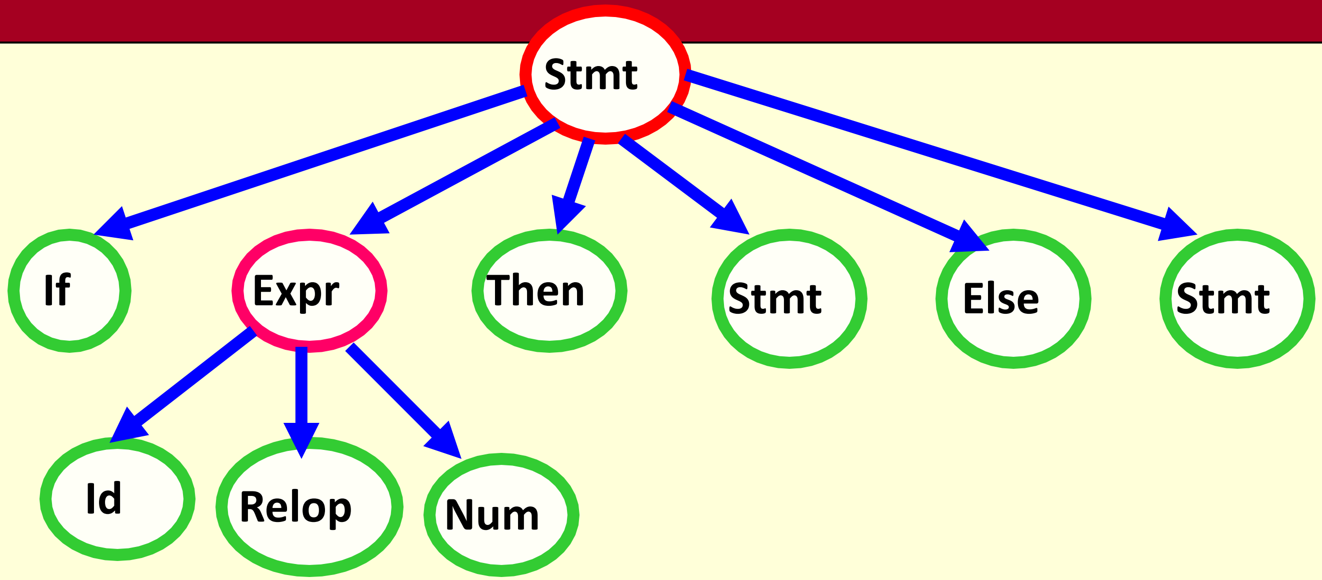
Suppose we have following code:

```
if ( x<0 )        then

x=3*(y+1);  else

y = y+1;
```

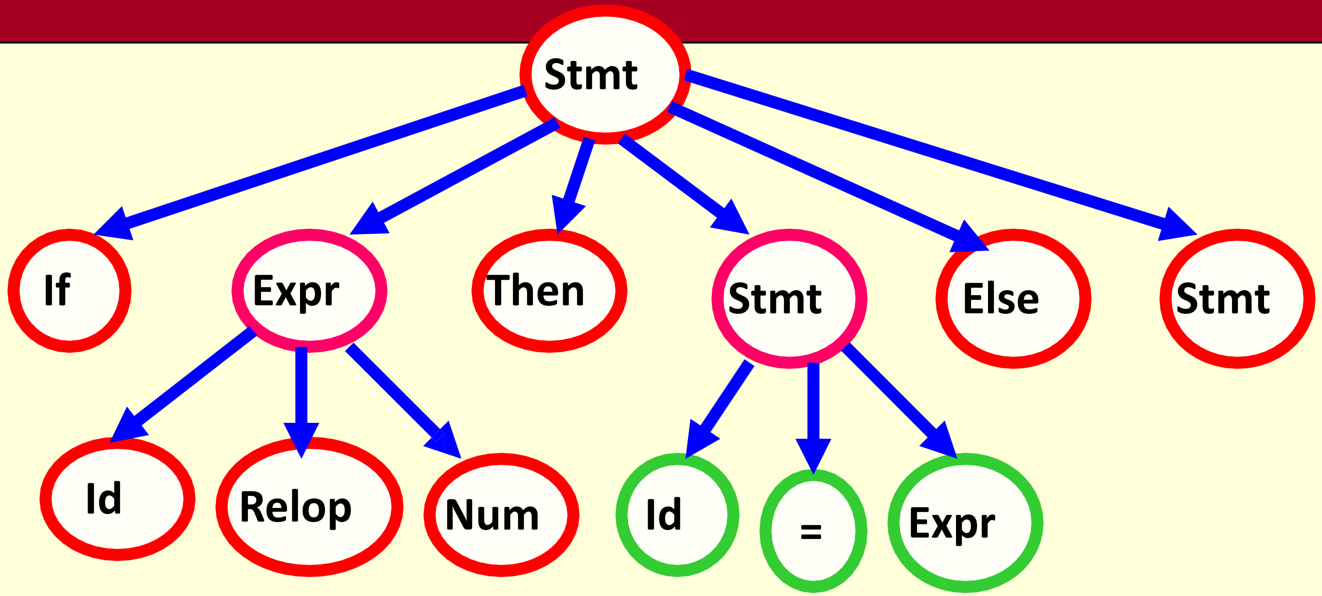
Example: Parse Tree



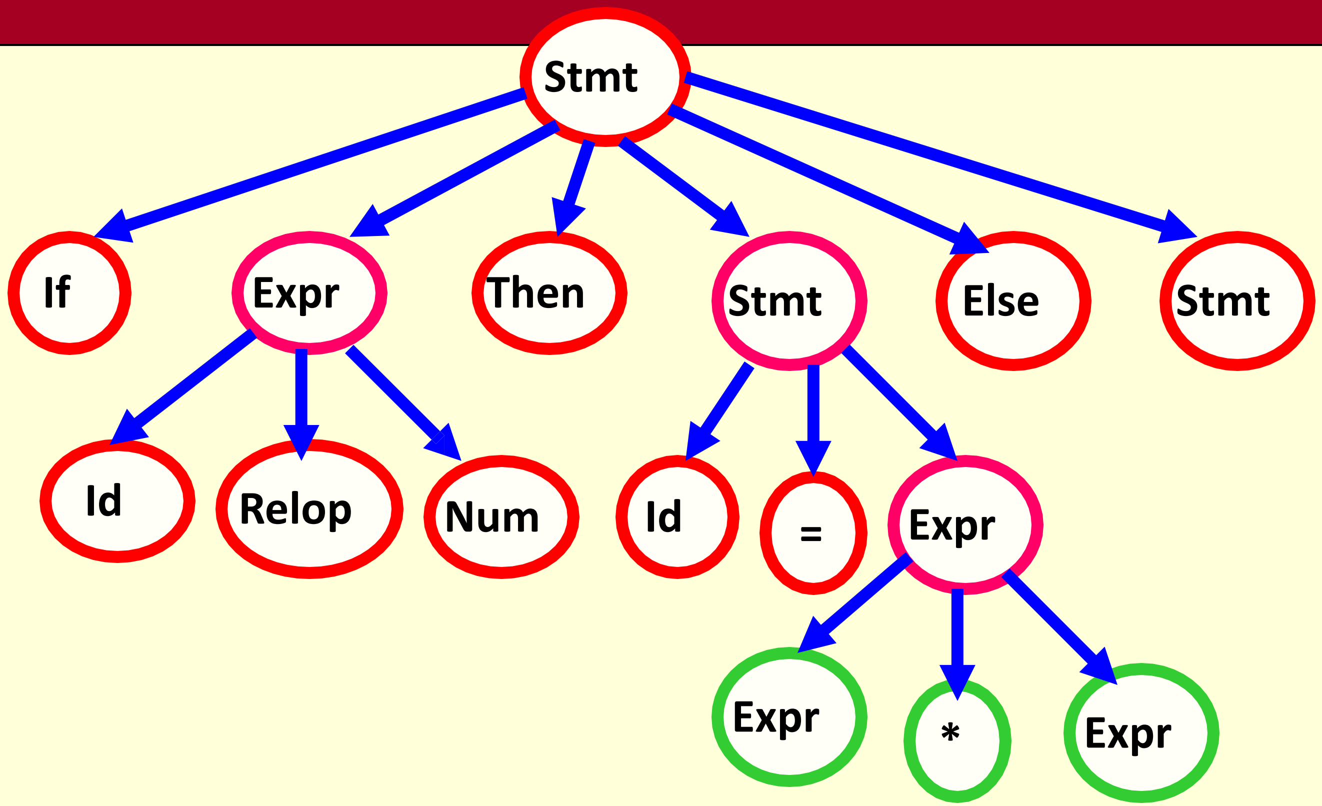
Example: Parse Tree



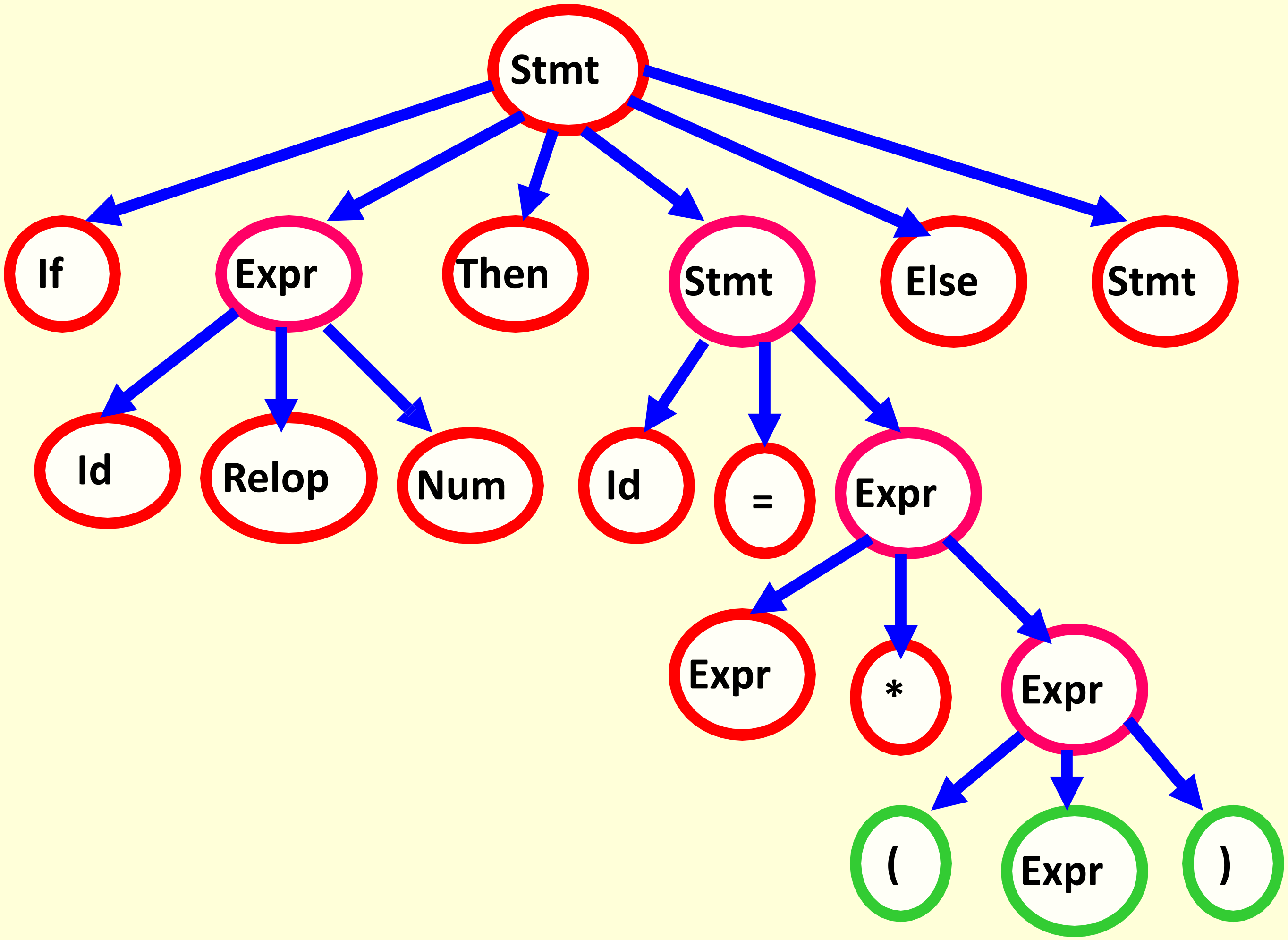
Example: Parse Tree



Example: Parse Tree

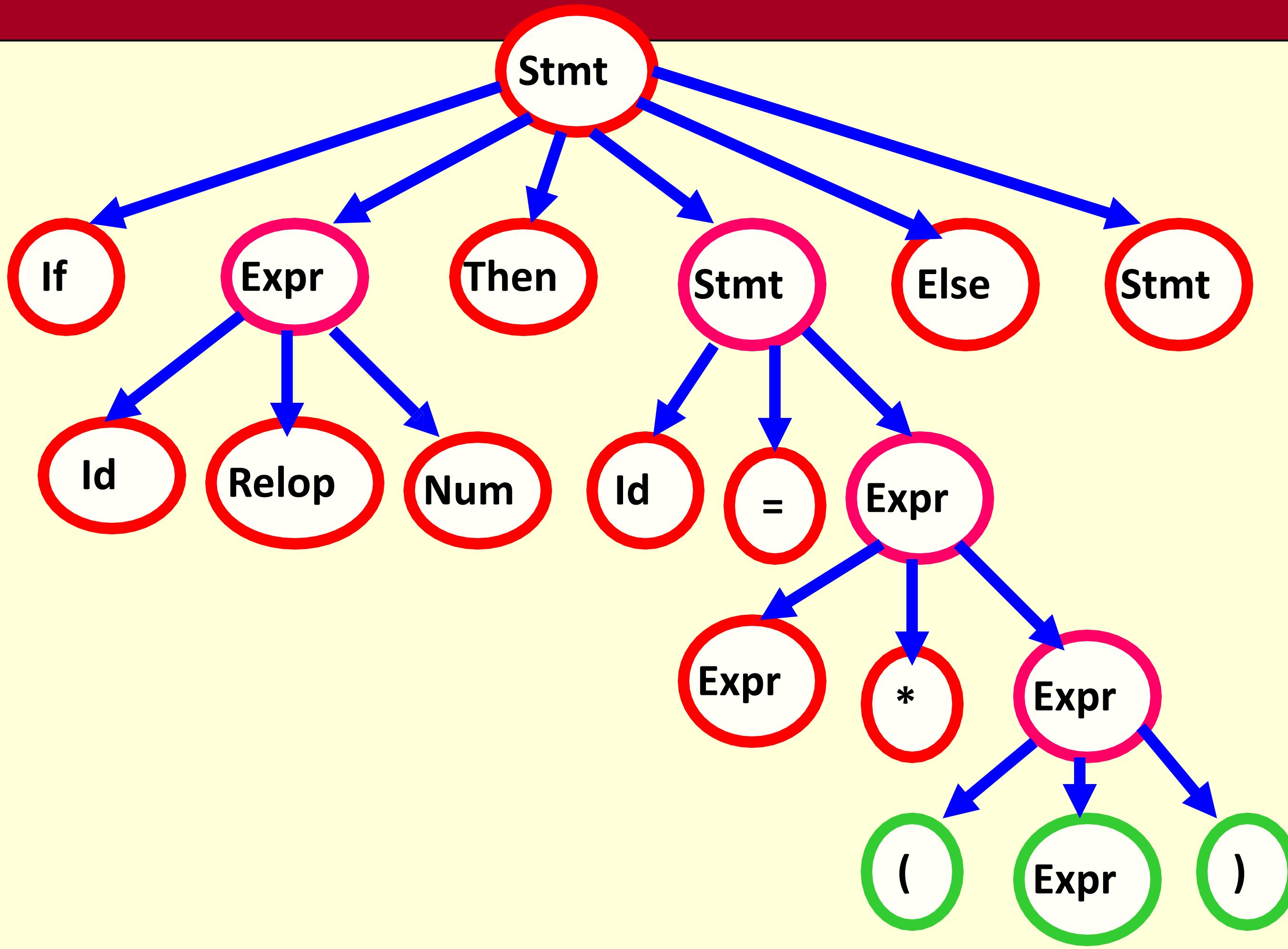


Parse Tree

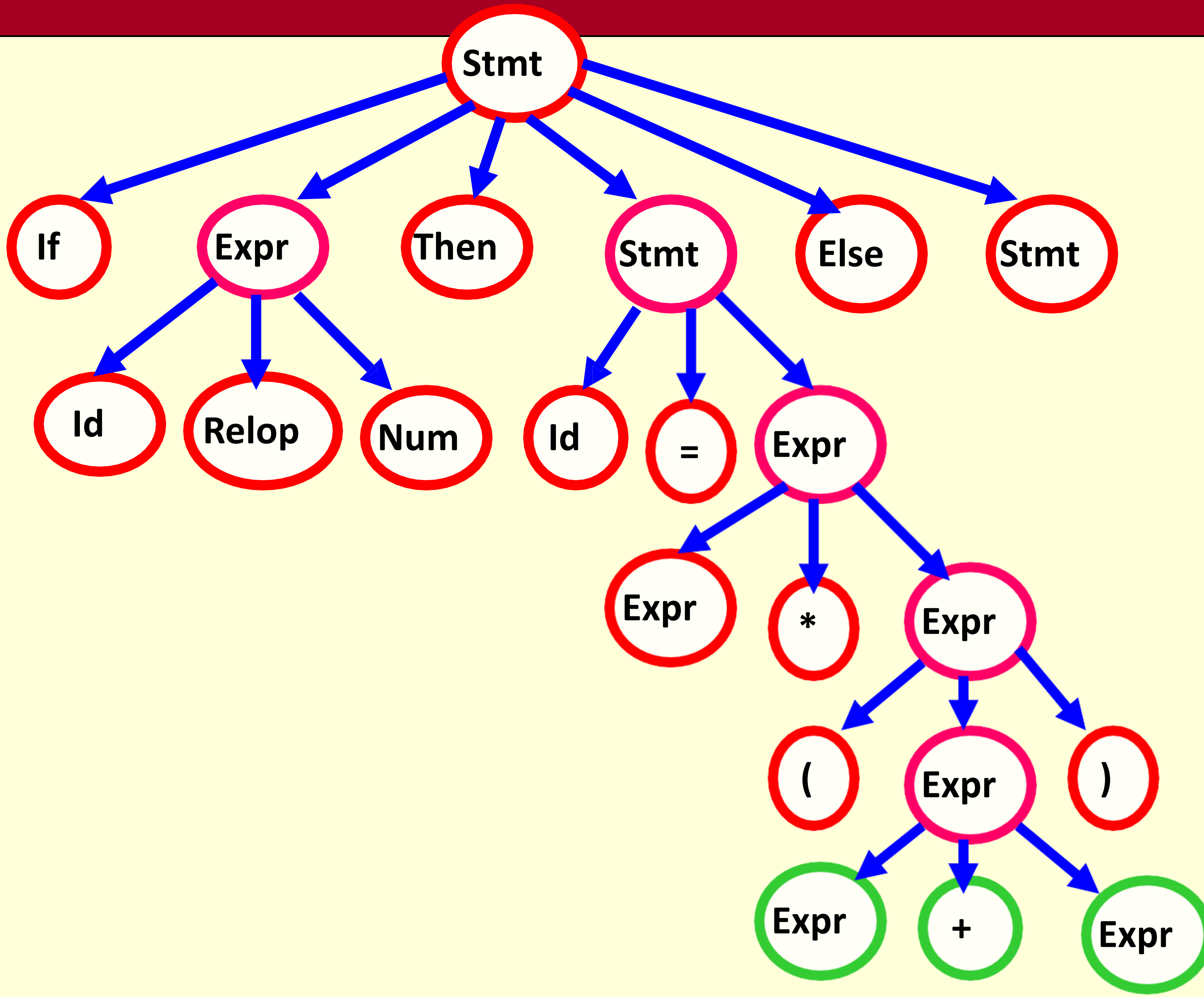


Stmt
If
Expr
Then
Stmt
Else
Stmt
Id
Relop
Num
Id
=
Expr
Expr
*
Expr
(
Expr
)

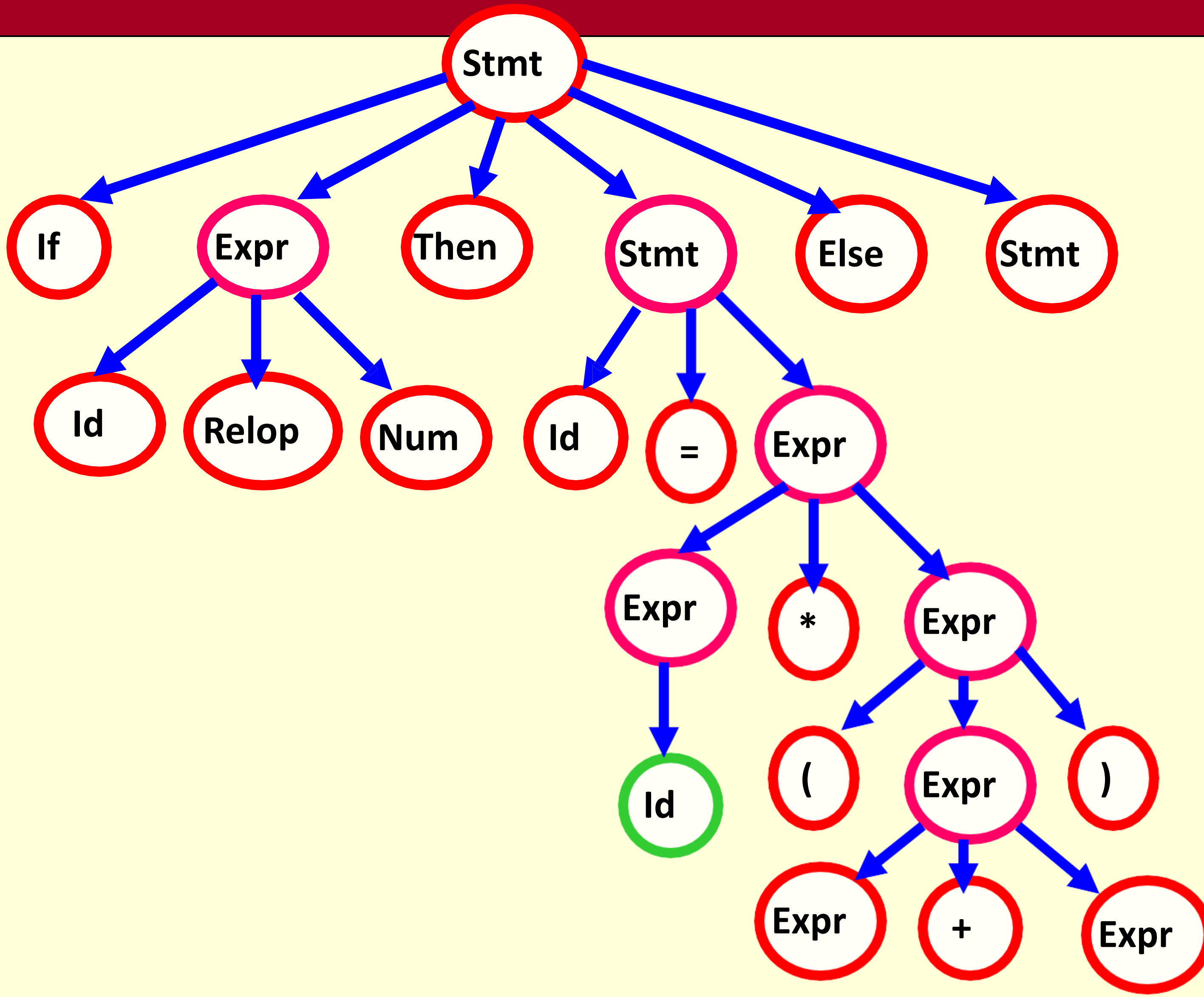
Example: Parse Tree



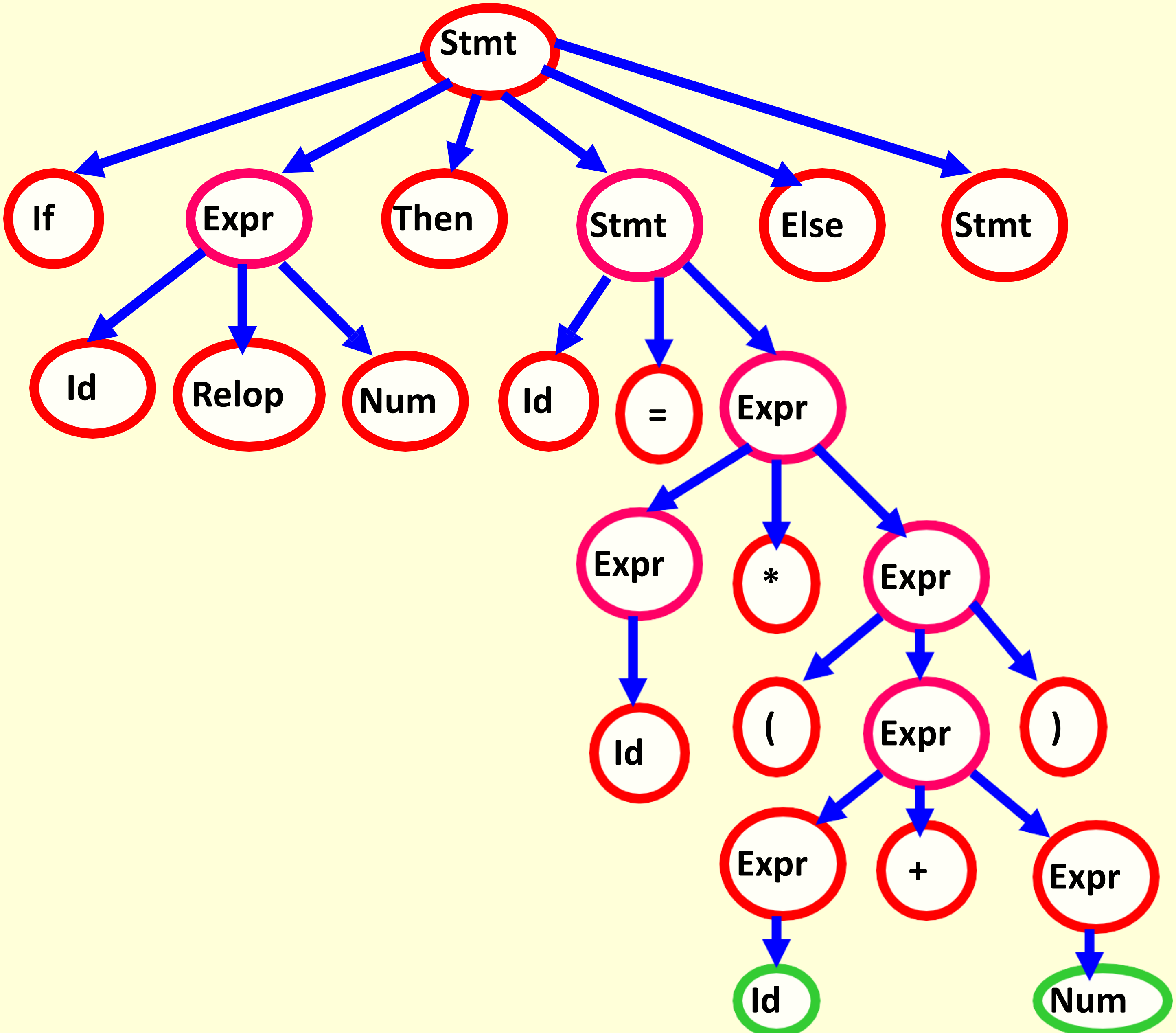
Example: Parse Tree



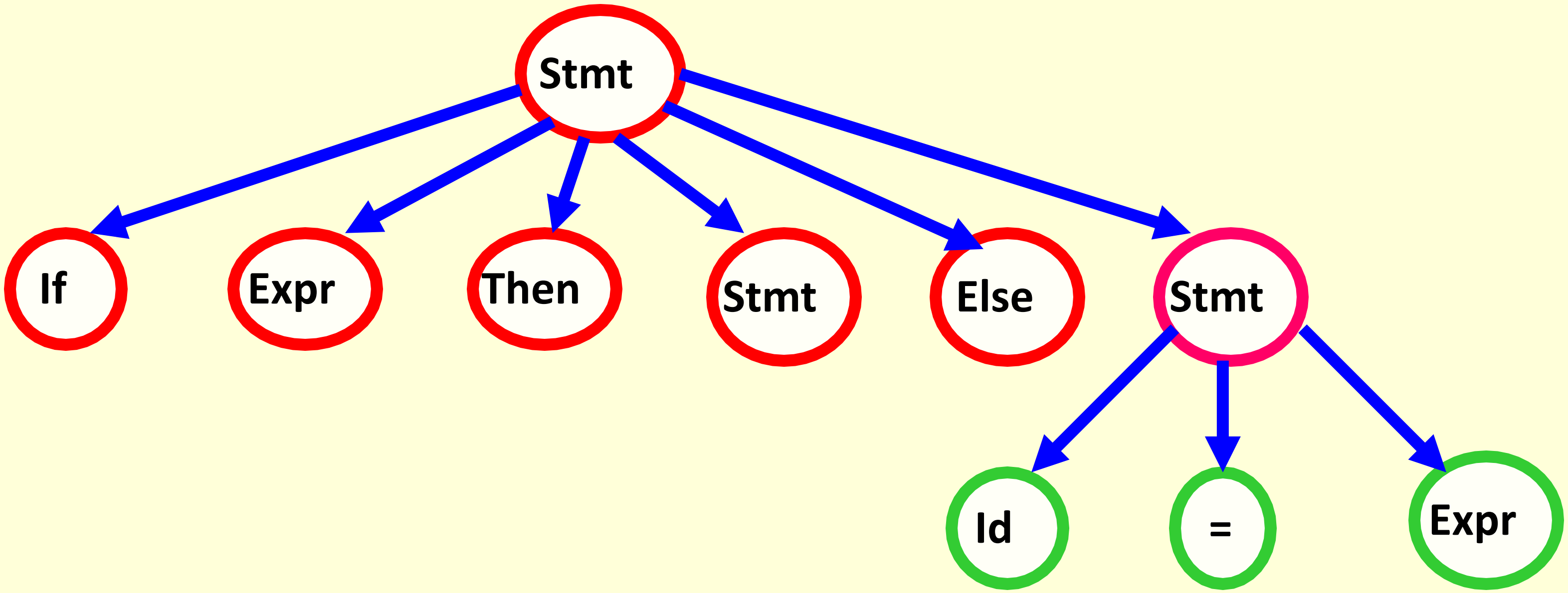
Example: Parse Tree



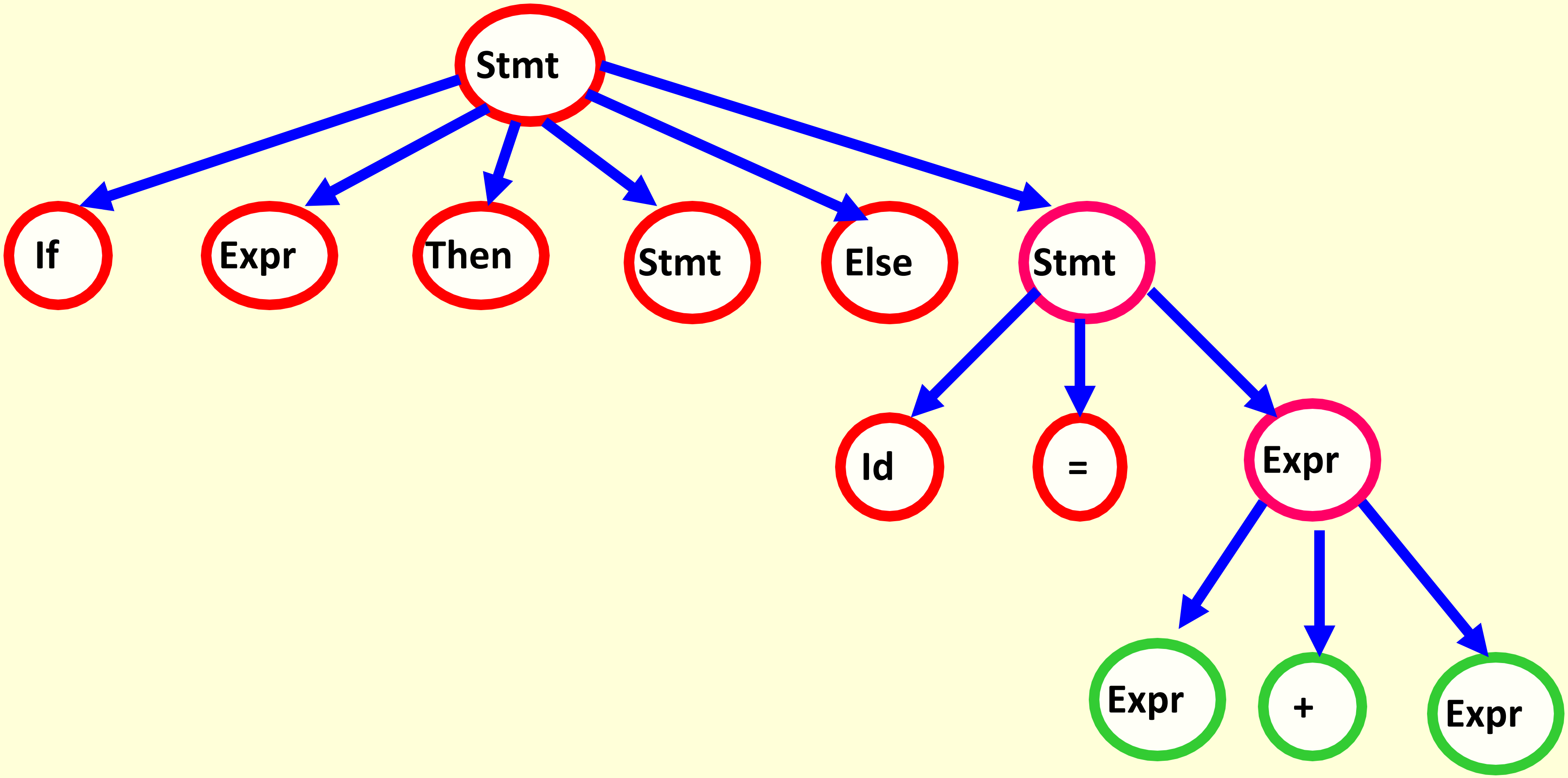
Example: Parse Tree



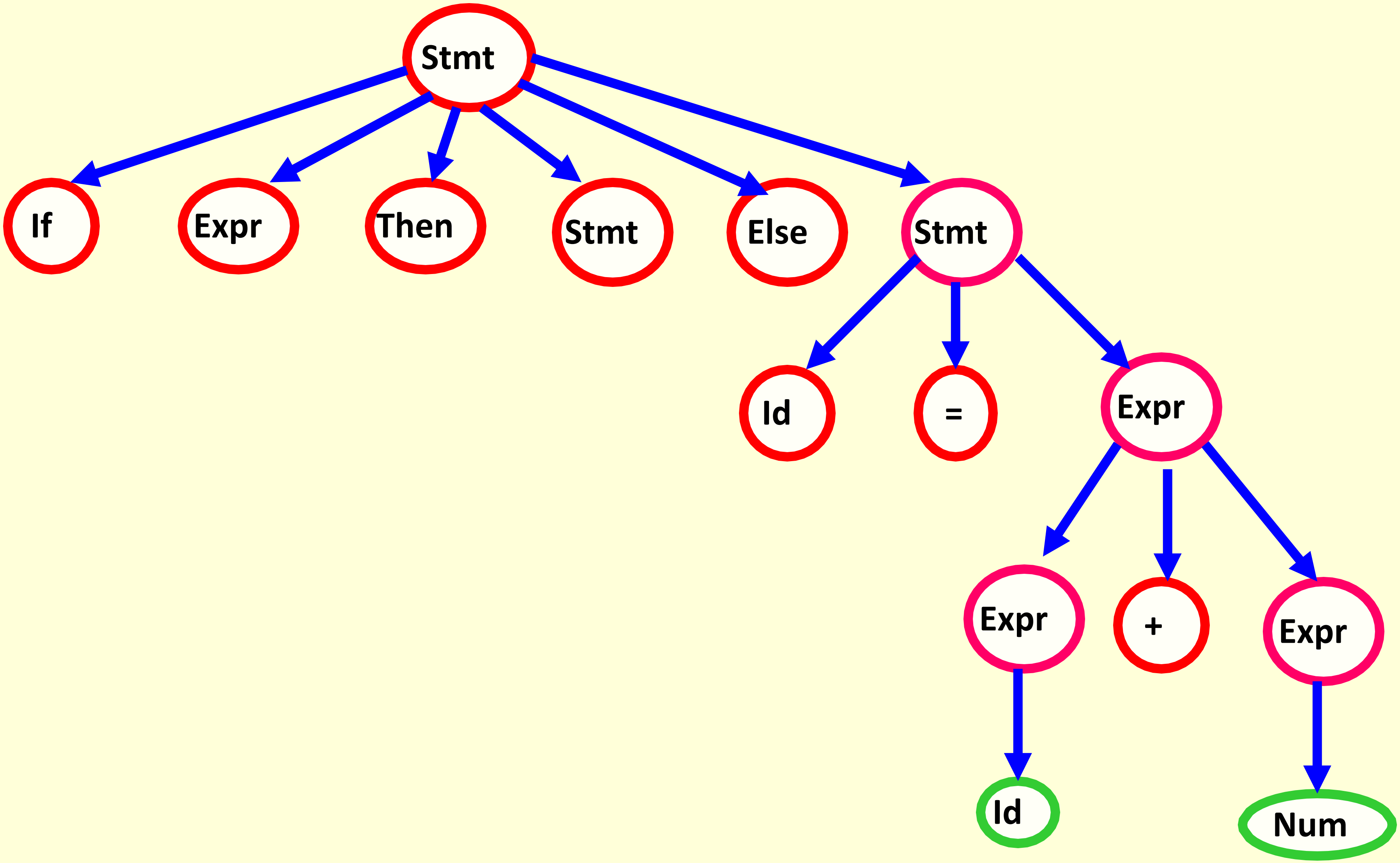
Example: Parse Tree(Else Part)



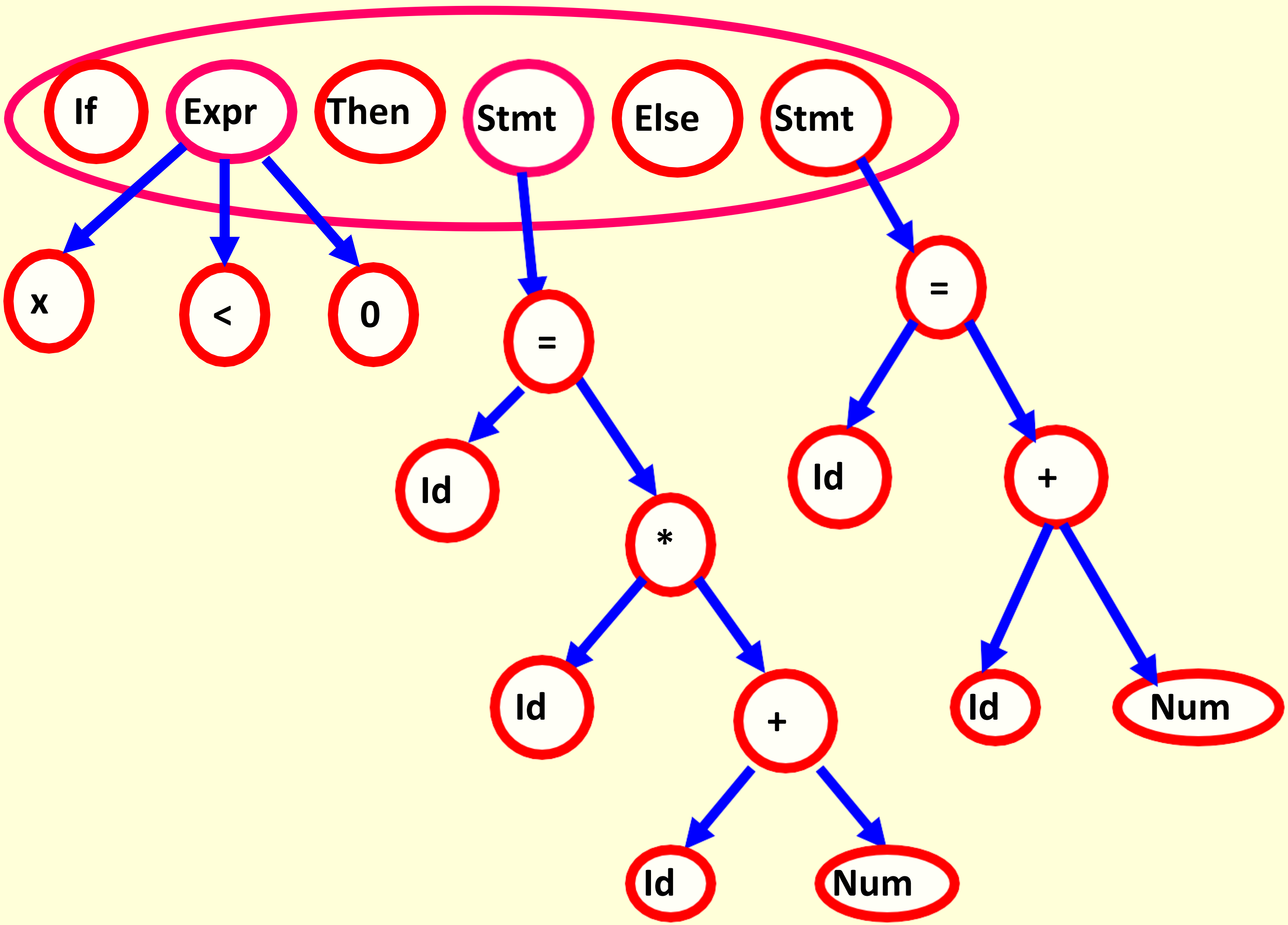
Example: Parse Tree(Else Part)



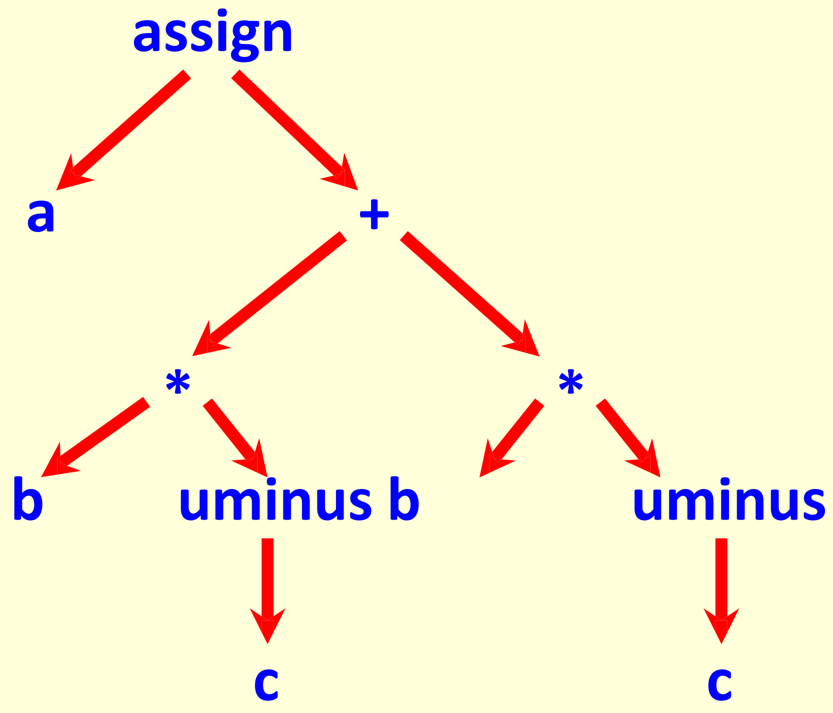
Example: Parse Tree (Else Part)



Example: Syntax Tree

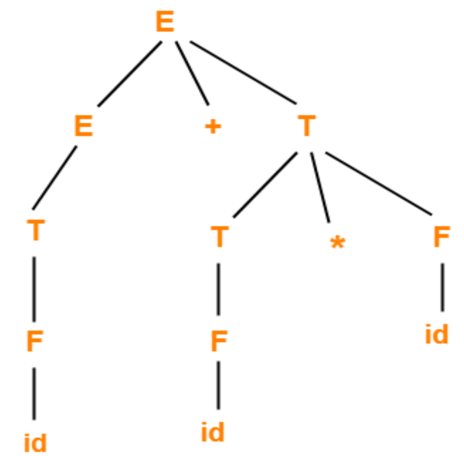


Example-2: Syntax Tree a:=b* -c + b * - c

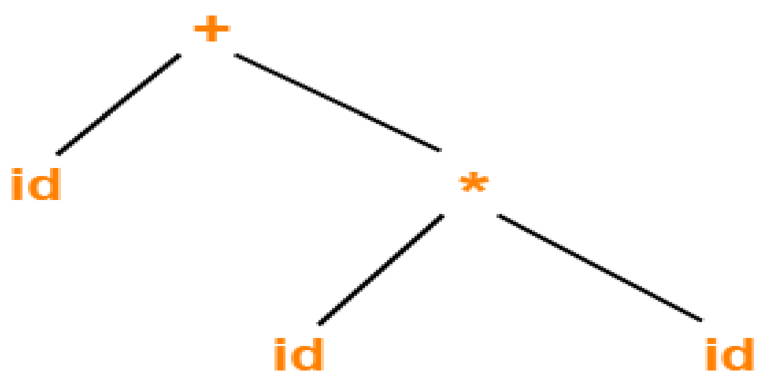


Example of Syntax Tree

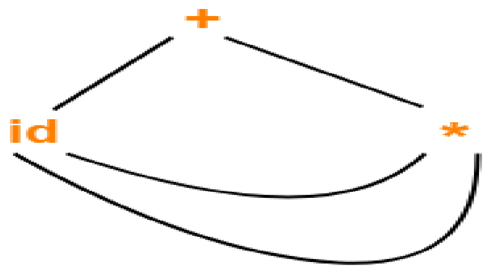
- Considering the following grammar-
- E → E + T | T
- T → T x F | F
- F → (E) | id
- Generate the following for the string id + id x id
- 1)Parse tree
- 2)Syntax tree
- 3) Directed Acyclic Graph (DAG)



Parse Tree



Syntax Tree



Directed Acyclic Graph

Construction of Syntax tree

postfix

Construct a syntax tree for the following arithmetic expression-

(a + b) * (c – d) + ((e / f) * (a + b))

solution

(a + b) * (c – d) + ((e / f) * (a + b))

ab+ * (c – d) + ((e / f) * (a + b))

ab+ * cd- + ((e / f) * (a + b))

ab+ * cd- + (ef/ * (a + b))

ab+ * cd- + (ef/ * ab+)

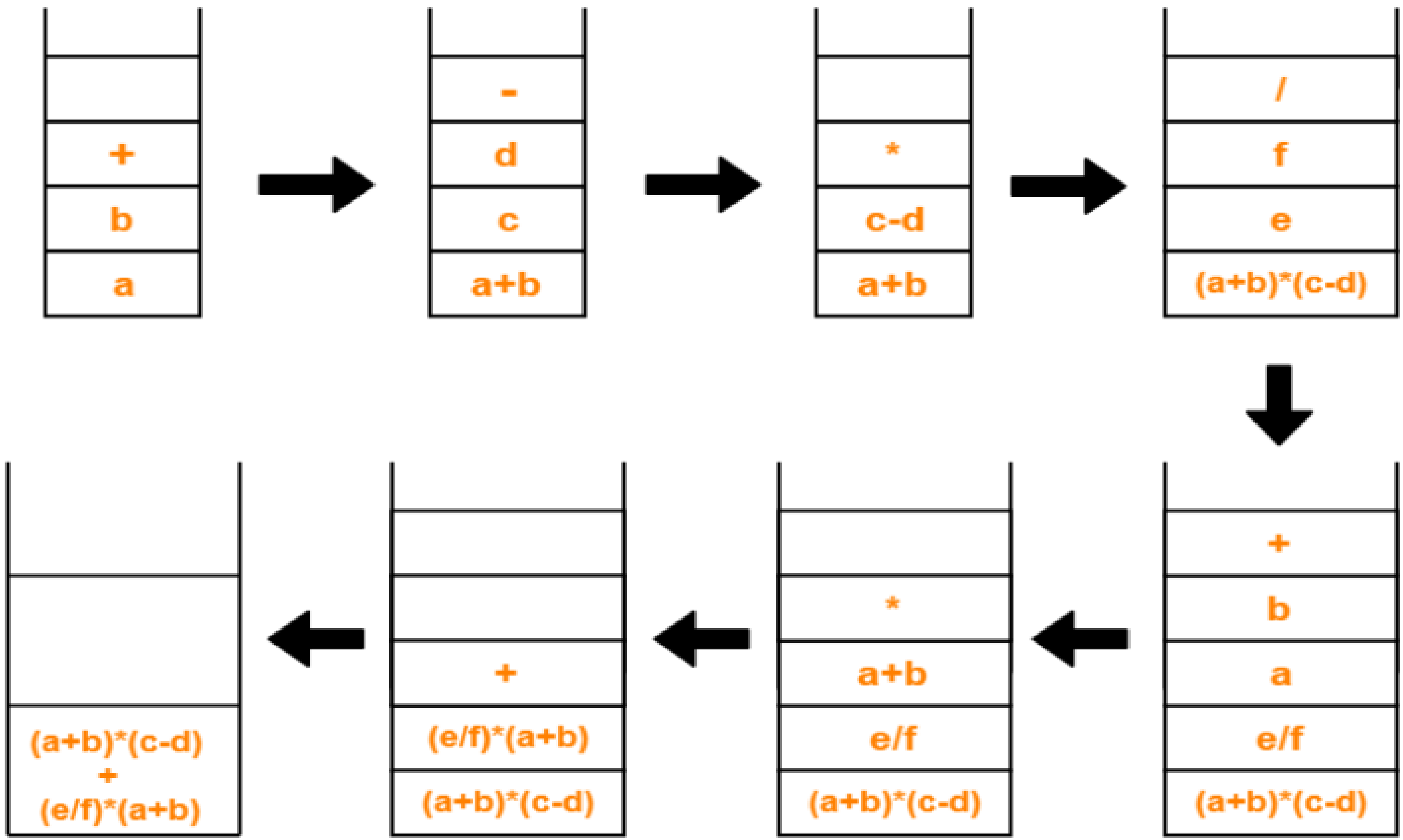
ab+ * cd- + ef/ab+*

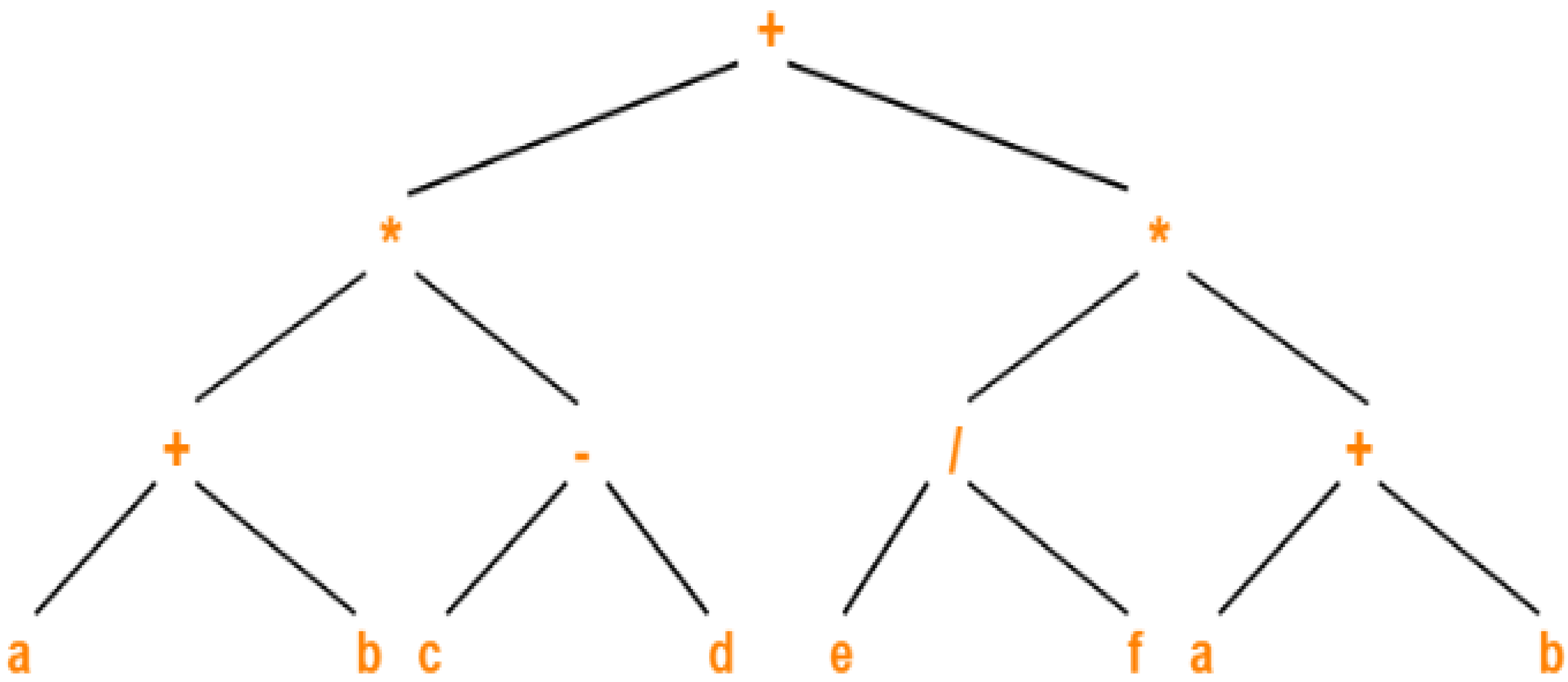
ab+cd-* + ef/ab+*

ab+cd-*ef/ab+*+

Postfix evaluation algorithm

- Start pushing the symbols of the postfix expression into the stack one by one.
- When an operand is encountered,
- Push it into the stack.
- When an operator is encountered
- Push it into the stack.
- Pop the operator and the two symbols below it from the stack.
- Perform the operation on the two operands using the operator you have in hand.
- Push the result back into the stack.
- Continue in the similar manner and draw the syntax tree simultaneously.


+
b
a
-
d
c
a+b
*
c-d
a+b
/
f
e
(a+b)*(c-d)
+
b
a
e/f
(a+b)*(c-d)
*
a+b
e/f
(a+b)*(c-d)
+
(e/f)*(a+b)
(a+b)*(c-d)
(a+b)*(c-d)
+
(e/f)*(a+b)



Syntax Tree

Syntax directed translation

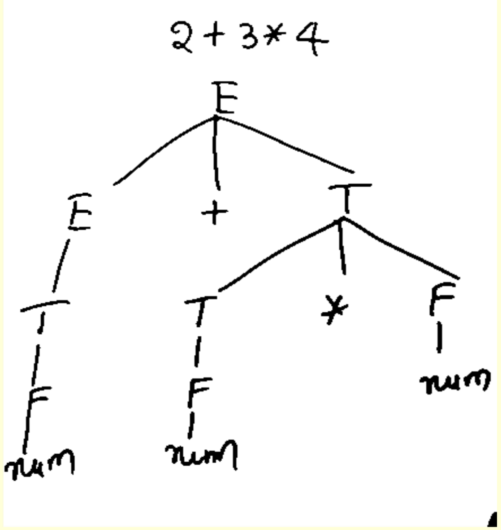
- Grammar + Semantic rules = SDT

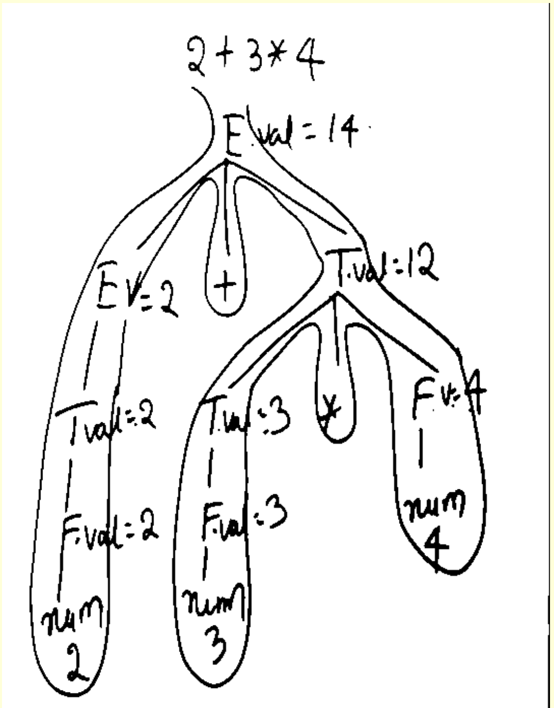
- SDT for evaluation of expression

- E -> E+T / T { E.value =E.value + T.value }
  { E.value = T.value }

- T -> T * F / F { T.value = T.value * F.value
  { T.value = F.value }

- F-> NUM { F.value = num.L value }





Concrete tree and abstract tree

- E → E1+T { E.nptr =mknode(E1.nptr , '+' , T.nptr); }
- E →T { E.nptr = T.nptr; }
- T → T1 * F { T.nptr =mknode(T1.nptr , '*' , F.nptr);}
- T→F { T.nptr = F.nptr ;}
- F-> NUM { F.nptr = mknode(null,idname,null); }

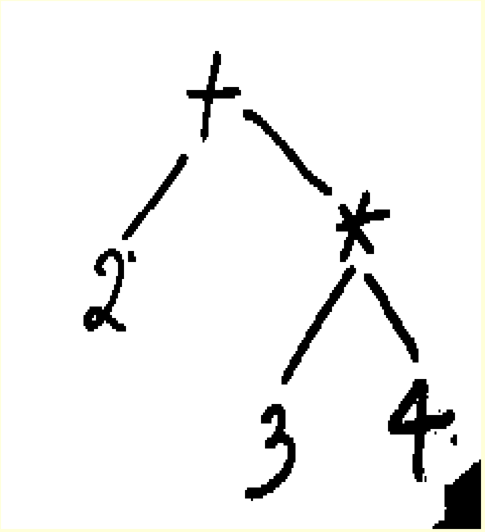


Fig: Abstract tree

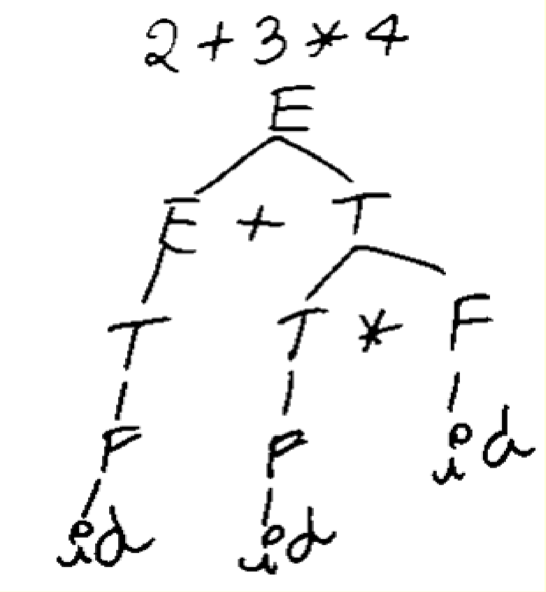
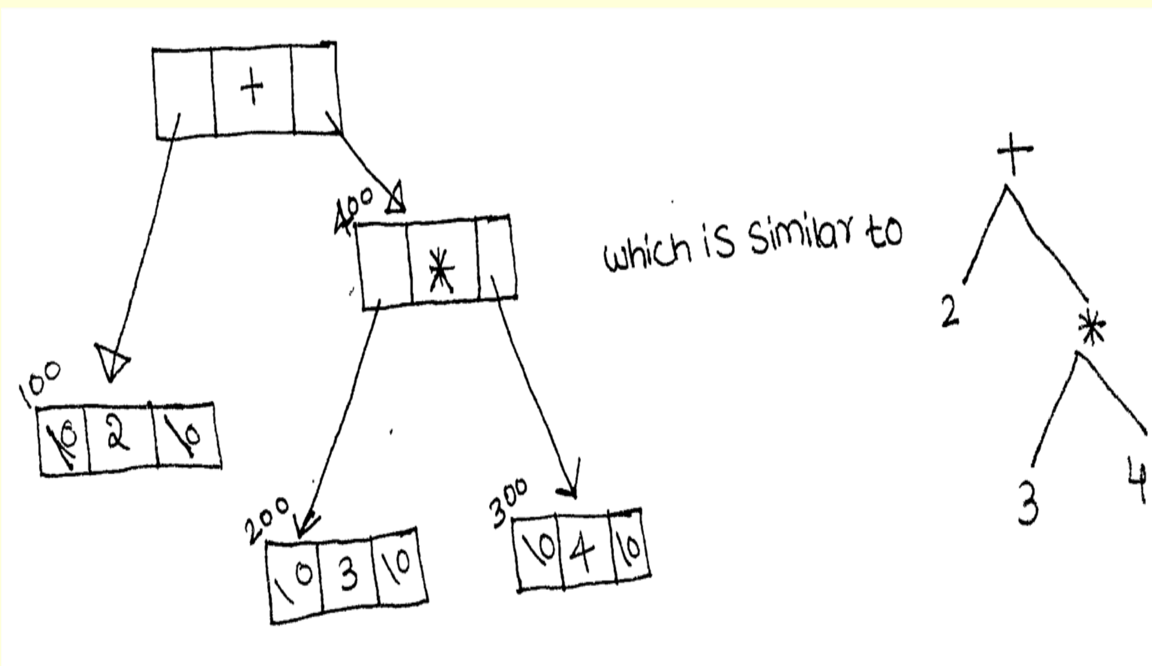
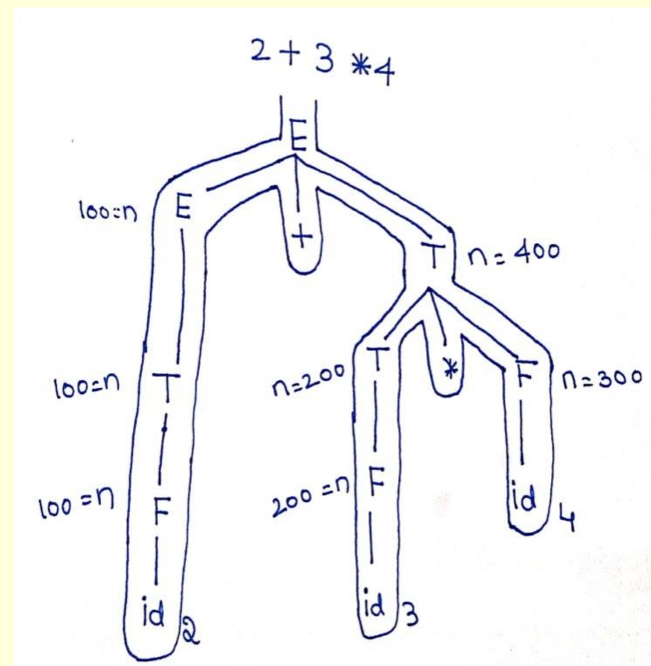


Fig: Concrete tree

Concrete parse tree

- E → E1+T   { E.nptr =mknode(E1.nptr , ’+’ , T.nptr); }
- E →T   { E.nptr = T.nptr; }
- T → T1 * F   { T.nptr =mknode(T1.nptr , ‘*’ , F.nptr);}
- T→F   { T.nptr = F.nptr ;}
- F-> NUM   { F.nptr = mknode(null,idname,null); }





Concrete parse tree

Attribute Grammar

- Attribute grammar is a special form of context-free grammar where some additional information (attributes) is appended to one or more of its non-terminals in order to provide context-sensitive information.
- Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.
- Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language.
- Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example:

- **E → E + T { E.value = E1.value + T.value }**
- The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted.
- Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.
- Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions.
- Attributes may be divided into two categories

  1) synthesized attributes

  2)inherited attributes.

Synthesized attributes

- A Synthesized attribute is an attribute of the non-terminal on the left-hand side of a production with semantic value
- The attribute can take value only from its children (Variables in the RHS of the production).
- For e.g. 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.
- Ex: A->BCD
- A is calculate with its children B,C,D values

Example of S-attributed SDD

• Production	• Semantic Rules
1)L -> E n	L.val = E.val
2)E -> E1 + T	E.val = E1.val + T.val
3)E -> T	E.val = T.val
4)T -> T1 * F	T.val = T1.val * F.val
5)T -> F	T.val = F.val
6)F -> (E)	F.val = E.val
7)F -> digit	F.val = digit.lexval

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.
- Example of S-attributed SDD

Production	Semantic Rules
L -> E n	L.val = E.val
E -> E1 + T	E.val = E1.val + T.val

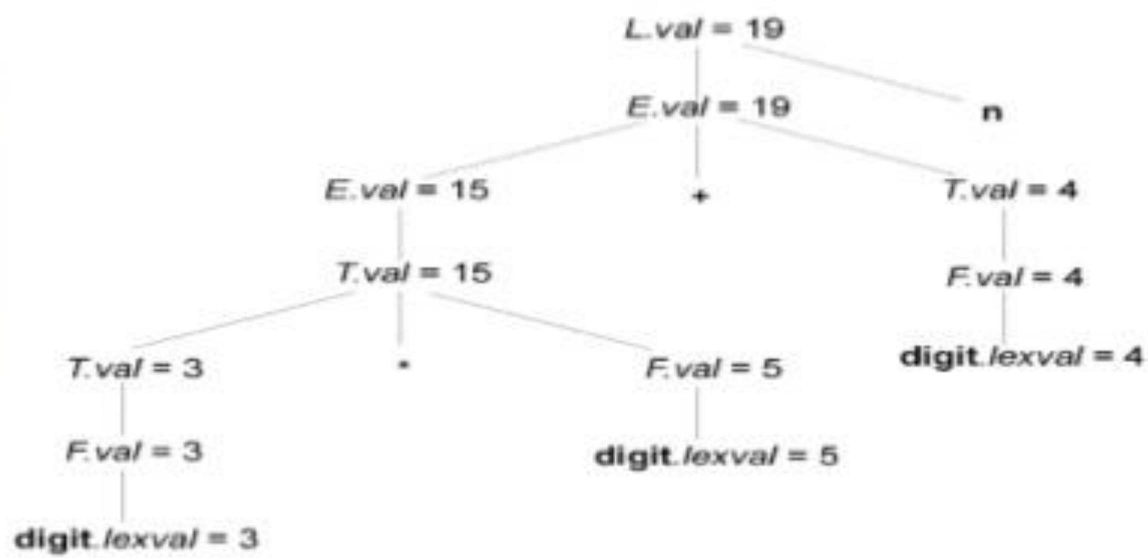
Evaluating an SDD at the Nodes of a Parse Tree

Production	Semantic Rules
1) $L \rightarrow E$ **n**	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow$ **digit**	$F.val =$ **digit**.$lexval$

val and ***lexval*** are synthesized attributes

With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree.

Annotated parse tree:
3*5 + 4 n



Inherited attributes

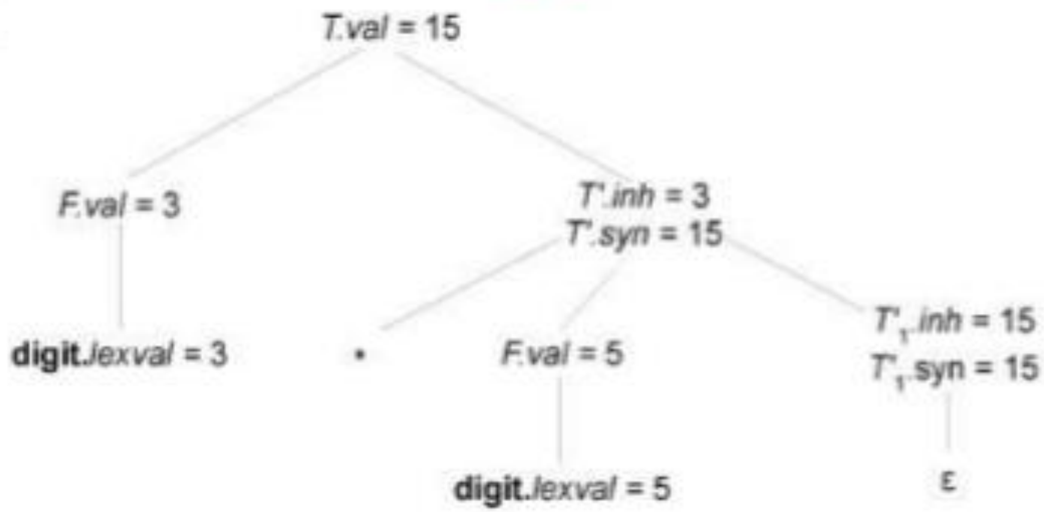
- An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute.
- The attribute can take value either from its parent or from its siblings (variables in the LHS or RHS of the production)
- For example, let’s say A -> BC is a production of a grammar and C’s attribute is dependent on A’s attributes or B’s attributes then it will be inherited attribute.

Evaluating an SDD at the Nodes of a Parse Tree

Production	Semantic Rules
$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow {}^*F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
$F \rightarrow$ **digit**	$F.val =$ **digit**.*lexval*

An SDD with both inherited and synthesized attributes does not ensure any guaranteed order; even it may not have an order at all.

Annotated parse tree: 3*5



L-attributed SDT:

- If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.
- Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS.
- Example : S -> MN {S.val= M.val + N.val}

Evaluation orders for SDD's

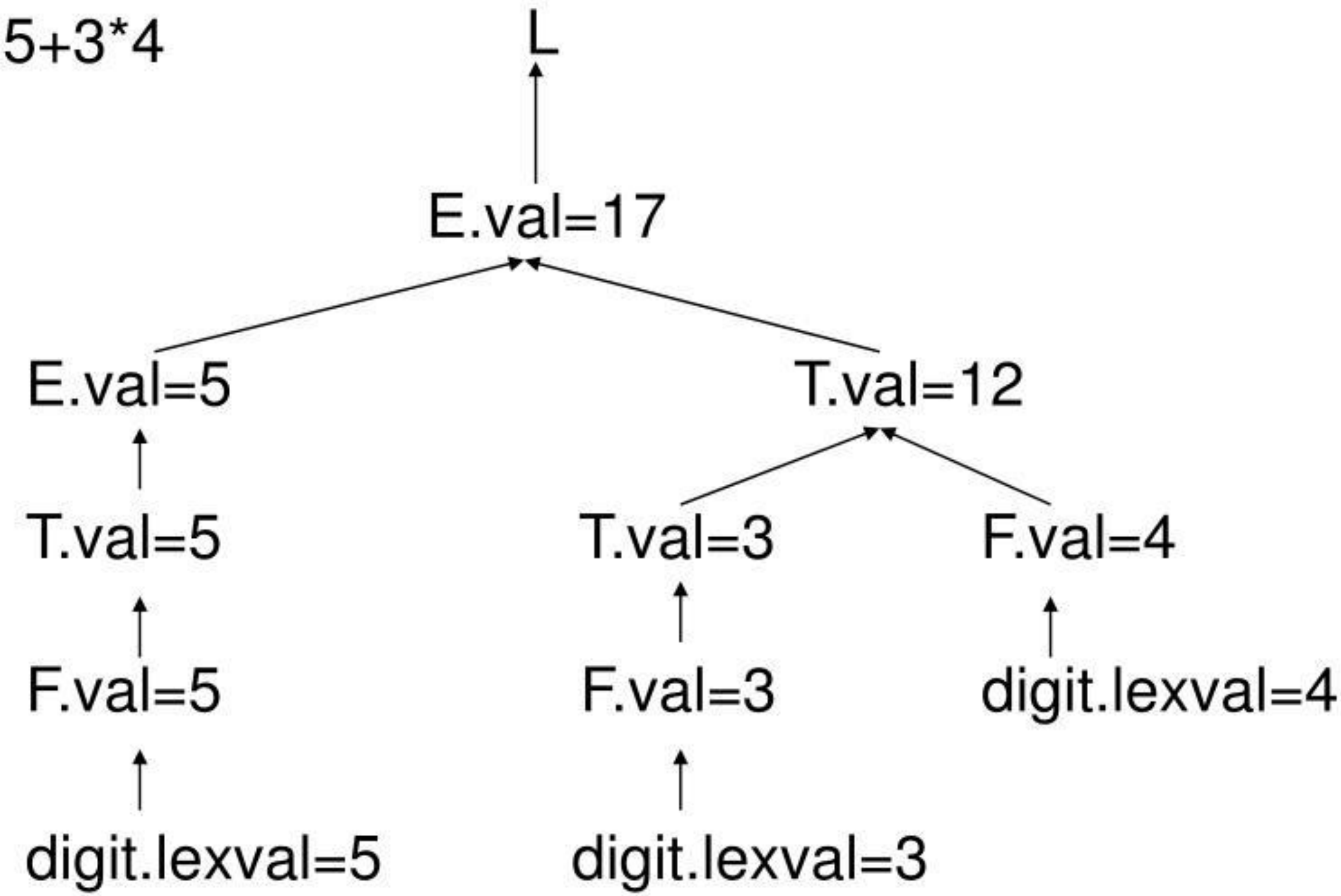
- A dependency graph is used to determine the order of computation of attributes
- Dependency graph
- If a semantic rule defines the value of synthesized attribute A.b in terms of the value of X.c then the dependency graph has an edge from X.c to A.b
- If a semantic rule defines the value of inherited attribute B.c in terms of the value of X.c then the dependency graph has an edge from X.c to B.c

Ordering the evaluation of attributes

- If dependency graph has an edge from M to N then M must be evaluated before the attribute of N
- Thus the only allowable orders of evaluation are those sequence of nodes N1,N2,…,Nk such that if there is an edge from Ni to Nj then i<j
- Such an ordering is called a topological sort of a graph

Dependency Graph Example

Input: 5+3*4

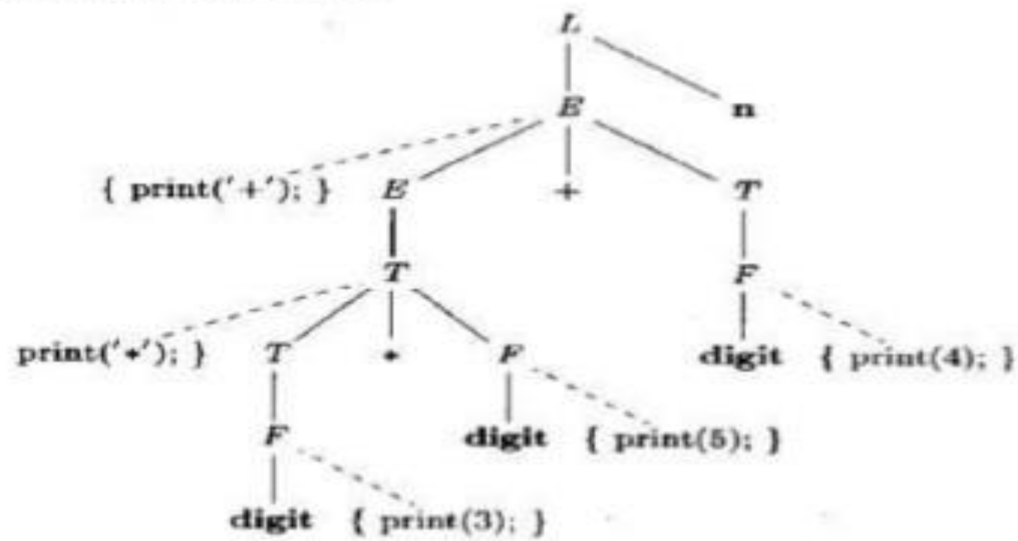


SDTs with Actions inside Productions

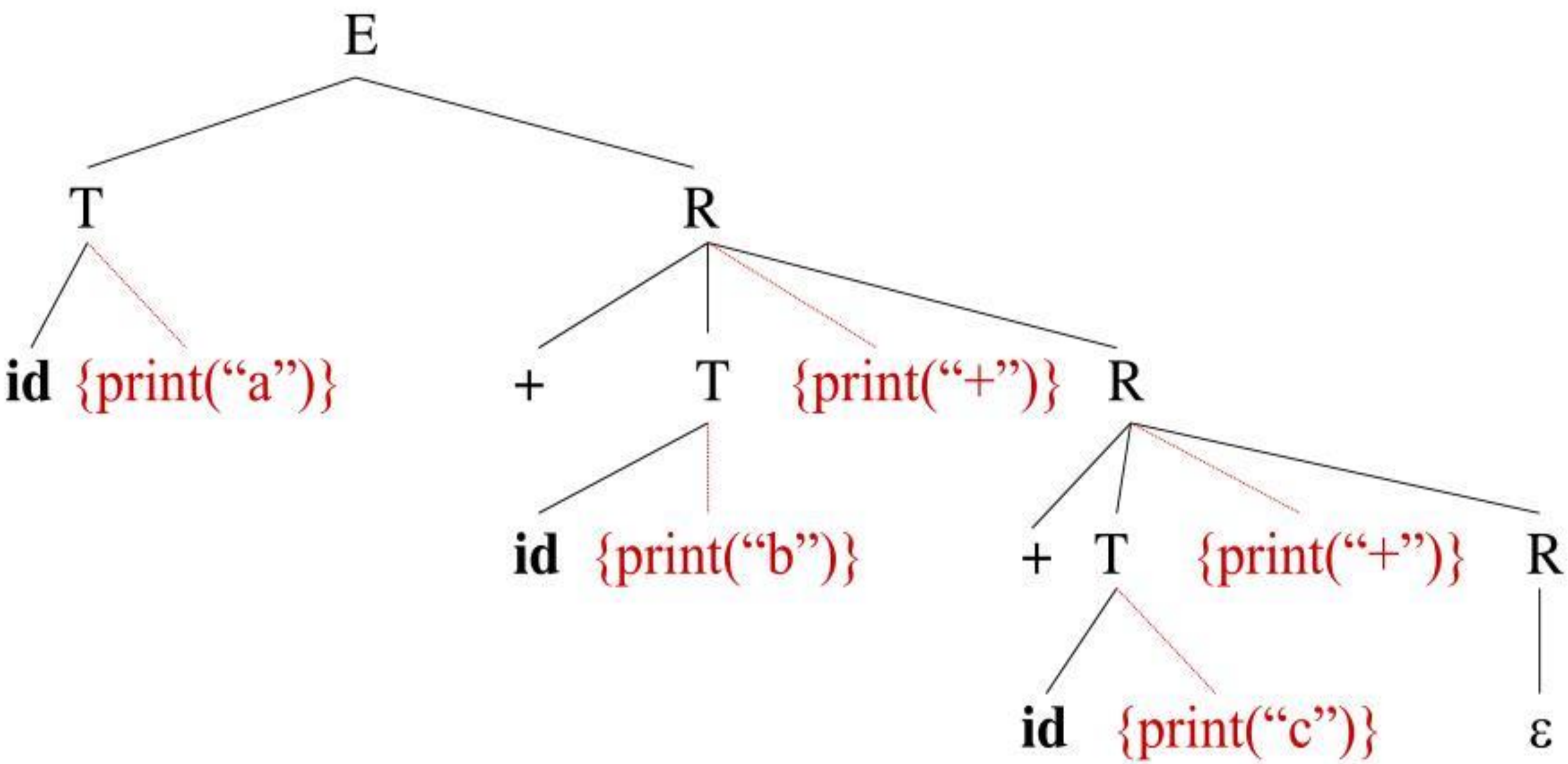
SDT for infix-to-prefix translation during parsing

1) $L \rightarrow E\ \mathbf{n}$
2) $E \rightarrow \{\ \text{print}('+');\ \}\ E_1 + T$
3) $E \rightarrow T$
4) $T \rightarrow T_1 * F\ \{\ \text{print}('*');\ \}$
5) $T \rightarrow F$
6) $F \rightarrow (\ E\ )$
7) $F \rightarrow \mathbf{digit}\ \{\ \text{print}(\mathbf{digit}.\mathit{lexval});\ \}$

Parse Tree with Actions Embedded



A Translation Scheme Example



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

Applications

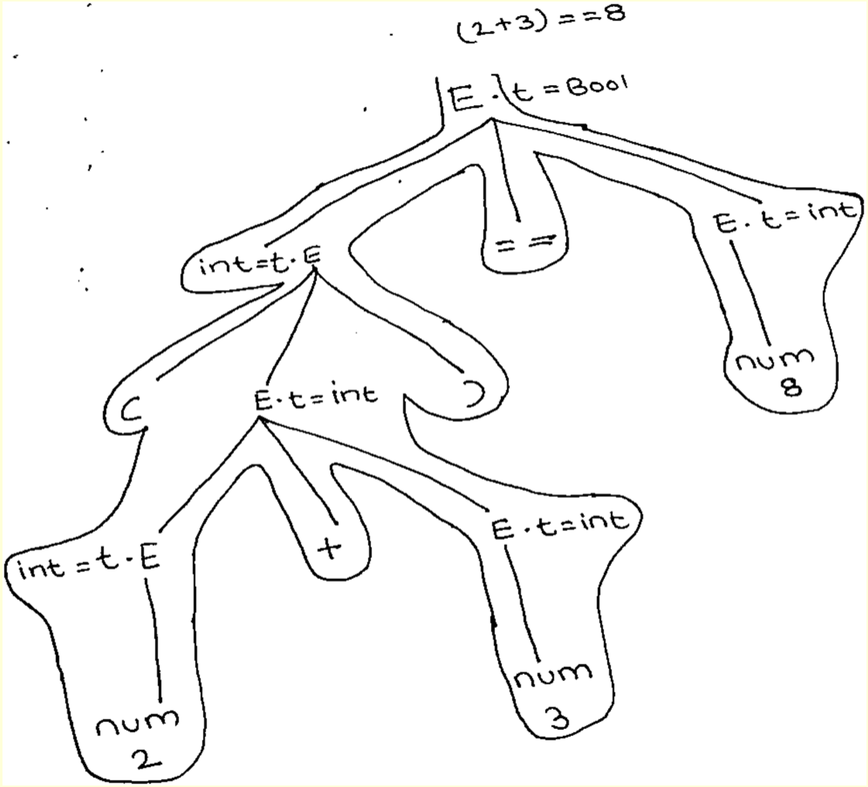
- Type checking
- Postfix evaluation
- Intermediate Code generation

Declarations

$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \mathbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$

Type check

- E → E1 + E2 { if((E1.type == E2.type) && (E1.type = int)) then
  E.type = int else error;}
- E → E1==E2 { if((E1.type == E2.type) && (E1.type =int/boolean)) then
  E.type = boolean else error;}
- E →(E1) { E.type = E1.type;}
- E → num { E.type = int;}
- E → true { E.type = boolean;}
- E → false { E.type = boolean;}



Example of postfix SDT schema

1) L -> E n {print(E.val);}

2) E -> E1 + T E.val=E1.val+T.val;}

3) E -> T {E.val = T.val;}

4) T -> T1 * F T.val=T1.val*F.val;}

5) T -> F {T.val=F.val;}

6) F -> (E) {F.val=E.val;}

7) F -> digit {F.val=digit.lexal;}

Example

```
L -> E n          {print(stack[top-1].val);
                   top=top-1;}
E -> E1 + T       {stack[top-2].val=stack[top-2].val+stack.val;
                  top=top-2;}
E -> T
T -> T1 * F       {stack[top-2].val=stack[top-2].val+stack.val;
                  top=top-2;}
T -> F
F -> (E)                {stack[top-2].val=stack[top-1].val
                top=top-2;}
F -> digit
```

Gate Questions

QUESTION NO 1:

- Incompatable types work with the _______________

A. Syntax tree

B.semantic analyzer

C.Code optimizer

D.Lexical analyzer

Question 2

Generation of intermediate code based on an abstract machine model is useful in compilers because

(a) it makes implementation of lexical analysis and syntax analysis easier

(b) syntax-directed translations can be written for intermediate code generation

(c) it enhances the portability of the front end of the complier

(d) it is not possible to generate code for real machines directly from high level language programs

[1994 : 1 Mark]

Explanation

(a)

Generation of intermediate code based on an abstract machine model is useful in compilers because it makes implementation of lexical analysis and syntax analysis easier.

Question 3

A linker is given object modules for a set of programs that were compiled separately. What information need to be included in an object module?

(a) Object code

(b) Relocation bits

(c) Names and locations of all external symbols defined in the object module

(d) Absolute addresses of internal symbols

[1995 : 1 Mark]

(d)

A linker is a computer program that takes one or more object files generated by a compiler and combines them into a single executable program. The linker also takes care of arranging the objects in a program's address space. Therefore absolute addresses of internal symbols need to be included in an object module.

Question 4

In the following grammar

$$X ::= X \oplus Y / Y$$
$$Y ::= Z \odot Y / Z$$
$$Z ::= id$$

Which of the following is true?

(a) '$\oplus$' is left associative while '$\odot$' is right associative

(b) Both '$\oplus$' and '$\odot$' is left associative

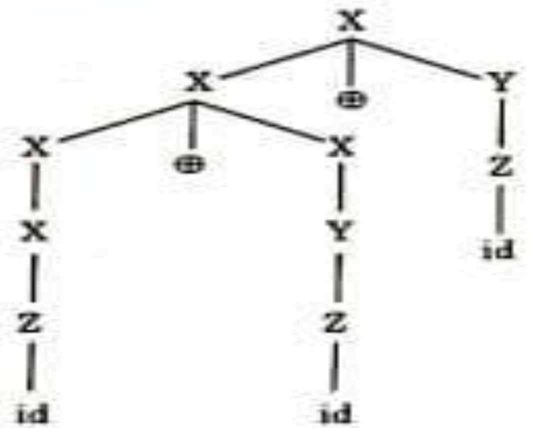
(c) '$\oplus$' is the right associative while '$\odot$' is left associative

(d) None of the above

[1997 : 1 Mark]

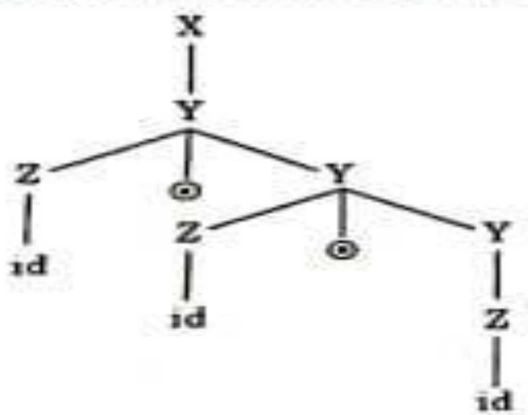
(a)

Creating syntax tree for id⊕id⊕d :



Therefore ⊕ is left associative.

Creating syntax tree for id⊙id ⊙id :



Therefore, ⊙ is right associative.

Question 5

Consider the translation scheme shown below:

$S \rightarrow T\ R$

$R \rightarrow +\ T\ \{print\ ('+');\}\ R \mid \varepsilon$

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

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

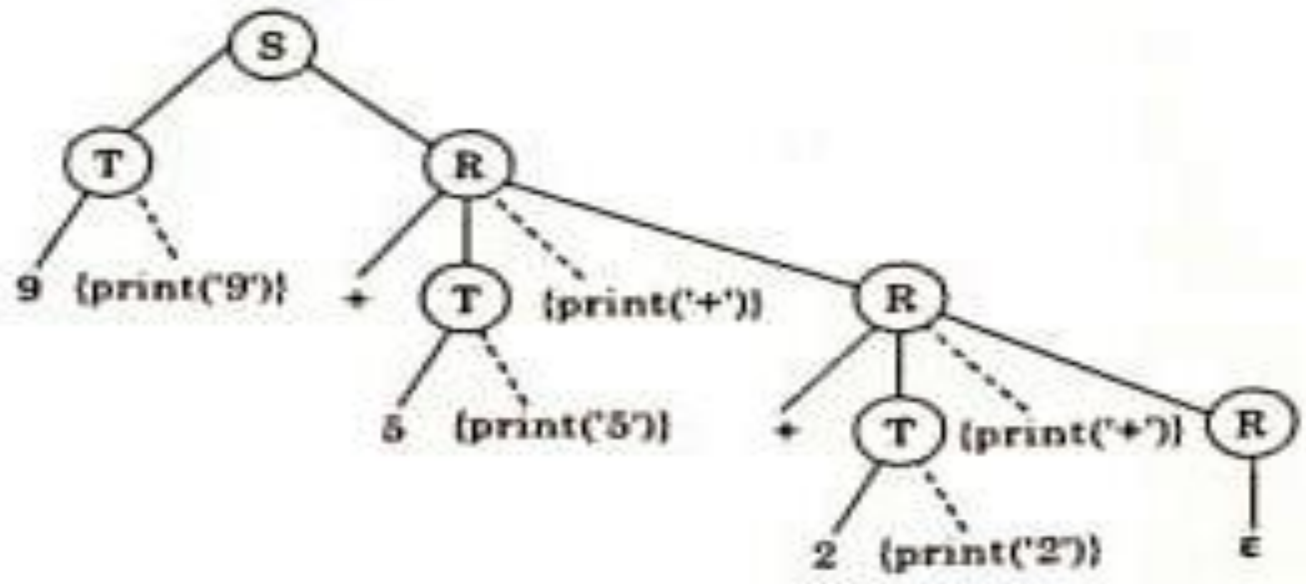
(a) 9 + 5 + 2 (b) 9 5 + 2 +

(c) 9 5 2 + + (d) + + 9 5 2

[2003 : 2 Marks]

3.5 (b)

For the input '9 + 5 + 2' the translation scheme is 95 + 2 + shown below:



Question 6

Consider the syntax directed definition shown below:

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

$E \rightarrow E_1 + E_2$ {t = newtemp ( );

gen (t = E_1. place + E_2. place;);

E.place = t}

$E \rightarrow id$ {E.place = id.place;}

Here, gen is a function that generates the output code, and newtemp is a function that returns the name of a new temporary variable on every call. Assume that t_i's are the temporary variable names generated by newtemp.

For the statement 'X: = Y + Z', the 3-address code sequence generated by this definition is

(a) $X = Y + Z$

(b) $t_1 = Y + Z; X = t_1$

(c) $t_1 = Y; t_2 = t_1 + Z; X = t_2$

(d) $t_1 = Y; t_2 = Z; t_3 = t_1 + t_2; X = t_3$

[2003 : 2 Marks]

(b)

gen( ) function will be used only two times for $X = Y + Z$ and only one temp variable is created with newtemp( ).

$\therefore\ t_1 = Y + Z;\ X = t_1$

Question 7

Consider the grammar rule $E \rightarrow E_1 - E_2$ for arithmetic expressions. The code generated is targeted to a CPU having a single user register. The subtraction operation requires the first operand to be in the register. If E_1 and E_2 do not have any common subexpression, in order to get the shortest possible code

(a) E_1 should be evaluated first

(b) E_2 should be evaluated first

(c) Evaluation of E_1 and E_2 should necessarily be interleaved

(d) Order of evaluation of E_1 and E_2 is of no consequence

[2004 : 1 Mark]

(b)

To optimize the solution evaluate the expression E_2. Then we can calculate E_1 and finally E_1 will be one of operands that will be in register and we can perform subtraction directly. But if we follow the opposite then we have to make move and store operations.

Question 8

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

$E \rightarrow E_1 \# T$ {$E.value = E_1.value * T.value$}
$\quad | \ T$ {$E.value = T.value$}
$T \rightarrow T_1 \& F$ {$T.value = T_1.value + F.value$}
$\quad | \ F$ {$T.value = F.value$}
$F \rightarrow num$ {$F.value = num.value$}

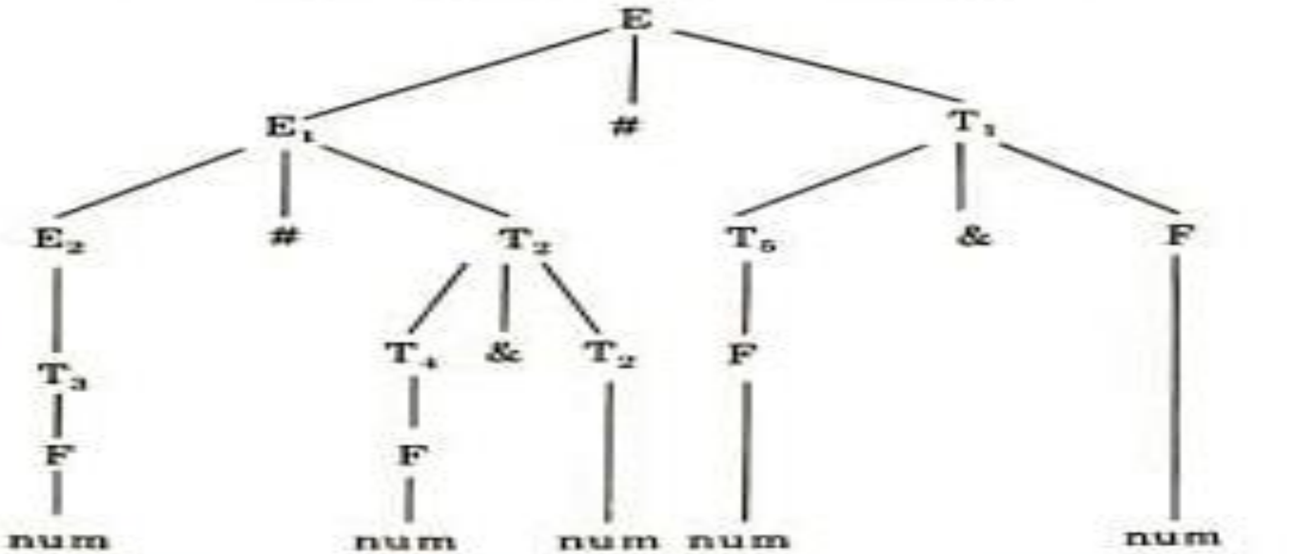
Compute E. value for the root of the parse tree for the expression: 2 # 3 & 5 # 6 & 4.

(a) 200
(b) 180
(c) 160
(d) 40

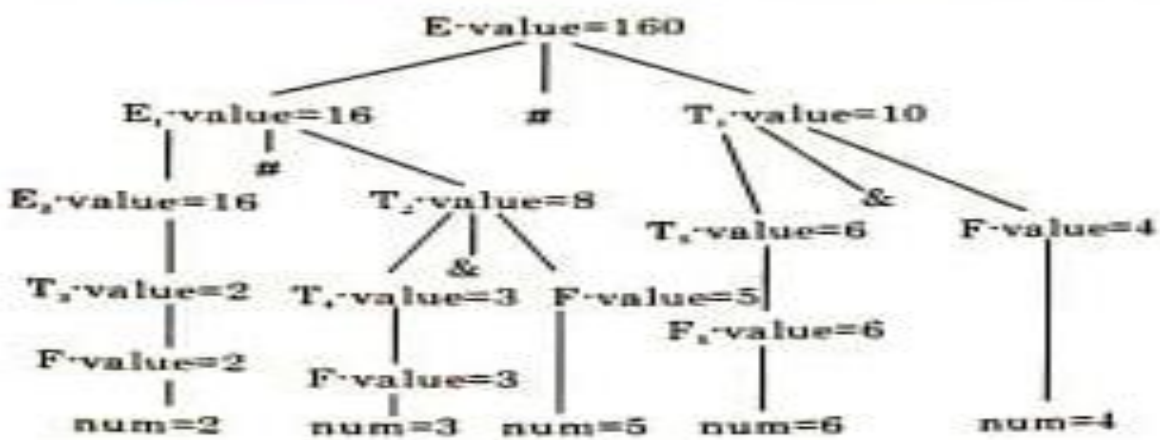
[2004 : 2 Marks]

Answer (C)

First we have to construct the parse tree.



Then we construct the annonated parse tree or parse tree with value at the leaf node.



Question 9

Consider the grammar $E \rightarrow E + n \mid E \times n \mid n$

For a sentence $n + n \times n$, the handles in the right-sentential form of the reduction are

(a) n, $E + n$ and $E + n \times n$

(b) n, $E + n$ and $E + E \times n$

(c) n, $n + n$ and $n + n \times n$

(d) n, $E + n$ and $E \times n$

[2005 : 2 Marks]

(d)

$E \rightarrow E + n \mid E \times n \mid n$

Input String $n + n \times n$

$\Rightarrow n + n \times n$

$\Rightarrow E + n \times n$ reduction $E \rightarrow n$

$\Rightarrow E \times n$ reduction $E \rightarrow E + n$

$\Rightarrow E$ reduction $E \rightarrow E \times n$

So the reductions are n, $E + n$, $E \times n$

Question 10

Consider the following translation scheme.

$S \rightarrow ER$

$R \rightarrow *E$ {print("*"); R | ε

$E \rightarrow F + E$ {print ('+'); | F

$F \rightarrow (S)$ | id {print (id. value);}

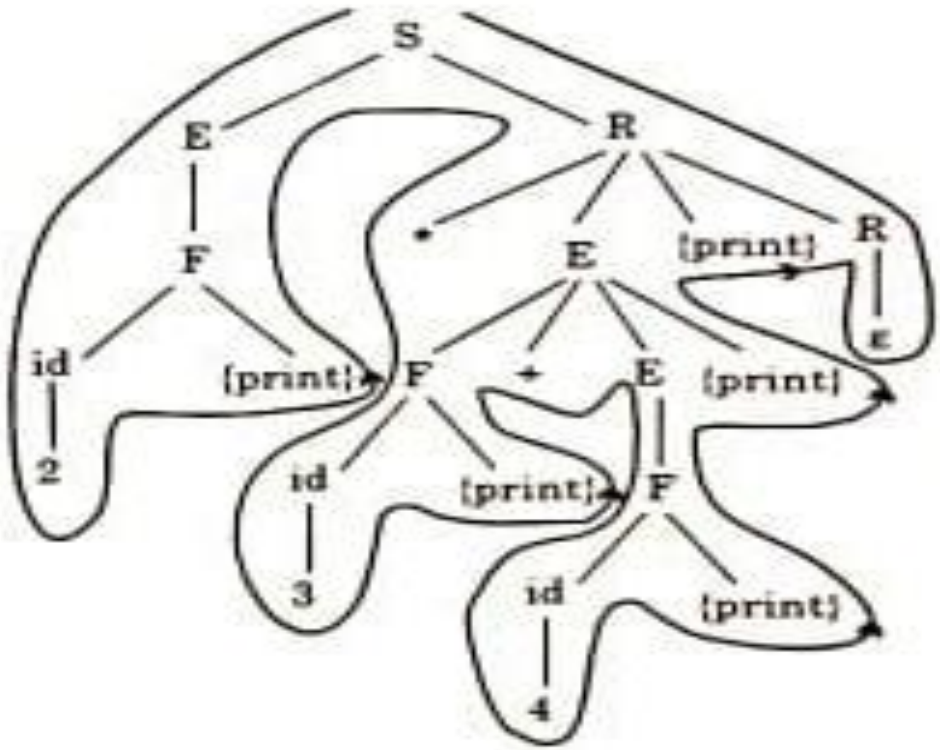
Here id is a token that represents an integer and id.value represents the corresponding integer value. For an input '2 * 3 + 4', this translation scheme prints

(a) 2 * 3 + 4 (b) 2 * + 3 4

(c) 2 3 * 4 + (d) 2 3 4 + *

[2006 : 2 Marks]

(d)

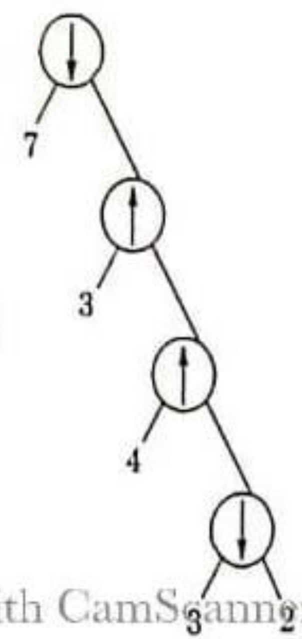


So an input 2 * 3 + 4, it prints from the above parse tree as 234 + *.

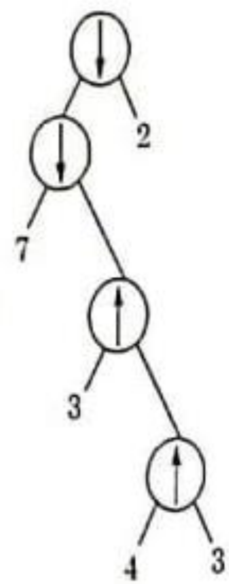
Question 11

Consider two binary operators '↑' and '↓' with the precedence of operator ↓ being lower than that of the operator ↑. Operator ↑ is right associative while operator ↓ is left associative. Which one of the following represents the parse tree for expression (7↓3↑4↑3↓2)

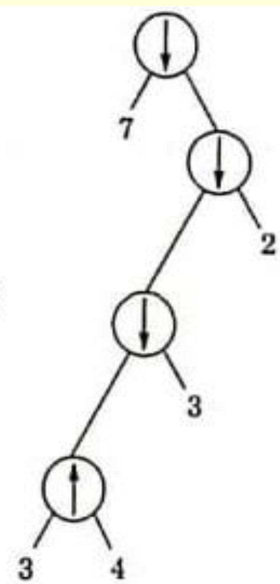
(a)



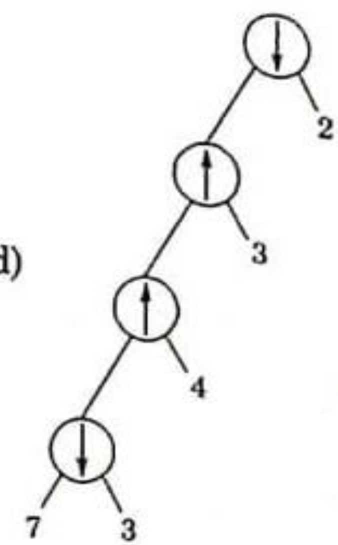
(b)



(c)

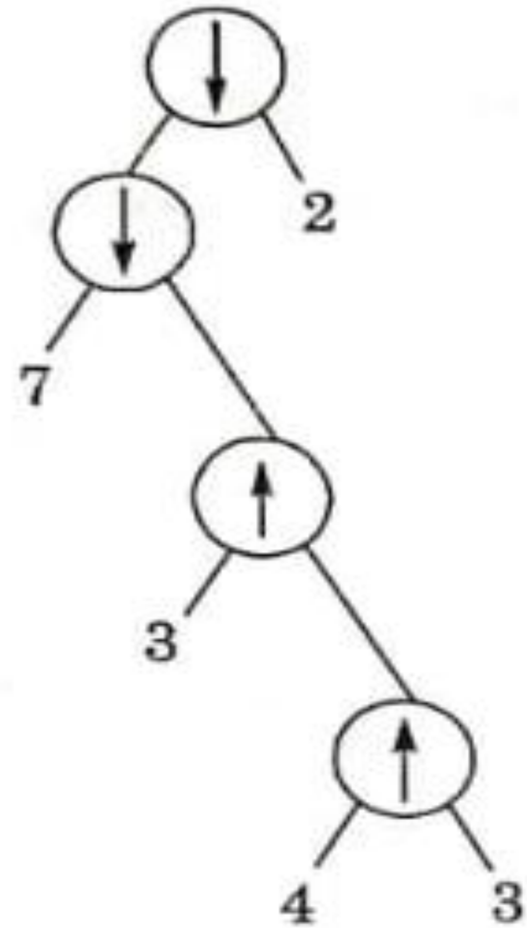


(d)



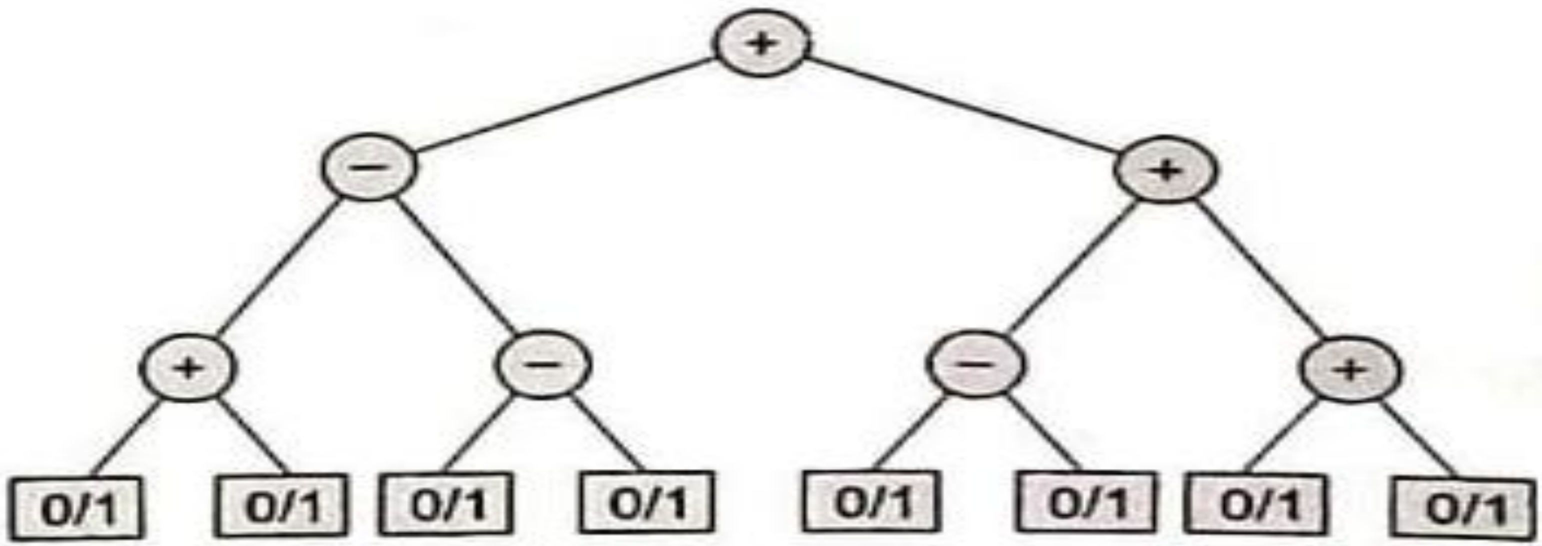
[2011 : 2 Marks]

(b)



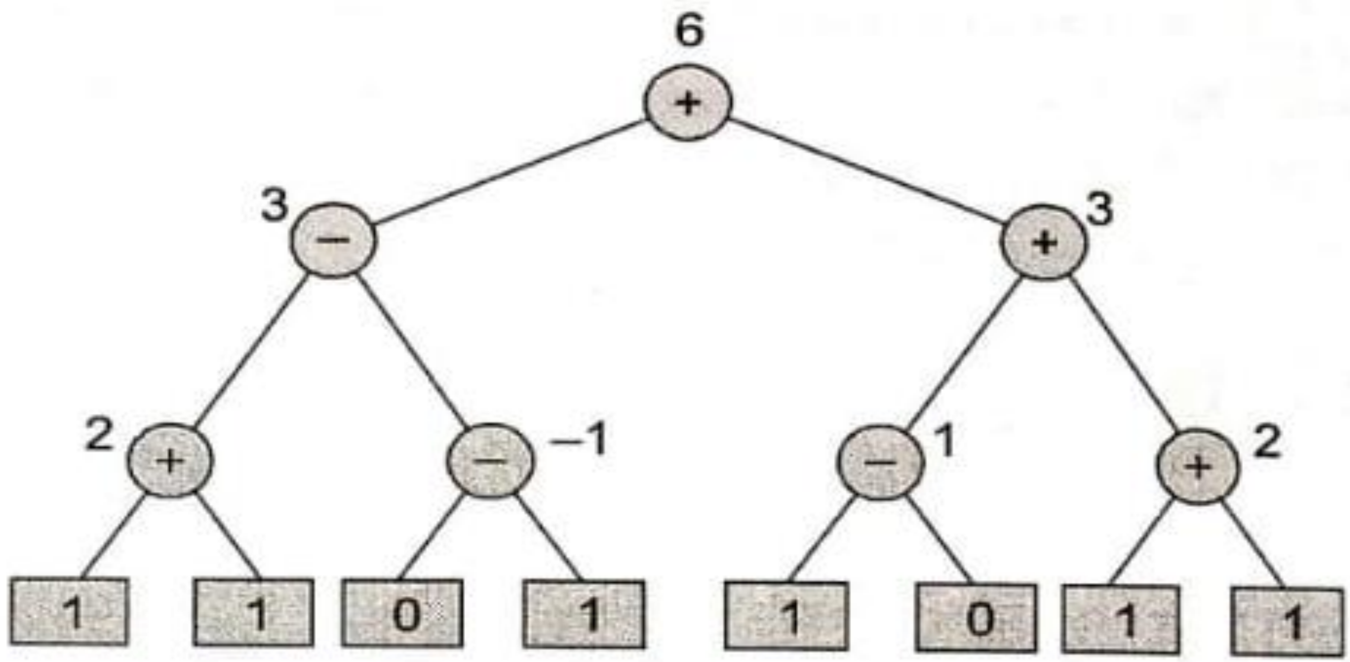
Question 12

3.15 Consider the expression tree shown. Each leaf represents a numerical value, which can either be 0 or 1. Over all possible choices of the values at the leaves, the maximum possible value of the expression represented by the tree is ________.



[2014 (Set-2) : 2 Marks]

3.15 Solution: (6)



Question 13

One of the purposes of using intermediate code in compilers is to

(a) make parsing and semantic analysis simpler.

(b) improve error recovery and error reporting.

(c) increase the chances of reusing the machine-independent code optimizer in other compilers.

(d) improve the register allocation.

[2014 (Set-3) : 1 Mark]

(c)

Intermediate code can be optimized using the machine-independent code optimizers. All compilers can use same machine independent code optimizers to optimize the intermediate code.

THANK YOU