S J P N Trust's

HIRASUGAR INSTITUTE OF TECHNOLOGY, NIDASOSHI.

Inculcating Values, Promoting Prosperity

Approved by AICTE, Recognized by Govt. of Karnataka and Permanently Affiliated to VTU Belagavi.

Accredited at 'A' Grade by NAAC

Programmes Accredited by NBA: CSE, ECE, EEE & ME

Subject: System Software and Compilers (18CS61)

Module 5: Syntax Directed Translation, Intermediate code generation, Code generation

Dr. Mahesh G. Huddar

Dept. of Computer Science and Engineering

Syntax-Directed Definitions

- A *syntax-directed definition* (SDD) is a context-free grammar together with, attributes (values) and rules (Semantic rules).
- Attributes are associated with grammar symbols and rules are associated with productions.
- If **X** is a symbol and **a** is one of its attributes, then we write **X.a** to denote the value of **a** at a particular parse-tree node labeled **X**.
- If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X.
- Attributes may be of any kind: numbers, types, table references, or strings, for instance.
- The strings may even be long sequences of code, say code in the intermediate language used by a compiler.

Syntax-Directed Definitions

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
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)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

Syntax-directed definition of a simple desk calculator

Syntax-Directed Definitions - Inherited and Synthesized Attributes

We shall deal with two kinds of attributes for nonterminals:

1. A ***synthesized attribute*** for a nonterminal *A* at a parse-tree node N is defined by a semantic rule associated with the production at N. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.
   - $A \rightarrow BCD$
   - A.val = B.val or A.val = C.val or A.val = D.val

2. An ***inherited attribute*** for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.
   - $A \rightarrow BCD$
   - C.val = A.val or C.val = B.val or C.val = D.val

Syntax-Directed Definitions – Types of SDD

1. A SDD that uses only synthesized attributes, then such SDD is called as **S-Attributed SDD**.
   - $A \rightarrow BCD$
   - A.val = B.val or A.val = C.val or A.val = D.val

2. A SDD that uses both synthesized and inherited attributes, then such SDD is called as **L-Attributed SDD**. Note: Each inherited attribute is restricted to inherit from parent or left sibling.
   - $A \rightarrow BCD$
   - A.val = B.val or C.val = A.val or C.val = B.val or C.val = D.val (not valid)

Evaluating an SDD at the Nodes of a Parse Tree

- A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree.
- How do we construct an annotated parse tree?
- In what order do we evaluate attributes?
- Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends.
- **If all attributes are synthesized,** then we must evaluate the *val* attributes at all the children of a node before we can evaluate the *val* attribute at the node itself.
- With synthesized attributes, we can evaluate attributes in any bottom-up order.

Evaluating an SDD at the Nodes of a Parse Tree

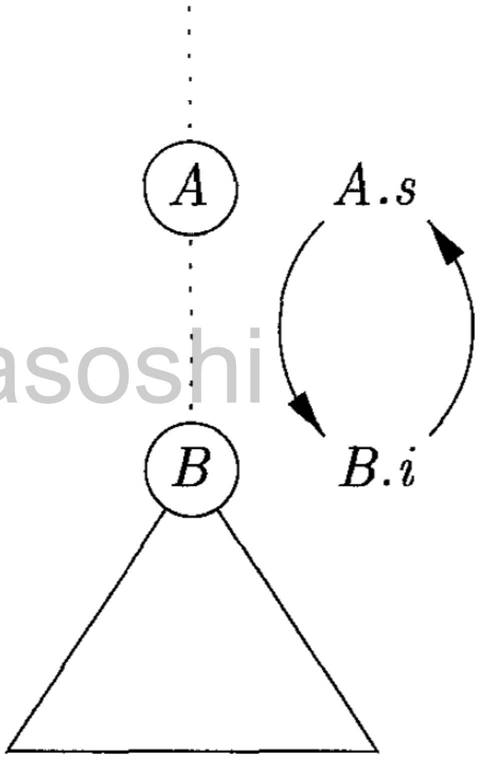
- For SDD's with **both inherited and synthesized attributes**, there is no guarantee that there is even one order in which to evaluate attributes at nodes.
- For instance, consider nonterminals A and B, with synthesized and inherited attributes *A.s* and B.i, respectively, along with the production and rules

PRODUCTION	SEMANTIC RULES
$A \rightarrow B$	$A.s = B.i;$
	$B.i = A.s + 1$

- These rules are circular; it is impossible to evaluate either A.s at a node N or B.i at the child of *N* without first evaluating the other.
- The circular dependency of *A.s* and B.i at some pair of nodes in a parse tree is suggested by Fig.

Evaluating an SDD at the Nodes of a Parse Tree

PRODUCTION	SEMANTIC RULES
$A \to B$	$A.s = B.i;$ $B.i = A.s + 1$

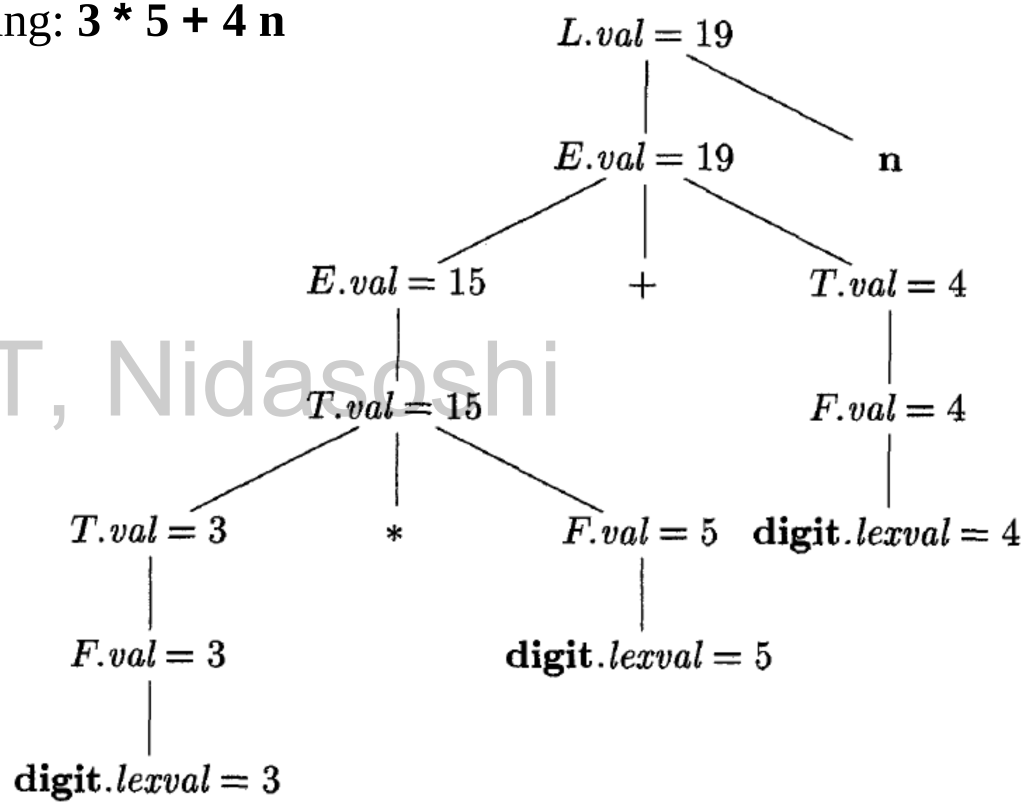


Evaluating an SDD at the Nodes of a Parse Tree

- Example 1: Annotated parse tree for string: **3 * 5 + 4 n**

	PRODUCTION	SEMANTIC RULES
1)	$L \to E\ \mathbf{n}$	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

Syntax-directed definition of a simple desk calculator

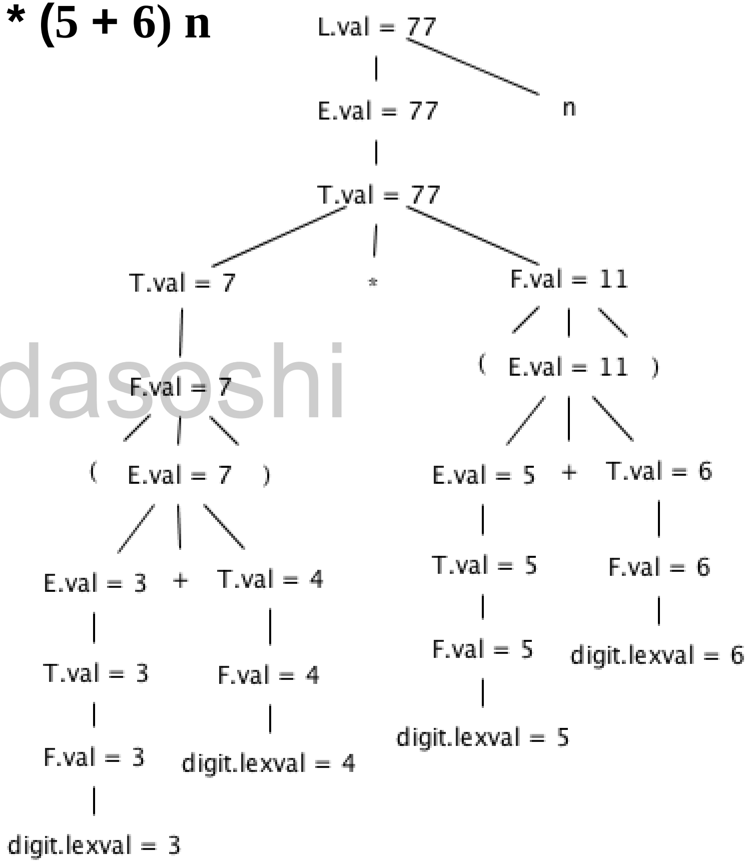


Evaluating an SDD at the Nodes of a Parse Tree

- Example 2: Annotated parse tree for string: **(3 + 4) * (5 + 6) n**

	PRODUCTION	SEMANTIC RULES
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

Syntax-directed definition of a simple desk calculator

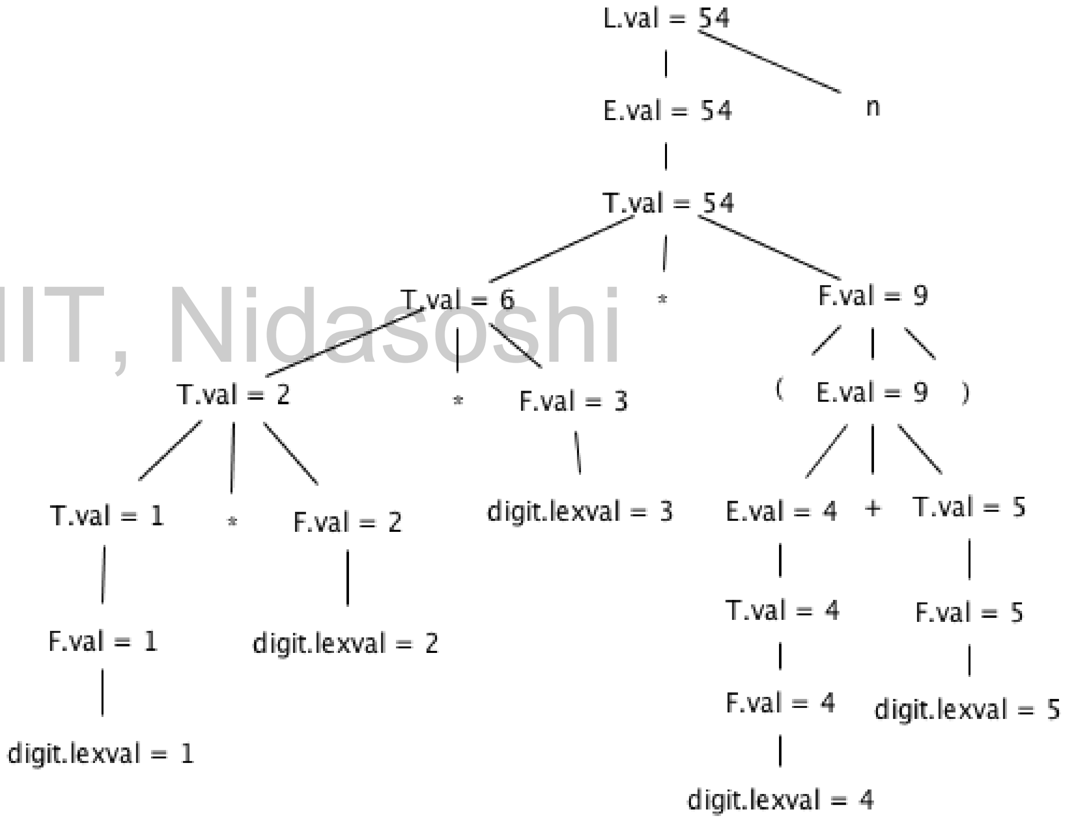


Evaluating an SDD at the Nodes of a Parse Tree

- Example 3: Annotated parse tree for string: **1 * 2 * 3 * (4 + 5) n**

	PRODUCTION	SEMANTIC RULES
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

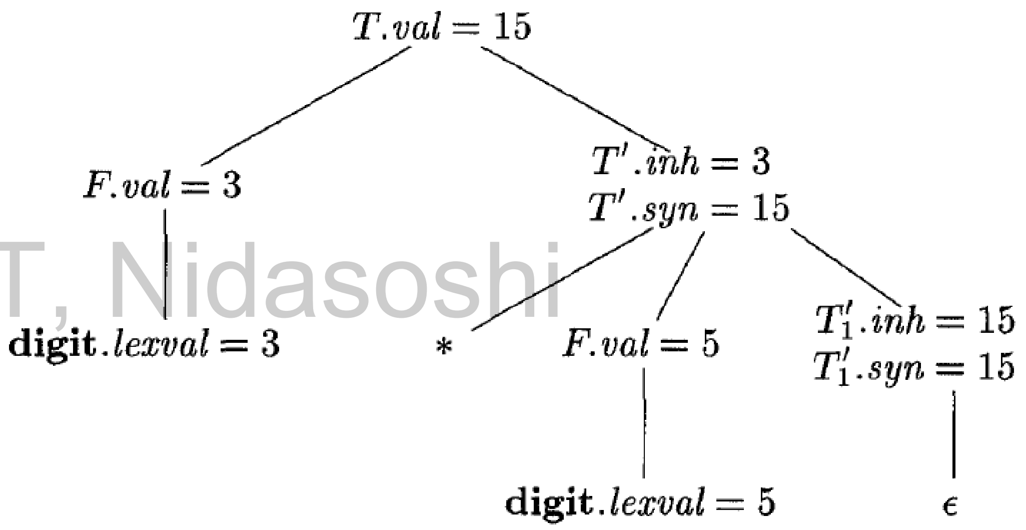
Syntax-directed definition of a simple desk calculator



Evaluating an SDD at the Nodes of a Parse Tree

- Example 2: Annotated parse tree for string: **3 * 5**

	Production	Semantic Rules
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \textbf{digit}$	$F.val = \textbf{digit}.lexval$



Evaluating an SDD at the Nodes of a Parse Tree

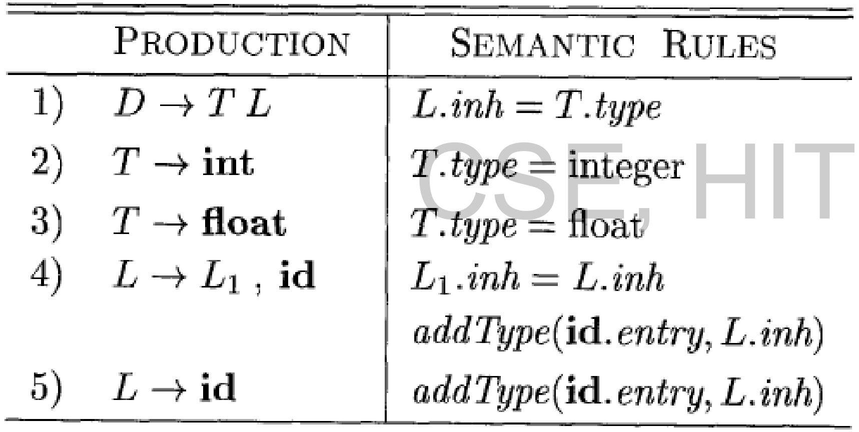
- Example 2: Annotated parse tree for string: **3 * 5**
- The top-down parse of input 3 * 5 begins with the production T $\rightarrow$ F T'.
- Here, F generates the digit 3, but the operator * is generated by T'.
- Thus, the left operand 3 appears in a different subtree of the parse tree from *.
- An inherited attribute will therefore be used to pass the operand to the operator.

Evaluating an SDD at the Nodes of a Parse Tree

- Example 2: Annotated parse tree for string: **3 * 5 * 7**

Evaluating an SDD at the Nodes of a Parse Tree

- Example 3: Annotated parse tree for string: **int a, b, c**



	Production	Semantic Rules
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow$ **int**	$T.type =$ integer
3)	$T \rightarrow$ **float**	$T.type =$ float
4)	$L \rightarrow L_1$, **id**	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5)	$L \rightarrow$ **id**	$addType(\mathbf{id}.entry, L.inh)$

Evaluation Orders for SDD's

- "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.

Evaluation Orders for SDD’s - Dependency Graphs

- ***A dependency graph*** depicts the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second.
- Edges express constraints implied by the semantic rules.

1. For each parse-tree node, say a node labeled by grammar symbol X, the dependency graph has a node for each attribute associated with ***X*** **.**
2. Suppose that a semantic rule associated with a production *p* 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.***
3. Suppose that a semantic rule associated with a production *p* defines the value of inherited attribute ***B.c*** in terms of the value of *X.a*. Then, the dependency graph has an edge from *X.a* to ***B.c.*** For each node N labeled B that corresponds to an occurrence of this B in the body of production *p*, create an edge to attribute c at N from the attribute *a* at the node *Ad* that corresponds to this occurrence of X. Note that M could be either the parent or a sibling of N.

Evaluation Orders for SDD's - Dependency Graphs

- **Example 1:** Consider the following production and rule:

PRODUCTION	SEMANTIC RULE
$E \to E_1 + T$	$E.val = E_1.val + T.val$

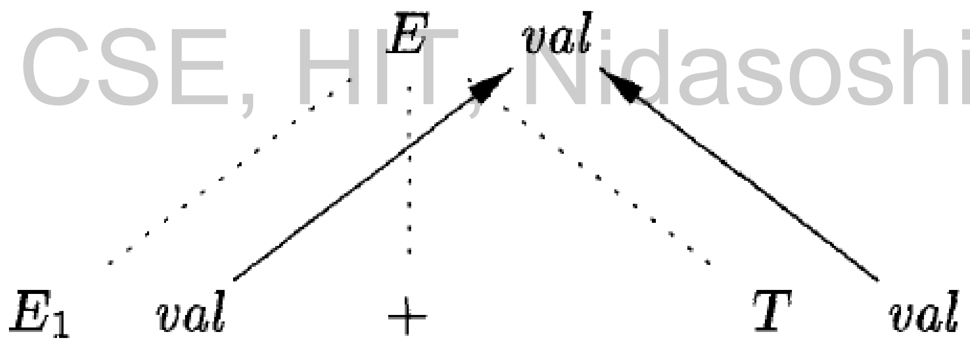
- At every node N labeled E, with children corresponding to the body of this production, the synthesized attribute *val* at N is computed using the values of v*al* at the two children, labeled E and T.

- Thus, a portion of the dependency graph for every parse tree in which this production is used looks like Fig.
- As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid.

Evaluation Orders for SDD's - Dependency Graphs

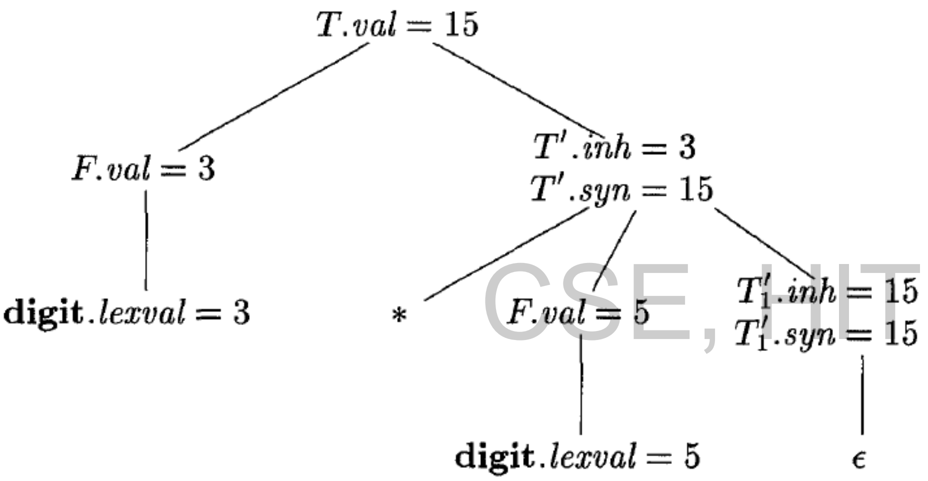
- **Example 1:** Consider the following production and rule:

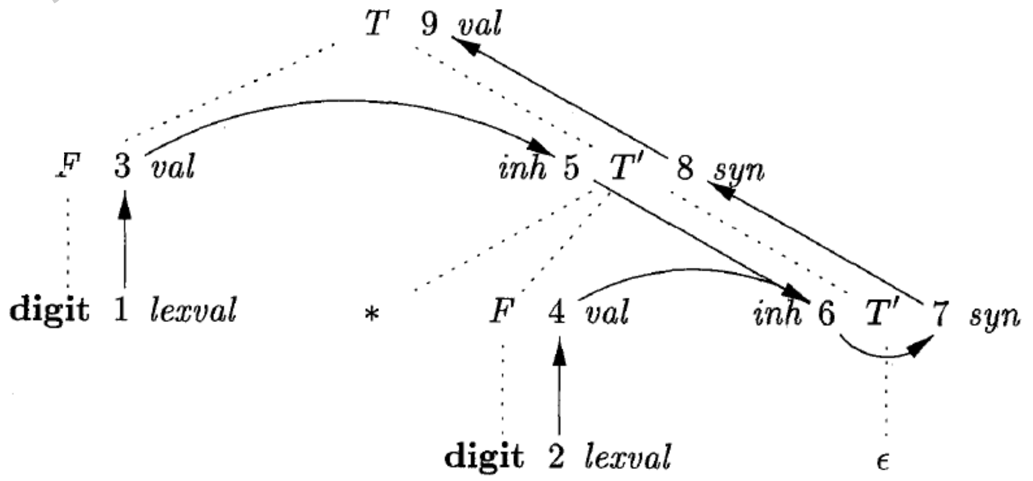
PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$



Evaluating an SDD at the Nodes of a Parse Tree

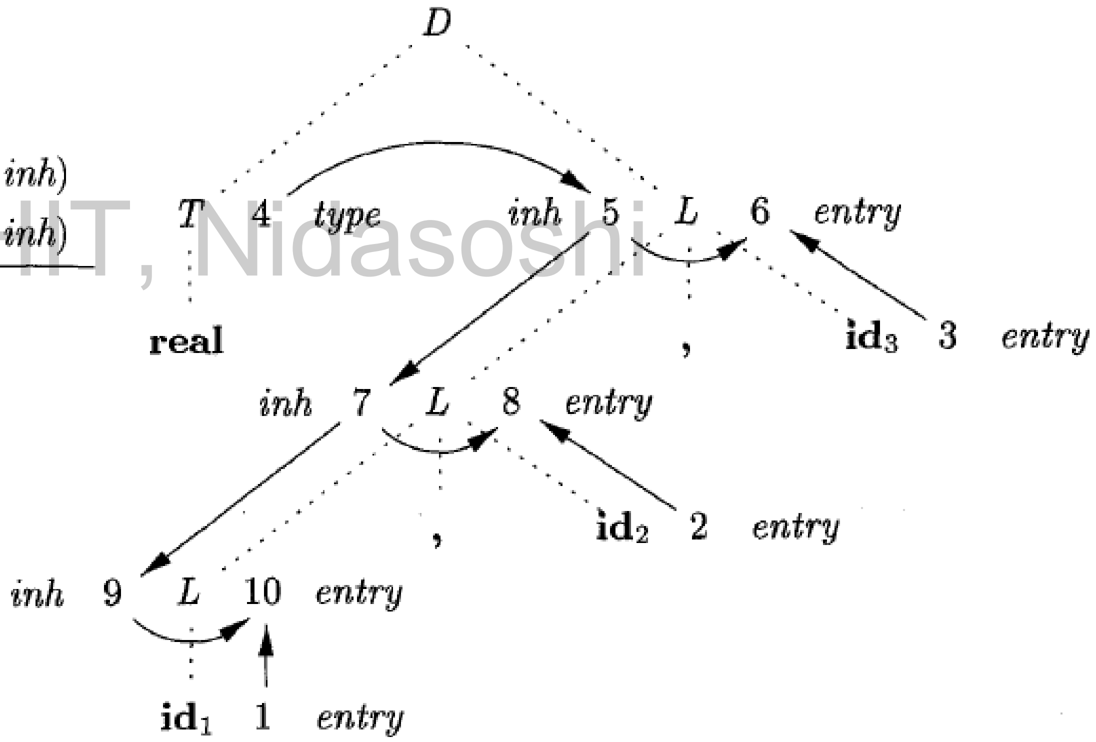
- Example 2: Annotated parse tree for string: **3 * 5**



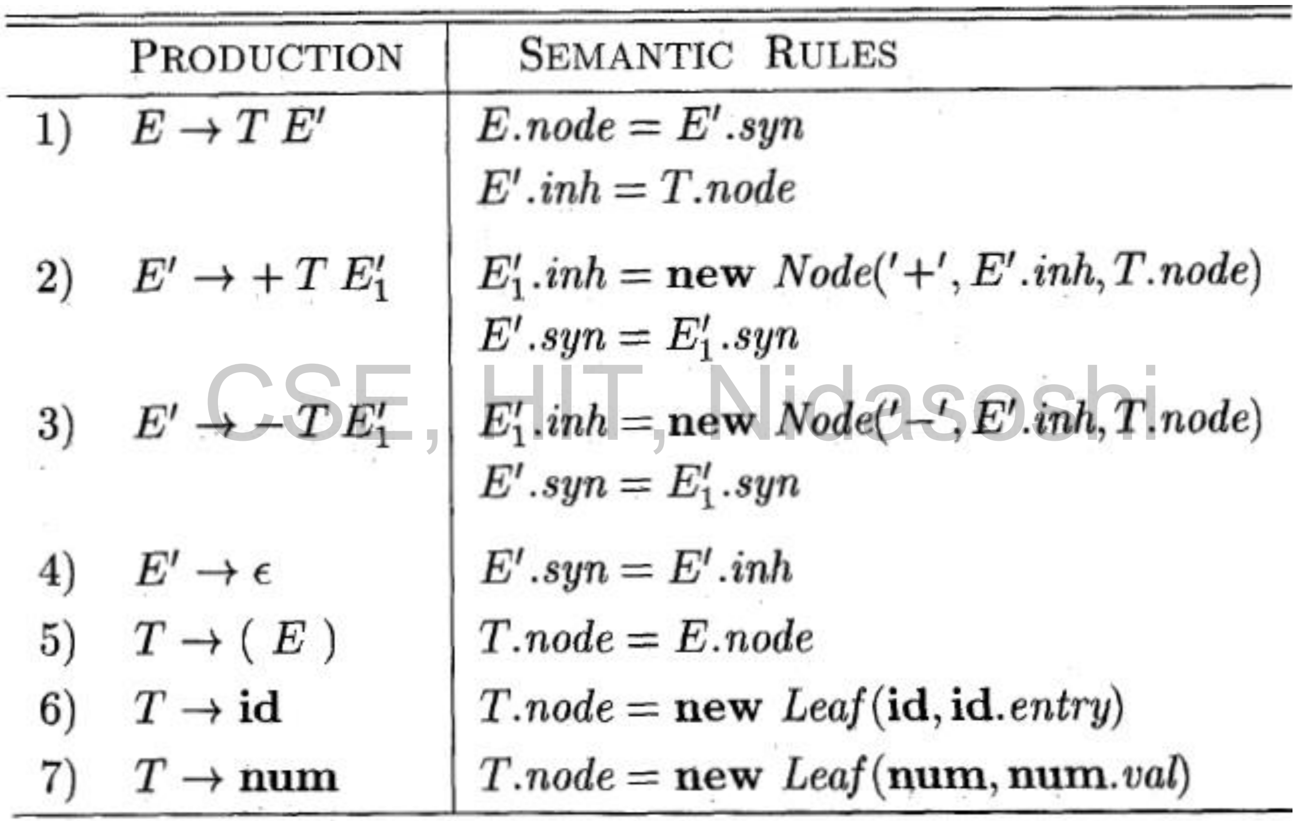


Syntax-directed definition for simple type declarations

	Production	Semantic Rules
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow$ **int**	$T.type =$ integer
3)	$T \rightarrow$ **float**	$T.type =$ float
4)	$L \rightarrow L_1$, **id**	$L_1.inh = L.inh$ $addType($**id**$.entry, L.inh)$
5)	$L \rightarrow$ **id**	$addType($**id**$.entry, L.inh)$



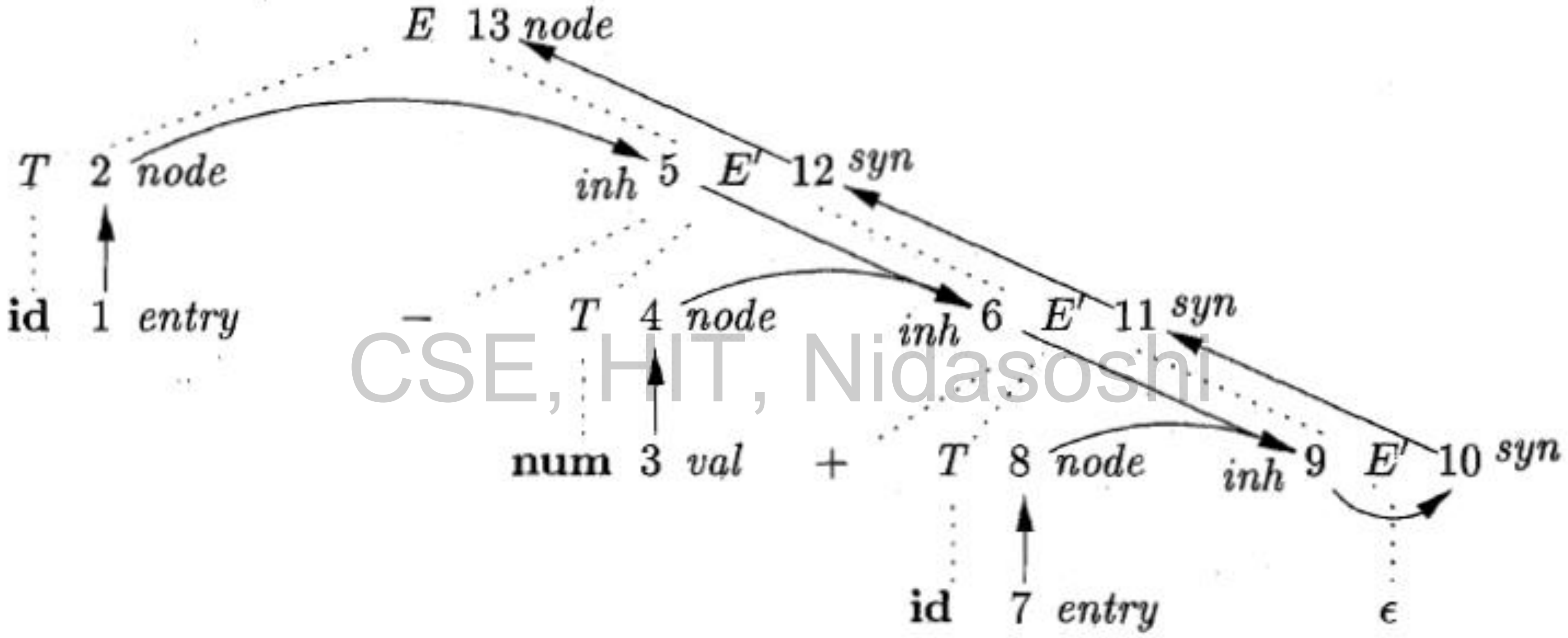
Syntax-directed definition for simple type declarations



	Production	Semantic Rules
1)	$E \rightarrow T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \rightarrow +\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \rightarrow -\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
5)	$T \rightarrow (\ E\ )$	$T.node = E.node$
6)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
7)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Constructing syntax trees during top-down parsing

Syntax-directed definition for simple type declarations



Dependency graph for $a - 4 + c$

Ordering the Evaluation of Attributes

- The dependency graph characterizes the possible orders in which we can evaluate the attributes at the various nodes of a parse tree.
- If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N.
- Thus, the only allowable orders of evaluation are those sequences of nodes N_l, N_2,. . . , N_k such that if there is an edge of the dependency graph from N_i to N_j; then $i < j$.
- Such an ordering embeds a directed graph into a linear order and is called a topological sort of the graph.

Semantic Rules with Controlled Side Effects

- In practice, translations involve side effects: a desk calculator might print a result; a code generator might enter the type of an identifier into a symbol table.
- With SDD's, we strike a balance between attribute grammars and translation schemes.
- Attribute grammars have no side effects and allow any evaluation order consistent with the dependency graph. Translation schemes impose left to right evaluation and allow semantic actions to contain any program fragment.

Semantic Rules with Controlled Side Effects

We shall control side effects in SDD's in one of the following ways:

- Permit incidental side effects that do not constrain attribute evaluation. In other words, permit side effects when attribute evaluation based on any topological sort of the dependency graph produces a “correct” translation, where “correct” depends on the application.
- Constrain the allowable evaluation orders, so that the same translation is produced for any allowable order. The constraints can be thought of as implicit edges added to the dependency graph

Applications of Syntax-Direct ed Translation

1. **Construction of Syntax Trees**

2. **The Structure of a Type**

Applications of Syntax-Directed Translation

1. **Construction of Syntax Trees**

- each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.
- A syntax-tree node representing an expression $E_1 + E_2$ has label **+** and two children representing the subexpressions E_1 and E_2.

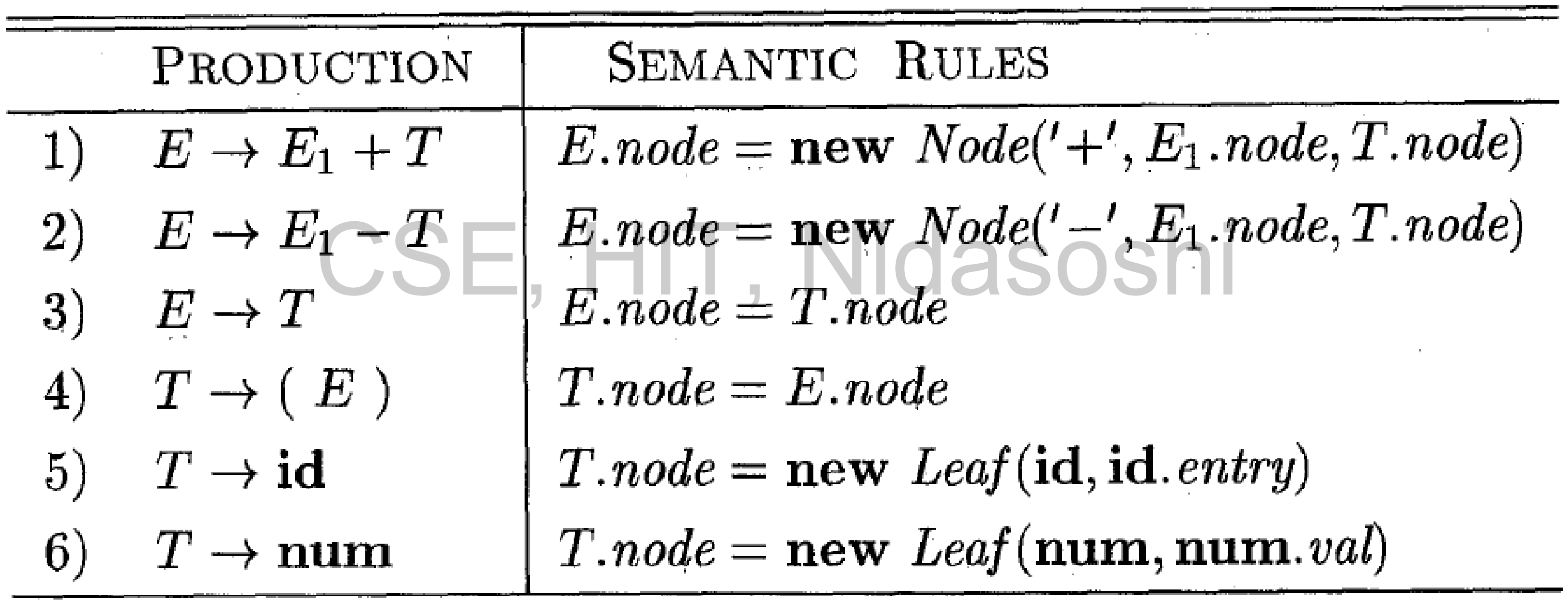
Applications of Syntax-Directed Translation

1. Construction of Syntax Trees - Continued

- We shall implement the nodes of a syntax tree by objects with a suitable number of fields.
- Each object will have an **op** field that is the label of the node.
- The objects will have additional fields as follows:
  - If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function **Leaf (op, val)** creates a leaf object.
  - If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function Node takes two or more arguments: **Node(op, cl, c2, . . . , *ck*)** creates an object with first field **op** and ***k*** additional fields for the ***k*** children **cl, . . . , ck**.

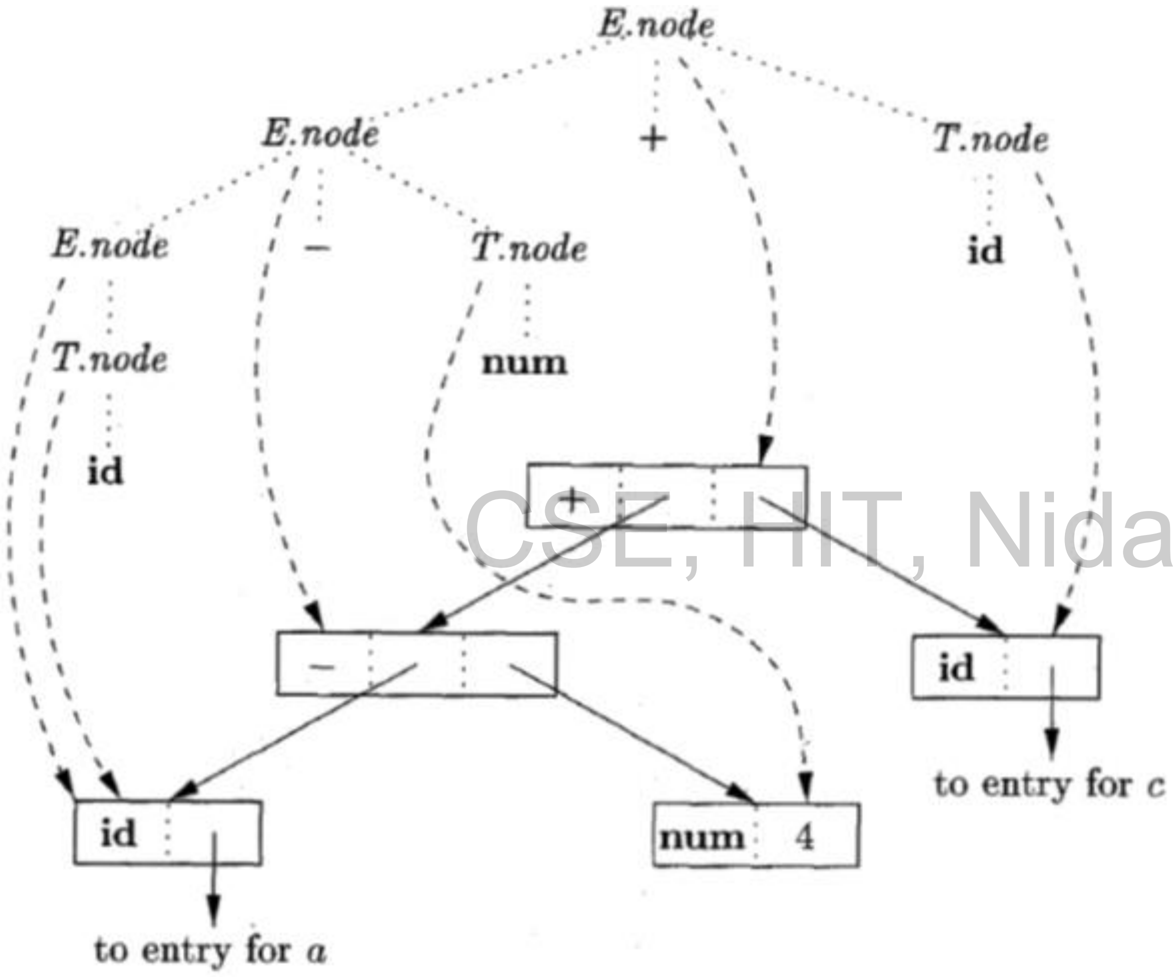
Applications of Syntax-Directed Translation

1. **Construction of Syntax Trees - Continued**



	Production	Semantic Rules
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new } Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new } Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new } Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new } Leaf(\textbf{num}, \textbf{num}.val)$

Constructing syntax trees for simple expressions



Syntax tree for $a - 4 + c$

1) $p_1 = \textbf{new } Leaf(\textbf{id}, entry\text{-}a);$
2) $p_2 = \textbf{new } Leaf(\textbf{num}, 4);$
3) $p_3 = \textbf{new } Node('-', p_1, p_2);$
4) $p_4 = \textbf{new } Leaf(\textbf{id}, entry\text{-}c);$
5) $p_5 = \textbf{new } Node('+', p_3, p_4);$

Applications of Syntax-Direct ed Translation

2. The Structure of a Type

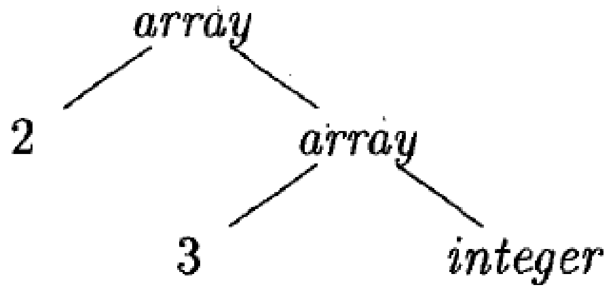
- Inherited attributes are useful when the structure of the parse tree differs from the abstract syntax of the input; attributes can then be used to carry information from one part of the parse tree to another.
- The next example shows how a mismatch in structure can be due to the design of the language, and not due to constraints imposed by the parsing method

Applications of Syntax-Direct ed Translation

2. The Structure of a Type – Example 1

- In C, the type **`int` [2][3]** can be read as, "array of 2 arrays of 3 integers."
- The corresponding type expression **array(2, array(3, integer))** is represented by the tree in Fig.

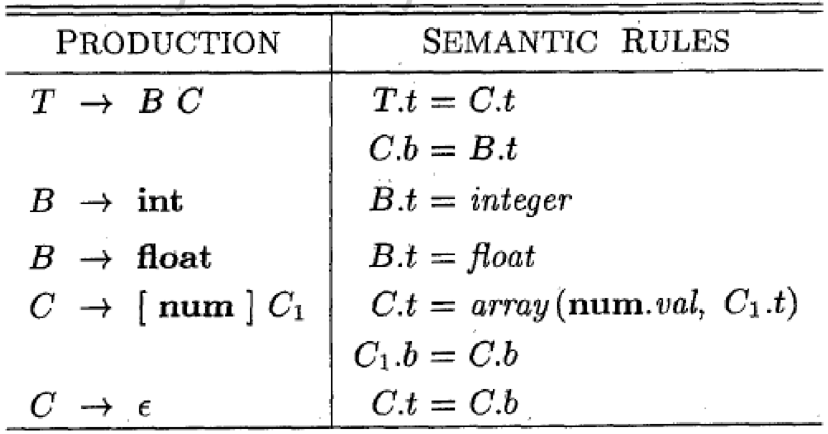
- The operator array takes two parameters, a number and a type.
- If types are represented by trees, then this operator returns a tree node labeled array with two children for a number and a type



Applications of Syntax-Directed Translation

2. The Structure of a Type – Example 2

- The SDD in Fig. nonterminal T generates either a basic type or an array type. Nonterminal B generates one of the basic types **`int`** and **`float`.**
- **`T`** generates a basic type when T derives B *C* and *C* derives ***E*.** Otherwise, *C* generates array components consisting of a sequence of integers, each integer surrounded by brackets



PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

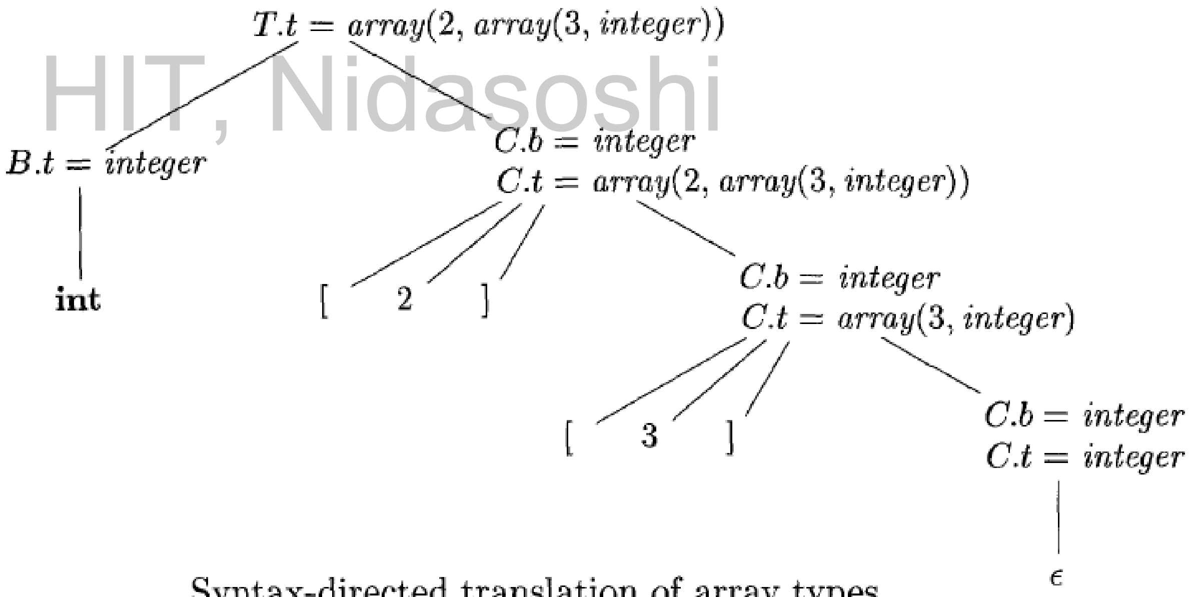
T generates either a basic type or an array type

Applications of Syntax-Directed Translation

2. The Structure of a Type – Example 2

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

T generates either a basic type or an array type



$T.t = array(2, array(3, integer))$
$B.t = integer$
int
$C.b = integer$
$C.t = array(2, array(3, integer))$
[
2
]
$C.b = integer$
$C.t = array(3, integer)$
[
3
]
$C.b = integer$
$C.t = integer$
ϵ

Syntax-directed translation of array types