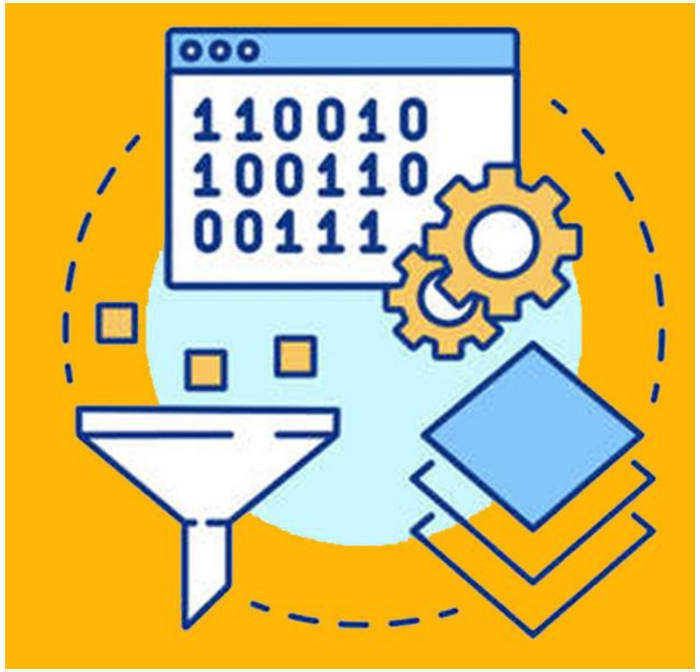
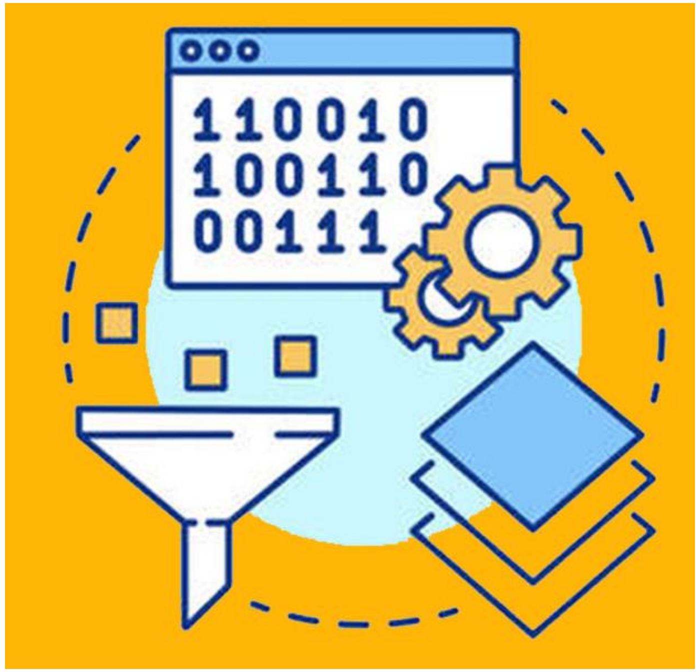
Compiler Design



Syntax Directed Translation



Prof. S. D. PADIYA| SSGMCE, Shegaon
padiyasagar@gmail.com

2021-2022 (Spring)

Syllabus: Unit 01

Syntax Directed Translation:

1. Syntax Directed Definitions,
    i. Synthesized Attributes
    ii. Inherited Attributes,
2. Dependency Graphs,
3. Evaluation Orders.

Construction of Syntax Trees:

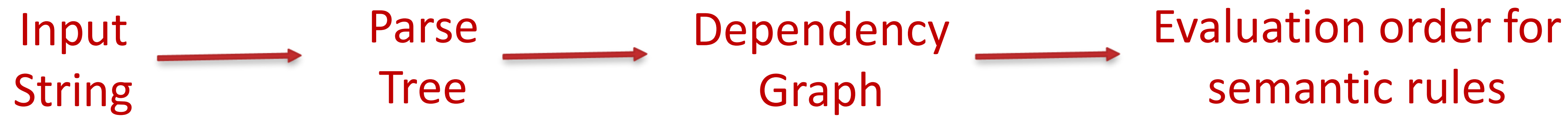
4. Syntax directed definition for constructing syntax trees,
5. Directed acyclic graphs for expressions.
6. Bottom-up Evaluation of S-attributed Definitions,
7. L-attributed Definition.
8. Top-down Translation,
9. Design of a Predictive Translator.

SYNTAX DIRECTED TRANSLATION

For associating semantic rules with productions, there are two notations:

1) Syntax-directed definitions
2) Translation Schemes

Conceptually, with both **syntax-directed definitions** and **translations** schemes, we parse the input token stream, build the parse tree, and then traverse the tree as needed to evaluate the semantic rules at the parse-tree nodes.



Input String → Parse Tree → Dependency Graph → Evaluation order for semantic rules

1) SYNTAX DIRECTED DEFINITION

Syntax directed definition is a generalization of context free grammar in which each grammar symbol has an associated set of attributes.

It is partitioned into two subsets called (Types of attributes)

1. Synthesized attribute
2. Inherited attribute

An attributes can be anything we choose: a value, a type, a memory location, a return type, a string, or whatever.

E. Type
E. Value
E. Memory Location
E. Return Type

1) SYNTAX DIRECTED DEFINITION

The value of an attribute at a parse tree node is defined by a semantic rule associated with the production used that node.

The value of a synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree;

The value of an inherited attributes is computed from the values of attributes at the siblings and parent of that node.

Semantic rules set up dependencies between attributes that will be represented by a **graph**. From the dependencies graph, we derive an evaluation order for the semantic rules.

Evaluation of the semantic rules defines the values of the attributes at the node in the parse tree for the input string.

1) SYNTAX DIRECTED DEFINITION

A parse tree showing the values of attributes at each node is called an annotated Parse Tree.

The process of computing the attributes values at the nodes is called annotating or decorating the parse tree.

Example-1: Syntax directed definition of simple desk calculator.

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) SYNTAX DIRECTED DEFINITION

Applications:

1) Executing Arithmetic Expressions
2) Conversion from INFIX to POSTFIX
3) Conversion from INFIX to PREFIX
4) Conversion from Binary to Decimal
5) Conversion from Decimal to Binary
6) Counting number of reductions
7) Creating syntax tree
8) Generating Intermediate Code
9) Type Checking
10) Storing Type Information into Symbol Table

1.1) SYNTHESIZED ATTRIBUTE

Synthesized attributes are used extensively in practice, It is bottom-up parsing from leaves to the root. A syntax-directed definition that uses synthesized attributes exclusively is said to be an S-attributed definition.

In syntax-directed definition, terminals are assumed to have synthesized attributes only, as the definition does not provide any semantic rules for terminals.

Value of synthesized attribute at a node can be computed from the value of attributes at the children of that node in the parse tree.

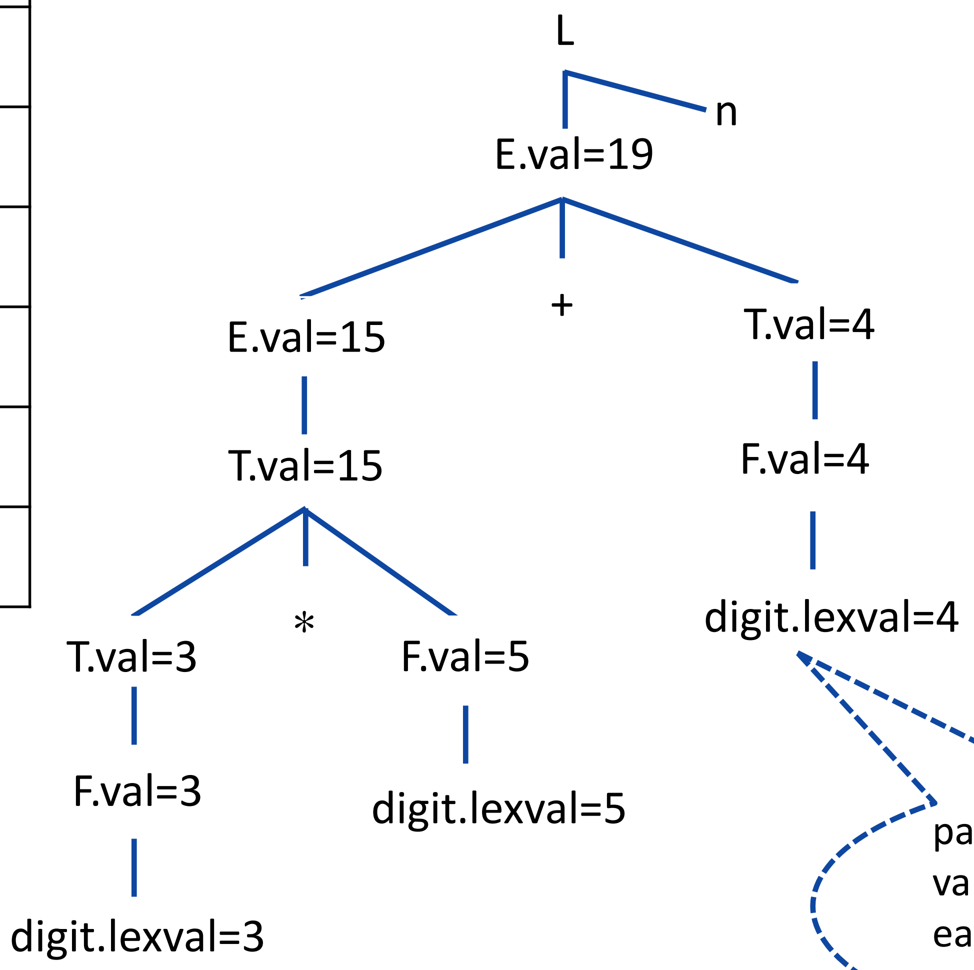
Values for attributes of terminals are usually supplied by the lexical analyzer and start symbol is assumed not to any inherited attributes, unless otherwise stated.

1.1) SYNTHESIZED ATTRIBUTE

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

String: 3*5+4n

Annotated parse tree for 3*5+4n



The process of computing the attribute values at the node is called annotating or decorating the parse tree

parse tree showing the value of the attributes at each node is called Annotated parse tree

1.1) SYNTHESIZED ATTRIBUTE

Example-2: Draw Annotated Parse tree for following:

1. 7+3*2n

2. 7*3+2n

3. (3+4)*(5+6)n

1.2) INHERITED ATTRIBUTE

An inherited value at a node in a parse tree is computed from the value of attributes at the parent and/or siblings of the node.

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)

Syntax directed definition with inherited attribute L.in

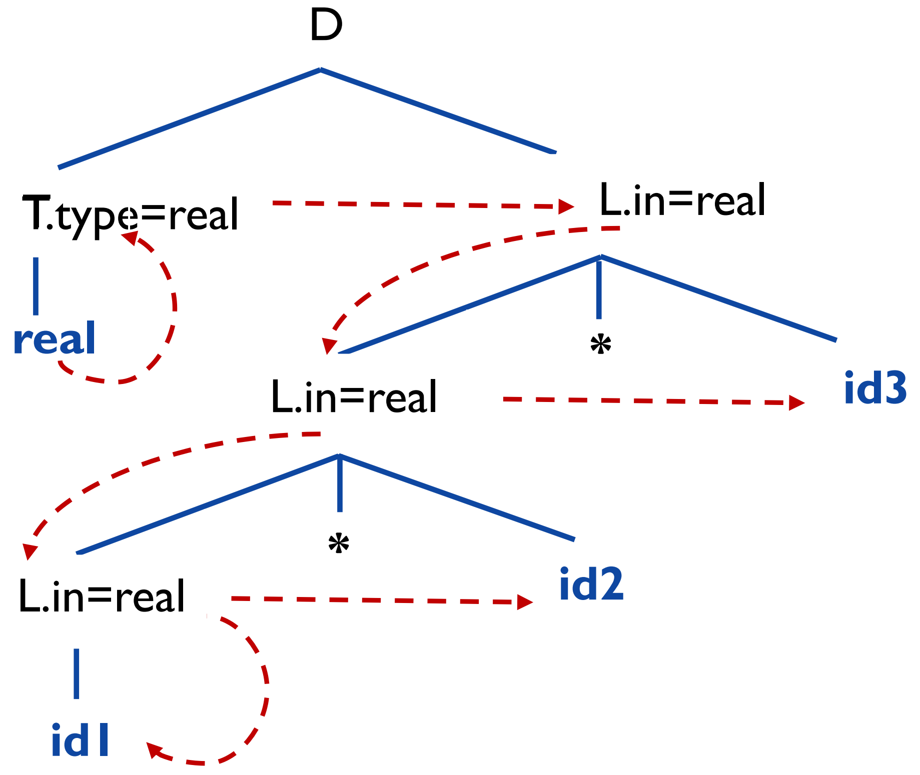
Symbol T is associated with a synthesized attribute type.

Symbol L is associated with an inherited attribute in.

1.2) INHERITED ATTRIBUTE

Example: Pass data types to all identifier real id1 * id2 * id3

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)



D → T L
L → L, id

2) DEPENDENCY GRAPHS

If an attributes b at a node in a parse tree depends on an attributes c, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c.

The interdependencies among the inherited and synthesized attributes a the nodes in a parse tree can be depicted by a directed graph called a ***dependency graphs***.

For the rule X→YZ the semantic action is given by X.x = f(Y.y, Z.z) then synthesized attribute X.x depends on attributes Y.y and Z.z.

The basic idea behind dependency graphs is for a compiler to look for various kinds of dependency among statements to prevent their execution in wrong order.

2) DEPENDENCY GRAPHS

Algorithm

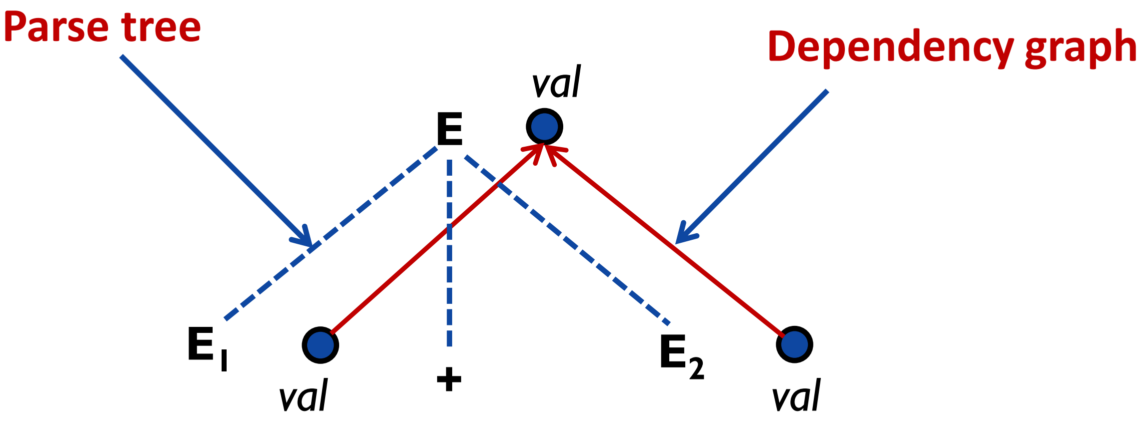
```
for each node n in the parse tree do
        for each attribute a of the grammar symbol at n do
                Construct a node in the dependency graph for a;

for each node n in the parse tree do
        for each semantic rule $b=f(c_1,c_2,.....,c_k)$
              associated with the production used at n do
            for i=1 to k do
              construct an edge from the node for $C_i$ to the node for b;
```

2) DEPENDENCY GRAPHS

Example:

E→E_1+E_2

Production	**Semantic rules**
E → E_1 + E_2	E.val = E_1.val + E_2.val



$E.val$ **is synthesized from** $E1.val$ **and** $E2.val$

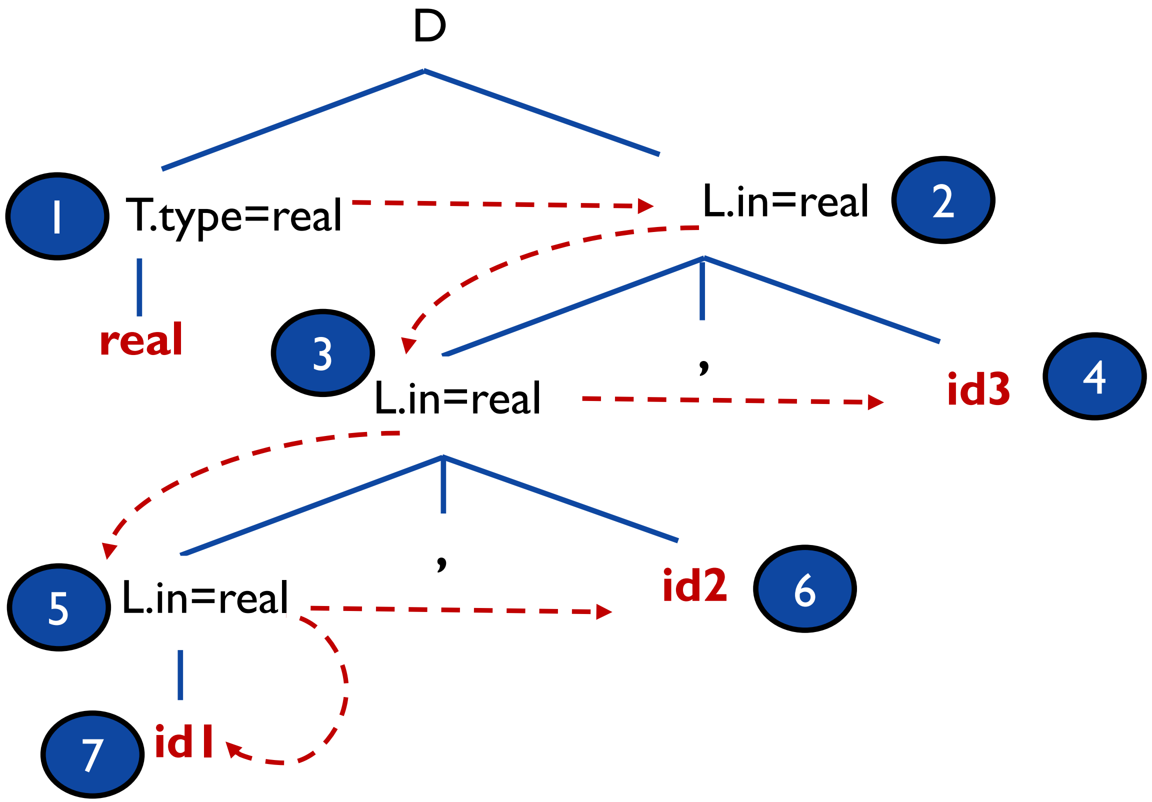
The edges to E.val from E_1.val and E_2.val shows that E.val is depends on E_1.val and E_2.val.

3) EVALUATION ORDER

A topological sort of a directed acyclic graph is any ordering $m_1, m_2, \dots \dots \dots .., m_k$ of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes.

If $m_i \rightarrow m_j$ is an edge from m_i to m_j then m_i appears before m_j in the ordering.

Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with the nodes in a parse tree can be evaluated.



CONSTRUCTION OF SYNTAX TREE

In this section, we show how syntax-directed definitions can be used to specify the construction of syntax trees and other graphical representations of language constructs.

The construction of syntax tree for an expression is similar to the translation of the expression into postfix form. We construct subtrees for the subexpressions by creating a node for each operator and operands.

The children of an operator node are the roots of the nodes representing the subexpressions constituting the operands of that operator.

Each node in a syntax tree can be implemented as a record with several fields. In the node for an operator one filed identifies the operator and the remaining fields contain pointers to the nodes for the operands.

CONSTRUCTION OF SYNTAX TREE

Following functions are used to create the nodes of the syntax tree.

1. **Mknode (op, left, right):** creates an operator node with label op and two fields containing pointers to left and right.

2. **Mkleaf (id, entry):** creates an identifier node with label id and a field containing entry, a pointer to the symbol table entry for the identifier.

3. **Mkleaf (num, val):** creates a number node with label num and a field containing val, the value of the number.

CONSTRUCTION OF SYNTAX TREE

Example 1: Construct syntax tree for **a – 4 + c**

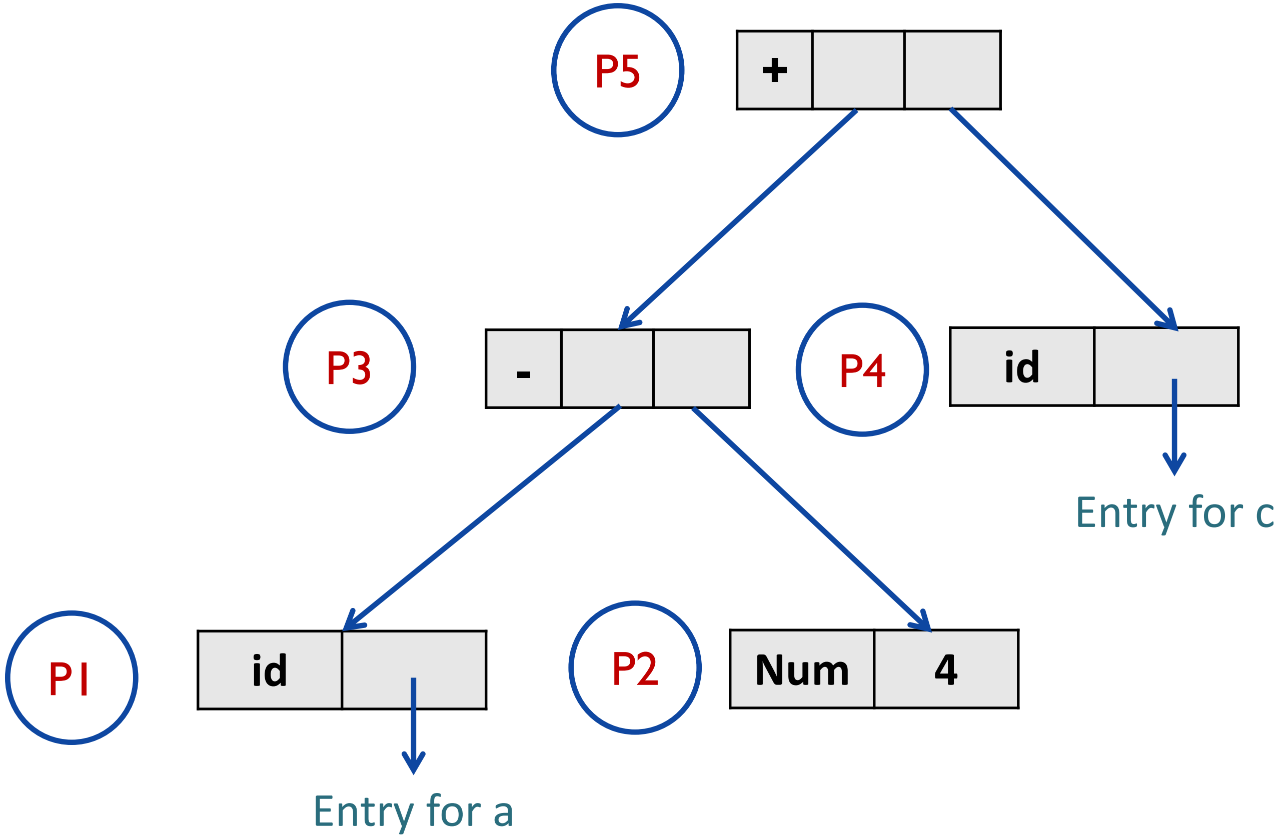
P1: mkleaf(id, entry for a);

P2: mkleaf(num, 4);

P3: mknode(‘-‘, p1, p2);

P4: mkleaf(id, entry for c);

P5: mknode(‘+’, p3, p4);



CONSTRUCTION OF SYNTAX TREE

Example 2: Construct syntax tree for **x + 9 * y**

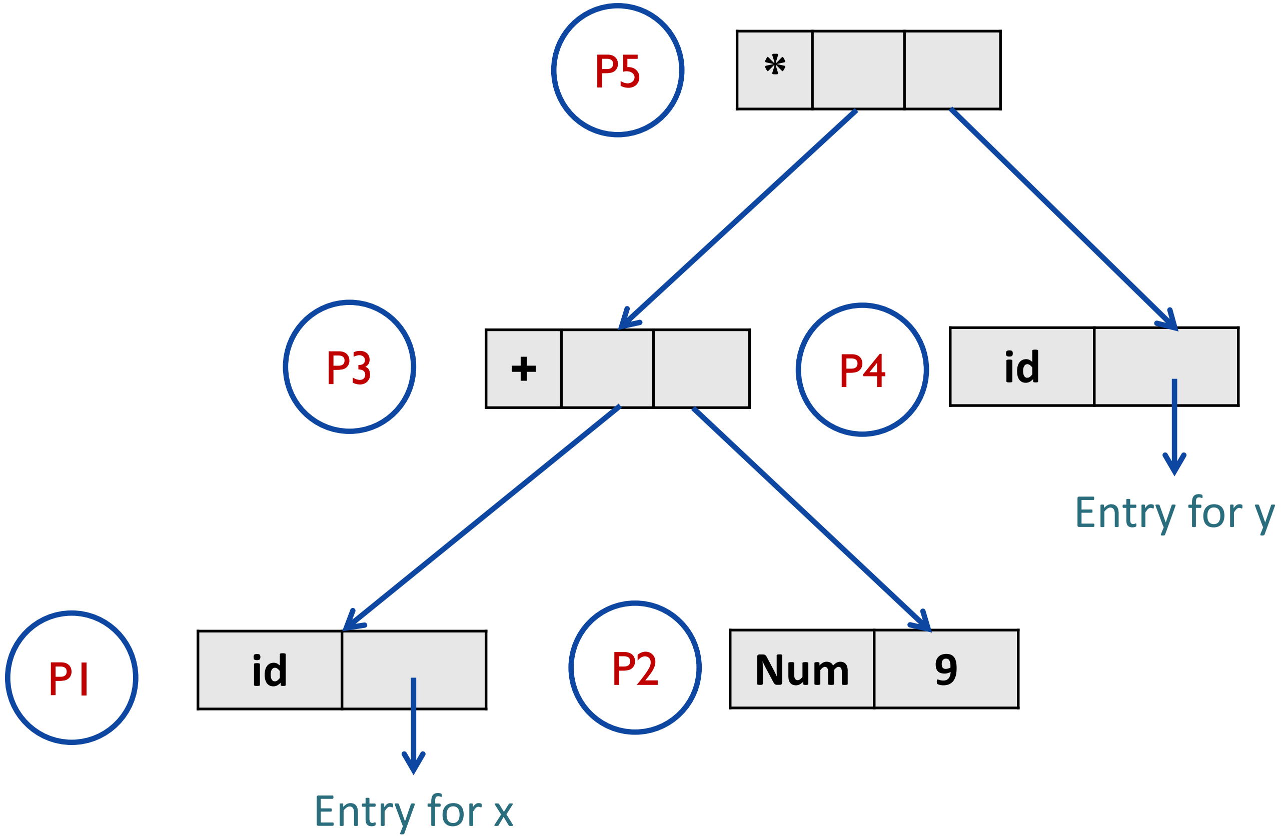
P1: mkleaf(id, entry for x);

P2: mkleaf(num, 10);

P3: mknode(‘+‘, p1, p2);

P4: mkleaf(id, entry for y);

P5: mknode(‘*’, p3, p4);



4) A SYNTAX-DIRECTED DEFINITION FOR CONSTRUCTING SYNTAX TREES

Production	**Semantic Rules**
$E \rightarrow E_1 + T$	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.$_{entry}$)
$T \rightarrow$ num	T.nptr := mkleaf(num, num.$_{val}$)

Above table contains an S-attributed definition for constructing a syntax tree for an expression containing the operators + and -.

It uses the underlying productions of the grammar to schedule the calls of the functions ***mknode*** and ***mkleaf*** to construct the tree.

The synthesized attribute ***nptr*** for E and T keeps track of the pointers returned by the function calls.

4) A SYNTAX-DIRECTED DEFINITION FOR CONSTRUCTING SYNTAX TREES

An annotated parse tree depicting the construction of a syntax tree for the expression **a – 4 + c**.

The parse tree is shown dotted. The parse-tree nodes labeled by the non-terminals **E** and **T** use the synthesized attributes ***nptr*** to hold a pointer to the syntax-tree node for the expression represented by the non-terminal.

The semantic rules associated with the productions **T → id** and **T → num** define attributes **T.nptr** to be a pointer to a new leaf for an identifier and a number respectively.

Attributes **id.**$_{\textbf{entry}}$ and **num.**$_{\textbf{val}}$ are the lexical values assumed to be returned by the lexical analyzer with the tokens **id** and **num**.

4) A SYNTAX-DIRECTED DEFINITION FOR CONSTRUCTING SYNTAX TREES

Example : Construct syntax tree for **a - 4 + c**

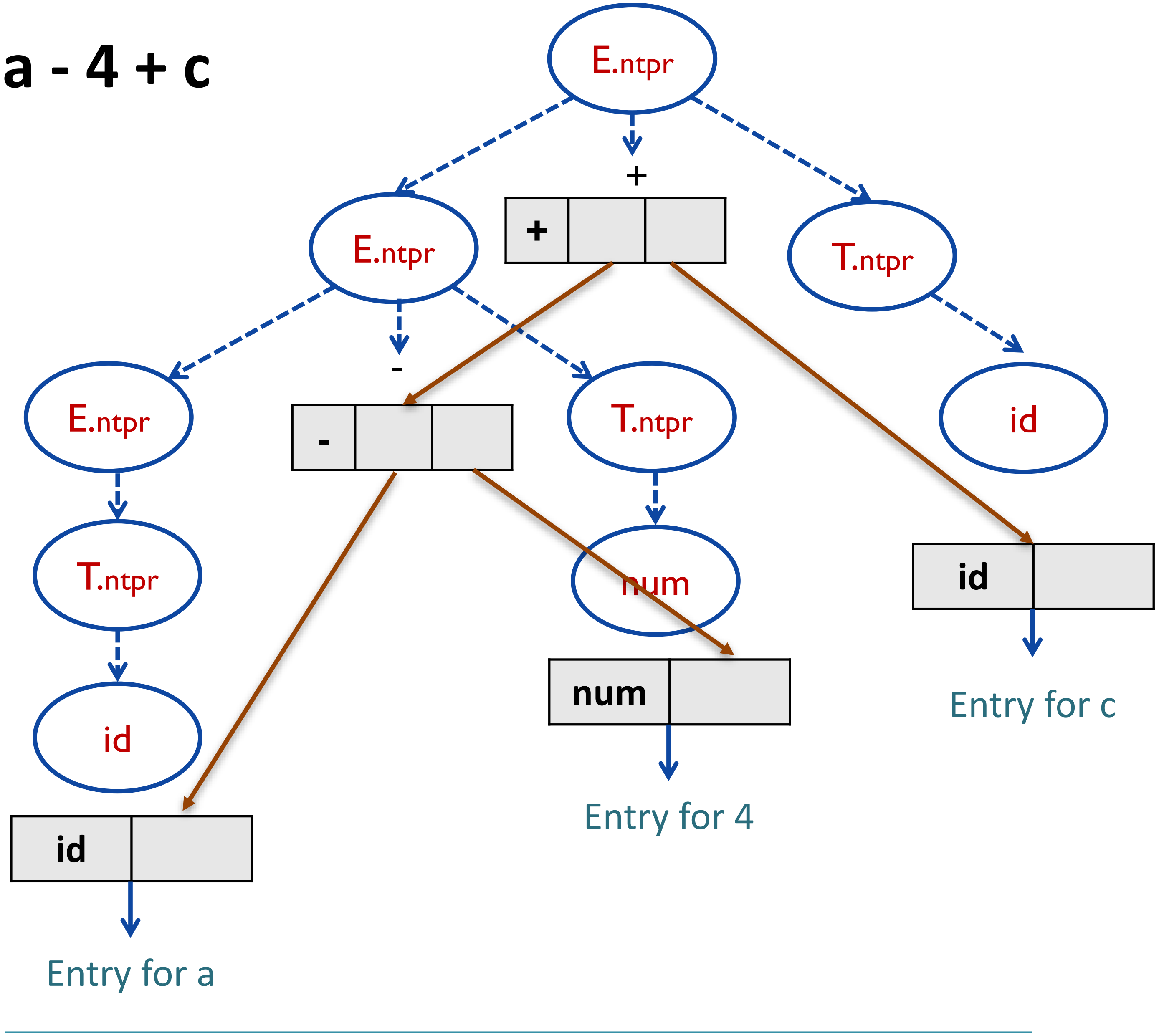
P1: mkleaf(id, entry for a);

P2: mkleaf(num, 4);

P3: mknode('-', P1, P2);

P4: mkleaf(id, entry for c);

P5: mknode('+', P3, P4);



5) DIRECTED ACYCLIC GRAPHS FOR EXPRESSIONS

The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as the result of an intermediate code generation.

Directed acyclic graphs are a type of data structure and they are used to apply transformations to basic blocks.

The Directed Acyclic Graph (DAG) facilitates the transformation of basic blocks.

DAG is an efficient method for identifying common sub-expressions.

It demonstrates how the statement’s computed value is used in subsequent statements.

5) DIRECTED ACYCLIC GRAPHS FOR EXPRESSIONS

A directed acyclic graph (***dag***) for an expression identifies the common subexpressions in the expression.

Like a syntax tree, a ***dag*** has a node for every subexpression of the expression; an interior node represents an operator and its children represent its operands.

The difference is that a node in a ***dag*** representing a common subexpression has more than one “parent;” in a syntax tree, the common subexpression would be represented as a duplicated subtree.

5) DIRECTED ACYCLIC GRAPHS FOR EXPRESSIONS

Construction of DAGs- Following rules are used for the construction of DAGs-

Rule-01:

In a DAG,

- Interior nodes always represent the operators.
- Exterior nodes (leaf nodes) always represent the names, identifiers or constants.

Rule-02:

While constructing a DAG,

- A check is made to find if there exists any node with the same value.
- A new node is created only when there does not exist any node with the same value.
- This action helps in detecting the common sub-expressions and avoiding the re-computation of the same.

Rule-03:

- The assignment instructions of the form x:=y are not performed unless they are necessary.

5) DIRECTED ACYCLIC GRAPHS FOR EXPRESSIONS

Example 1: a + a * (b + c) + (b - c) * d

Sequence of Instructions

P1 := mkleaf(id, b)

P2 := mkleaf(id, c)

P3 := mknode(‘+’, P1, P2)

P4 := mkleaf(id, b)

P5 := mkleaf(id, c)

P6 := mknode(‘-’, P4, P5)

P7 := mkleaf(id, a)

P8 := mknode(‘*’, P3, P7)

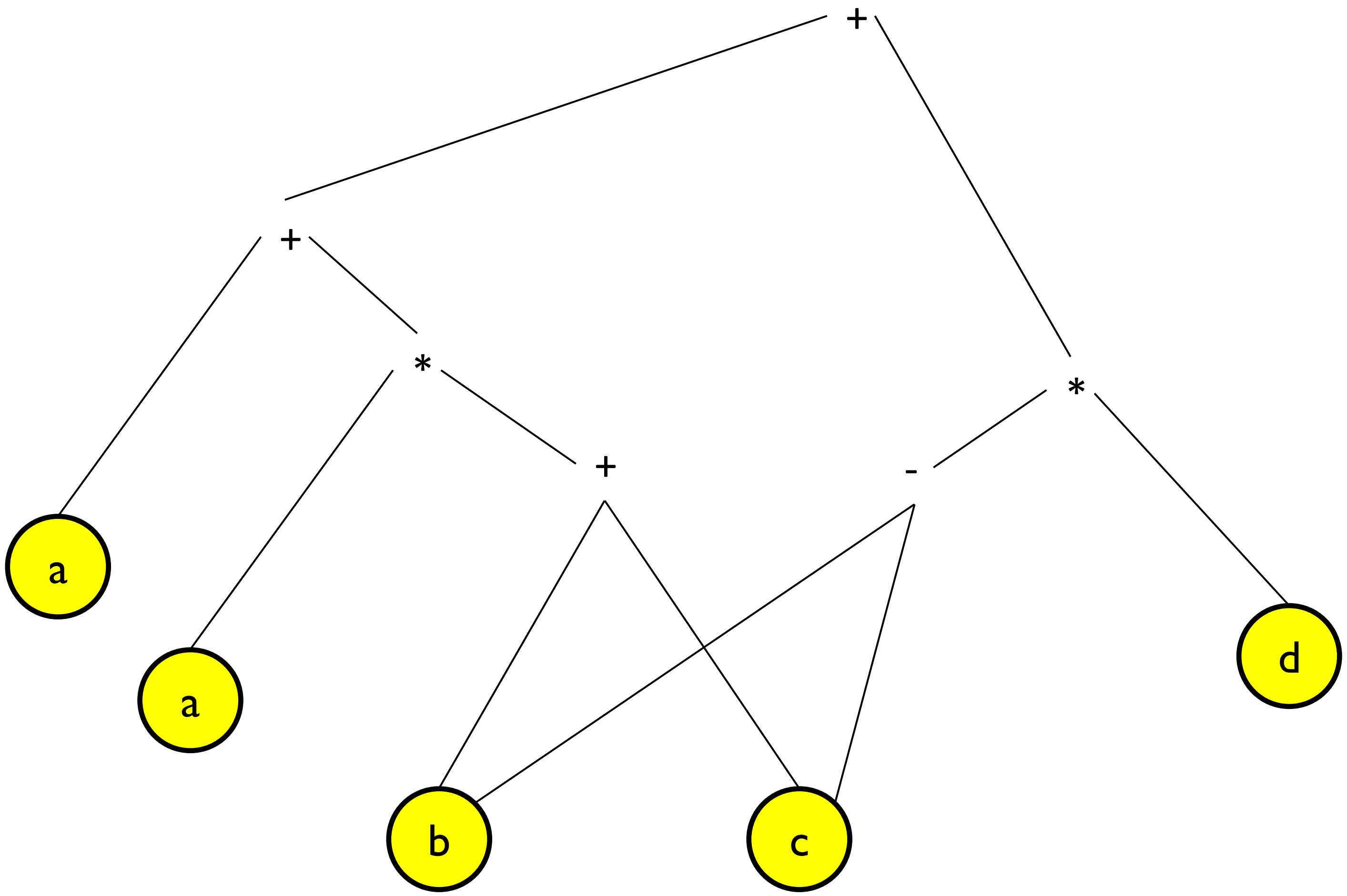
P9 := mkleaf(id, a)

P10 := mknode(‘+’, P8, P9)

P11 := mkleaf(id, d)

P12 := mknode(‘*’, P6, P11)

P13 := mknode(‘+’, P10, P12)



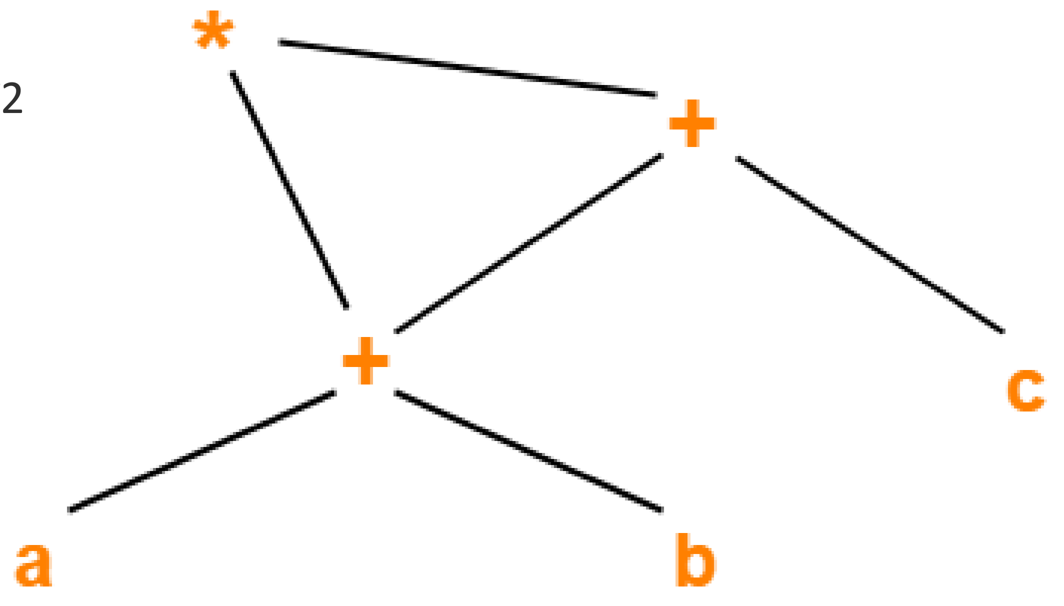
5) DIRECTED ACYCLIC GRAPHS FOR EXPRESSIONS

Example 2: Construct DAG for following expression (a + b) * (a + b + c)

T1 = a + b

T2 = T1 + c

T3 = T1 * T2



Directed Acyclic Graph

Sequence of Instructions

P1 := mkleaf(id, a)

P2 := mkleaf(id, b)

P3 := mknode(‘+’, P1, P2)

P4 := mkleaf(id, a)

P5 := mkleaf(id, b)

P6 := mkleaf(id, c)

P7 := mknode(‘+’, P4, P5)

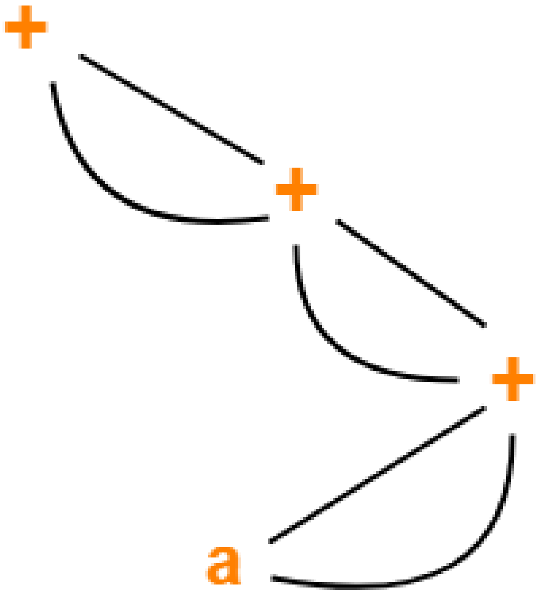
P8 := mknode(‘+’, P7, P6)

P9 := mknode(‘*’, P3, P8)

5) DIRECTED ACYCLIC GRAPHS FOR EXPRESSIONS

Example 3: Construct DAG for following expression

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



Directed Acyclic Graph

P1 := mkleaf(id, a)

P2 := mkleaf(id, a)

P3 := mknode('+', P1, P2)

P4 := mkleaf(id, a)

P5 := mkleaf(id, a)

P6 := mknode('+', P4, P5)

P7 := mknode('+', P3, P6)

P8 := mkleaf(id, a)

P9 := mkleaf(id, a)

P10 := mknode('+', P8, P9)

P11 := mkleaf(id, a)

P12 := mkleaf(id, a)

P13 := mknode('+', P11, P12)

P14 := mknode('+', P10, P13)

P15 := mknode('+', P7, P14)

5) DIRECTED ACYCLIC GRAPHS FOR EXPRESSIONS

Example 4: Construct DAG for following expression

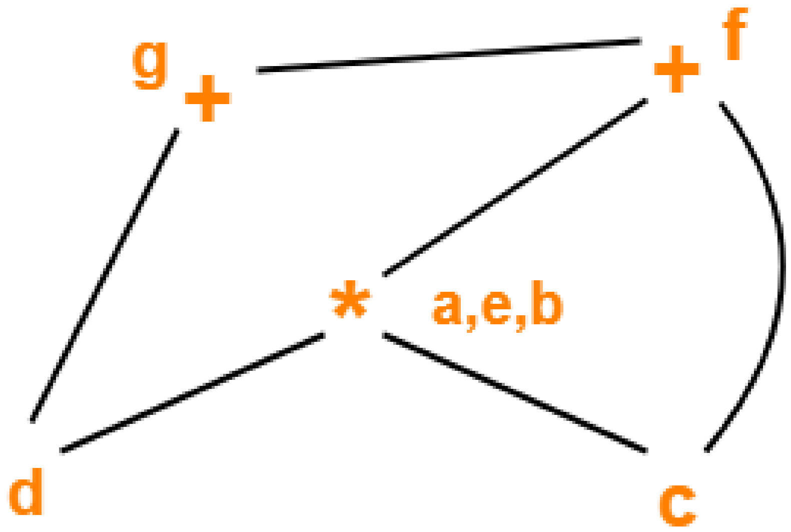
(1) a = b * c

(2) e = d * c

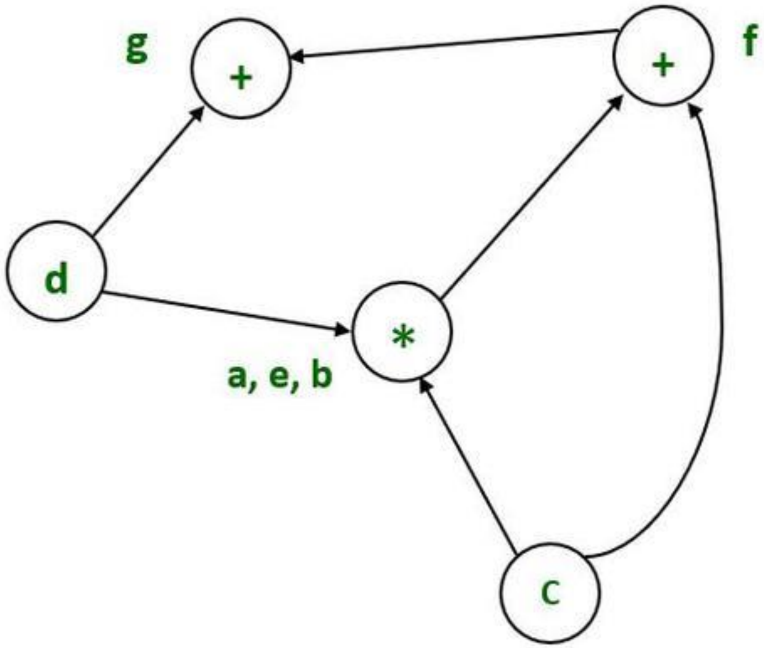
(3) b = e

(4) f = b + c

(5) g = f + d



Directed Acyclic Graph



5) DIRECTED ACYCLIC GRAPHS FOR EXPRESSIONS

Example 5: Construct DAG for following expression

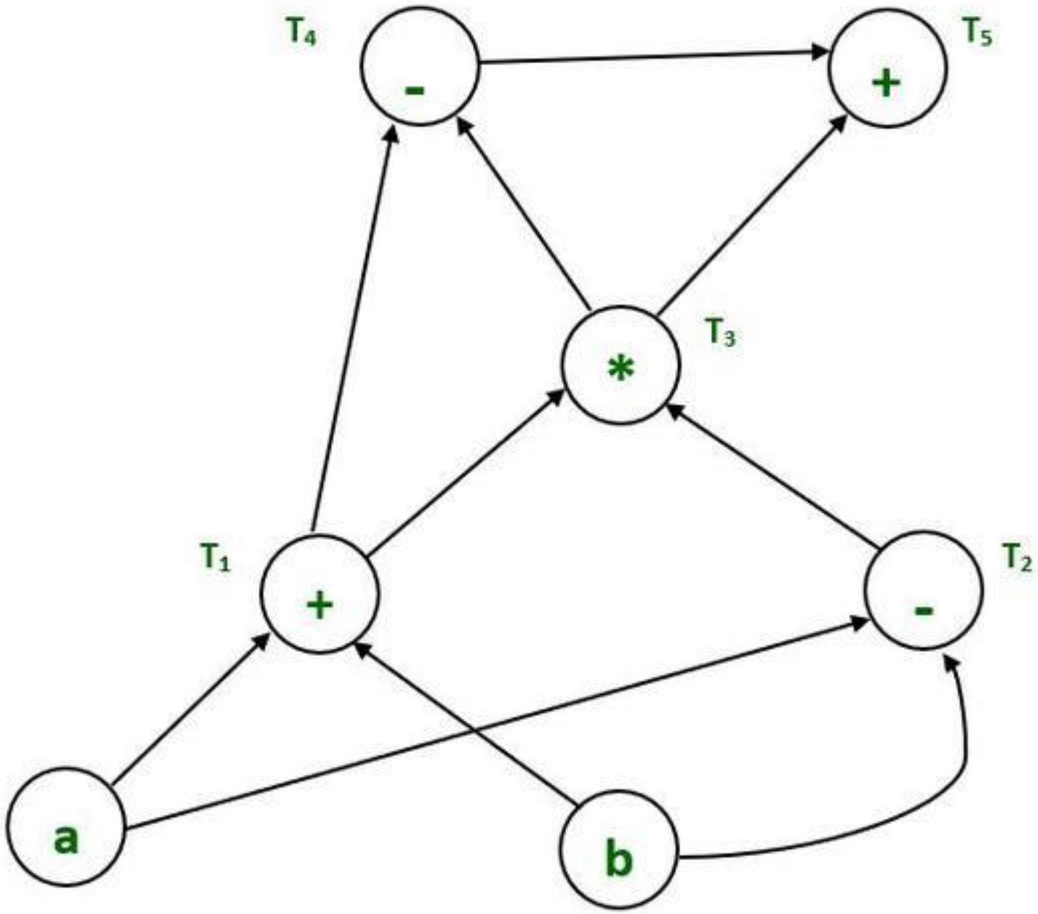
$T_1 = a + b$

$T_2 = a - b$

$T_3 = T_1 * T_2$

$T_4 = T_1 - T_3$

$T_5 = T_4 + T_3$



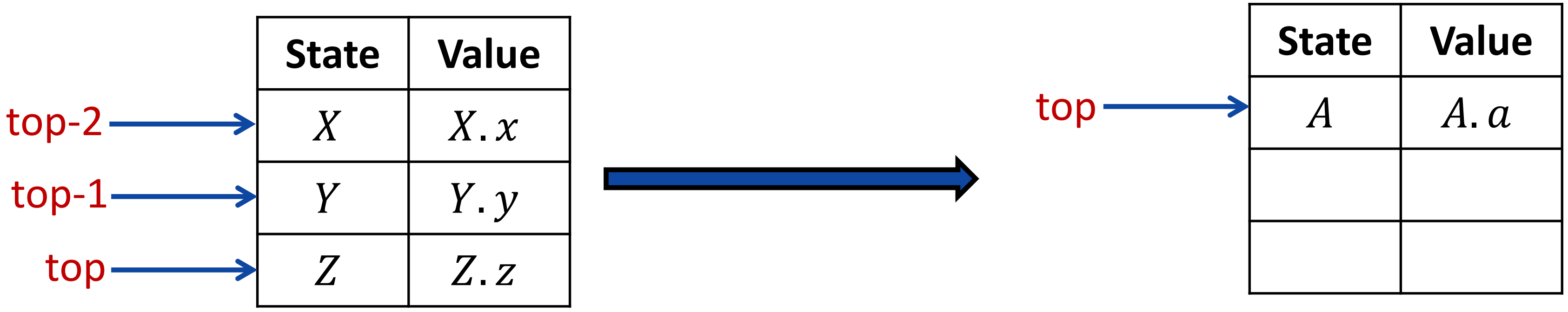
6) BOTTOM UP EVALUATION OF S-ATTRIBUTED DEFINITIONS

S-attributed definition is one such class of syntax directed definition with synthesized attributes only.

Synthesized attributes can be evaluated using bottom up parser only.

Synthesized attributes on the parser stack

Consider the production A→XYZ and associated semantic action is A.a=f(X.x, Y.y, Z.z)



State	Value
X	$X.x$
Y	$Y.y$
Z	$Z.z$

Before reduction

State	Value
A	$A.a$

After reduction

6) BOTTOM UP EVALUATION OF S-ATTRIBUTED DEFINITIONS

Production	Semantic rules
L → E_n	Print (val[top])
E → E_1+T	val[top]=val[top-2] + val[top]
E → T	
T → T_1*F	val[top]=val[top-2] * val[top]
T → F	
F → (E)	val[top]=val[top-2] - val[top]
F → digit	

Implementation of a desk calculator with bottom up parser

Input	State	Val	Production Used
3*5**n**	-	-	
*5**n**	3	3	
*5**n**	F	3	F→digit
*5**n**	T	3	T→F
	T*	3	
n	T*5	3,5	
	T*F	3,5	F→digit
n	T	15	T→T_1*F
n	E	15	E→T
	E**n**	15	
	L	15	L → E_n

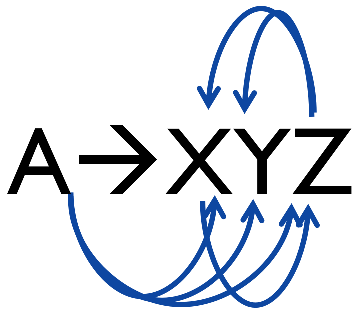
Move made by translator

7) L-ATTRIBUTE DEFINITION

A syntax directed definition is L-attributed if each inherited attribute of X_j, $1 <= j <= n$, on the right side of $A \rightarrow X_1, X_2 \ldots X_n$ depends only on:

1. The attributes of the symbols $X_1, X_2, \ldots X_{j\text{-}1}$ to the left of X_j in the production and
2. The inherited attribute of A.

Example:



Not L- Attributed ✗

L- Attributed ✓

7) L-ATTRIBUTE DEFINITION

Production	Semantic Rules

A→ LM	L.i:=l(A.i)
	M.i=m(L.s)
	A.s=f(M.s)

A→ QR	R.i=r(A.i)
	Q.i=q(R.s) A.s=f(Q.s)

Above syntax directed definition is *not L-attributed* because the inherited attribute Q.i of the grammar symbol Q depends on the attribute R.s of the grammar symbol to its right.

Thank You

Prof. S. D. Padiya
padiyasagar@gmail.com
SSGMCE, Shegaon