Semantic Analysis: Syntax Directed Translation

CSE 420

Lecture 10-11

Semantic Analysis

- **Semantic Analysis** computes additional information related to the meaning of the program once the syntactic structure is known.
- In typed languages as C, semantic analysis involves
  - *adding information to the symbol table* and
  - *performing type checking*.
- The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax.
- As for Lexical and Syntax analysis, also for Semantic Analysis we need both a *Representation Formalism* and an *Implementation Mechanism*.

Syntax Directed Definition

- A **syntax-directed definition** specifies the values of attributes by associating semantic rules with the grammar productions.
- An infix-to-postfix translator might have a production and rule:

Production	**Semantic Rule**
$E \rightarrow E_1 + T$	*E.code = E1.codde* || *T.code* || '+'

Syntax Directed Translation

- A **syntax-directed translation** scheme embeds program fragments called semantic actions within production bodies,

$$E \rightarrow E_1 + T \qquad \{\, print\ `+' \,\}$$

- The position of the semantic action in the production body determines the order in which the action is executed.

SDD vs SDT

- Syntax-Directed Definition
  - more readable
  - more useful for specification
- Syntax-Directed Translation
  - more efficient
  - more useful for implementation

Syntax Directed Definition: Intro

- A Syntax-directed definition (SDD) is a context-free grammar together with attributes and rules.
  - grammar symbols have an associated set of Attributes;
  - productions are associated with Semantic Rules for computing the values of attributes.
- Attributes can be of any kind:
  - *numbers, types, table references, strings* etc.
    - variables may have an attribute "type" (which records the declared type of a variable, useful later in type-checking) or,
    - an integer constant may have an attribute "value" (which we will later need to generate code).

Syntax Directed Definition: Intro (Cont.)

- Evaluation of Semantic Rules may:
  - Generate Code;
  - Insert information into the Symbol Table;
  - Perform Semantic Check;
  - Issue error messages; etc.

Syntax Directed Definitions

- Such formalism generates **Annotated Parse-Trees** where each node of the tree is a record with a field for each attribute (e.g., $X.a$ indicates the attribute a of the grammar symbol X).
- The attribute value for a parse node may depend on *information from its children nodes below or its siblings and parent node above.*

Syntax Directed Definitions

- Consider this production, augmented with a set of actions that use the "value" attribute for a digit node to store the appropriate numeric value.
- Below, we use the syntax *X.a to refer to the attribute a associated with symbol X*.

```
digit     -> 0  {digit.value = 0}
           | 1  {digit.value = 1}
           | 2  {digit.value = 2}
           ...
           | 9  {digit.value = 9}
```

Syntax Directed Definitions

- Attributes may be passed up a parse tree to be used by other productions:

```
int₁      -> digit        {int₁.value = digit.value}
           | int₂ digit   {int₁.value = int₂.value*10 + digit.value}
```

- We are using subscripts in this example to clarify which attribute we are referring to,
  - so int_1 and int_2 are different instances of the same non-terminal symbol

Syntax Directed Definitions (Cont.)

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

- We distinguish between two kinds of attributes:

1. **Synthesized Attributes.** They are computed from the values of the attributes of the children nodes.
2. **Inherited Attributes.** They are computed from the values of the attributes of both the siblings and the parent nodes.

Synthesized Attributes

- A synthesized attribute for a non-terminal ***A*** at a parse-tree node N is defined by
  - a semantic rule associated with the production at N. Note that the production must have A as its head.

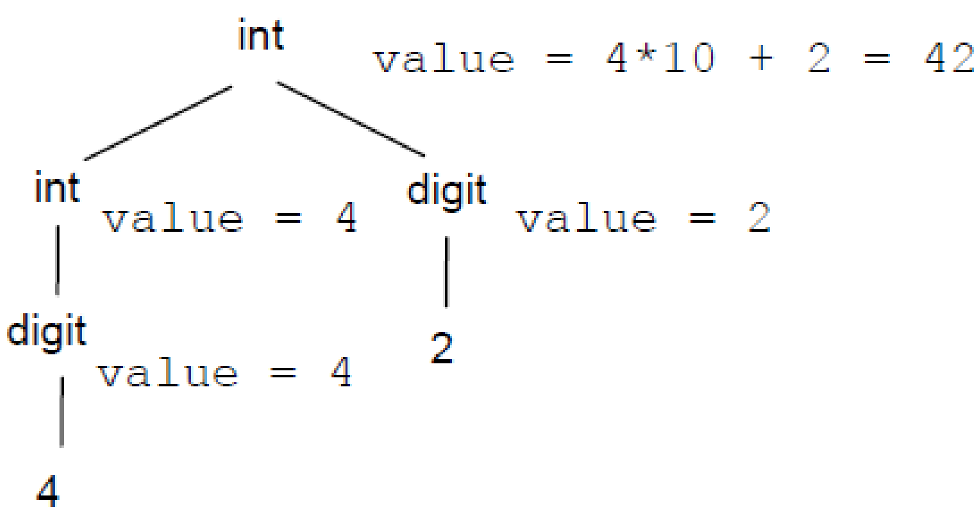
- *A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.*

Synthesized Attributes

- the lexical analyzer usually supplies the attributes of terminals
- the synthesized ones are built up for the non-terminals and passed up the tree.

$$X \rightarrow Y_1 Y_2 \ldots Y_n$$

$$X.a = f(Y_1.a,\ Y_2.a,\ \ldots Y_n.a)$$



Inherited Attributes

- An inherited attribute for a non-terminal B at a parse-tree node N is defined by
  - A semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body.

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

Inherited Attributes

- The right-side attributes are derived from the left-side attributes (or other right-side attributes).
- These attributes are used for passing information about the context to nodes further down the tree.

$$X \rightarrow Y_1 Y_2 \ldots Y_n$$

$$Y_k.a = f(X.a,\ Y_1.a,\ Y_2.a,\ \ldots,\ Y_{k-1}.a,\ Y_{k+1}.a,\ \ldots,\ Y_n.a)$$

Inherited and Synthesized Attributes

- Terminals can have synthesized attributes, which are given to it by the lexer (not the parser).
- There are no rules in an SDD giving values to attributes for terminals.
- Terminals do not have inherited attributes.
- A non-terminal can have both inherited and synthesized attributes.

Evaluating an SDD at the Nodes of a Parse Tree

- Parse tree helps us to visualize the translation specified by SDD.
- The rules of an SDD are applied by first constructing a parse tree
    - then using the rules to evaluate all of the attributes at each of the nodes of the parse tree.
- A parse tree, showing the value(s) of its attribute(s) is called an **annotated parse tree**.

Evaluating an SDD at the Nodes of a Parse Tree

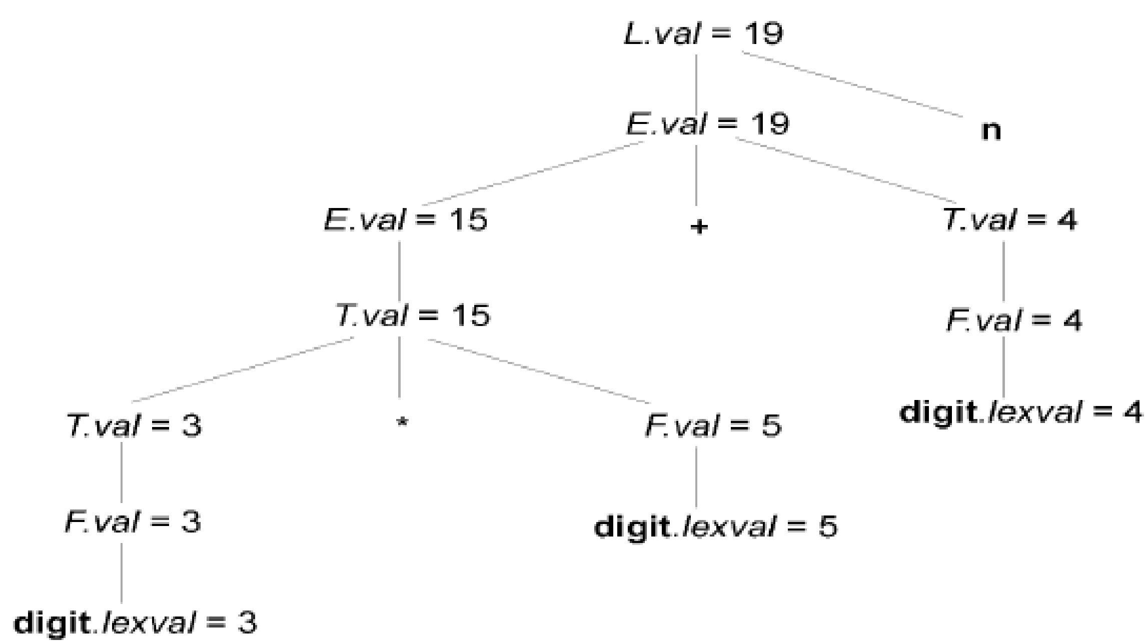
- For SDDs with synthesized attributes,
  - We can evaluate attributes in any bottom-up order, as a post-order traversal of the parse tree

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

Annotated parse tree:
3*5 + 4 n



Evaluating an SDD at the Nodes of a Parse Tree

- For SDDs with both inherited and synthesized attributes,
  - There is no guarantee that there is even one order in which to evaluate attributes at nodes.

e.g.

Production	Semantic Rules
$A \rightarrow B$	$A.s = B.i$
	$B.i = A.s + 1$

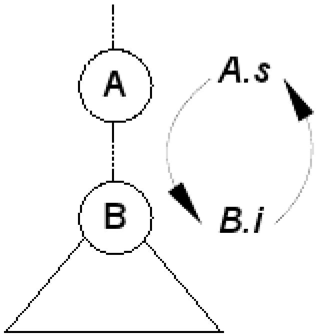


Fig 5.2: The circular dependency of *A.s* and *B.i* on one another

Evaluating an SDD at the Nodes of a Parse Tree

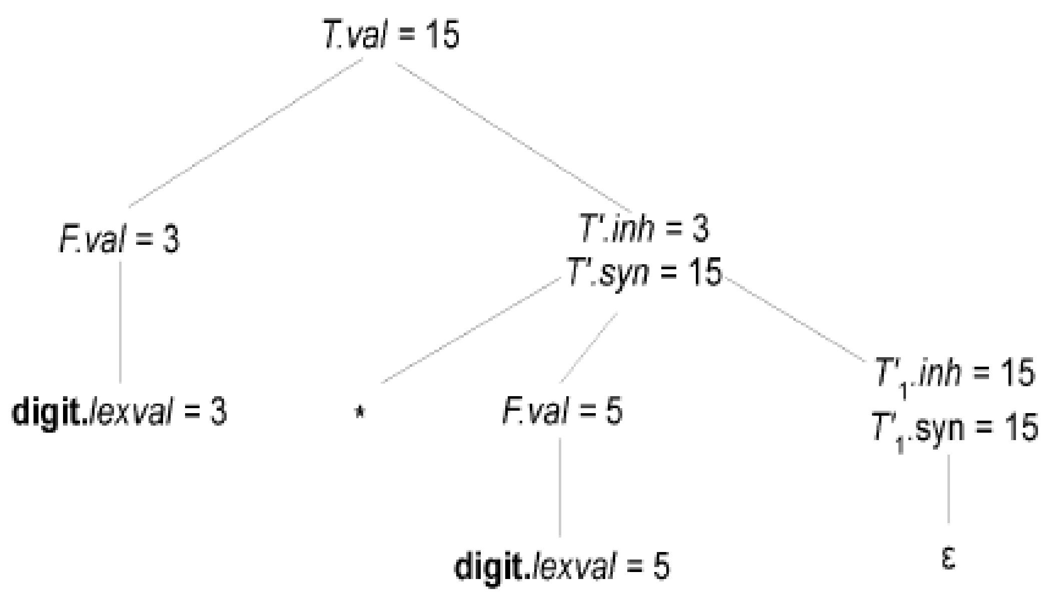
- It is computationally difficult to determine whether or not exist any circularities in any of the parse trees that a given SDD could have to translate.
- **Inherited attributes** are useful when the structure of a parse tree does not match the abstract syntax of the source code.
- They can be used to overcome the mismatch due to grammar designed for parsing rather than translation.

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



S-Attributed Definitions

- An SDD is *S-attributed if every attribute is synthesized*.
- *Attributes of an S-attributed* SDD can be evaluated in bottom-up order of the nodes of parse tree.
- Evaluation is simple using post-order traversal.

```
postorder(N) {
    for (each child C of N, from the left)
        postorder(C);
        evaluate attributes associated with node N;
}
```

- S-attributed definitions can be implemented during bottom-up parsing as
  - bottom-up parse corresponds to a postorder traversal
  - post-order corresponds to the order in which an LR parser reduces a production body to its head

L-Attributed Definitions

- Each attribute must be either
  - Synthesized, or
  - Inherited, but with the rules limited as follows.

Suppose that there is a production $A \rightarrow X_1 X_2 \bullet\bullet\bullet X_n$, there is an inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only:

- Inherited attributes associated with the head *A.*
- Either inherited or synthesized attributes associated with the occurrences of symbols $X_1 X_2 \bullet\bullet\bullet X_{i-1}$ *located to the left of* X_i
- Inherited or synthesized attributes associated with this occurrence of X_i itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i.

L-Attributed Definitions-Example

Production	Semantic Rules
$T \rightarrow F\,T'$	$T'.inh = F.val$
$T' \rightarrow *F T_1'$	$T_1'.inh = T'.inh \times F.val$

Production	Semantic Rules
$A \rightarrow B\ C$	$A.s = B.b$
	$B.i = f(C.c, A.s)$

Evaluating an SDD at the Nodes of a Parse Tree

Consider the following grammar that defines declarations and simple expressions in a Pascal-like syntax:

```
P  –>  DS
D  –>  var V; D | ε
S  –>  V := E; S | ε
V  –>  x | y | z
```

Evaluating an SDD at the Nodes of a Parse Tree

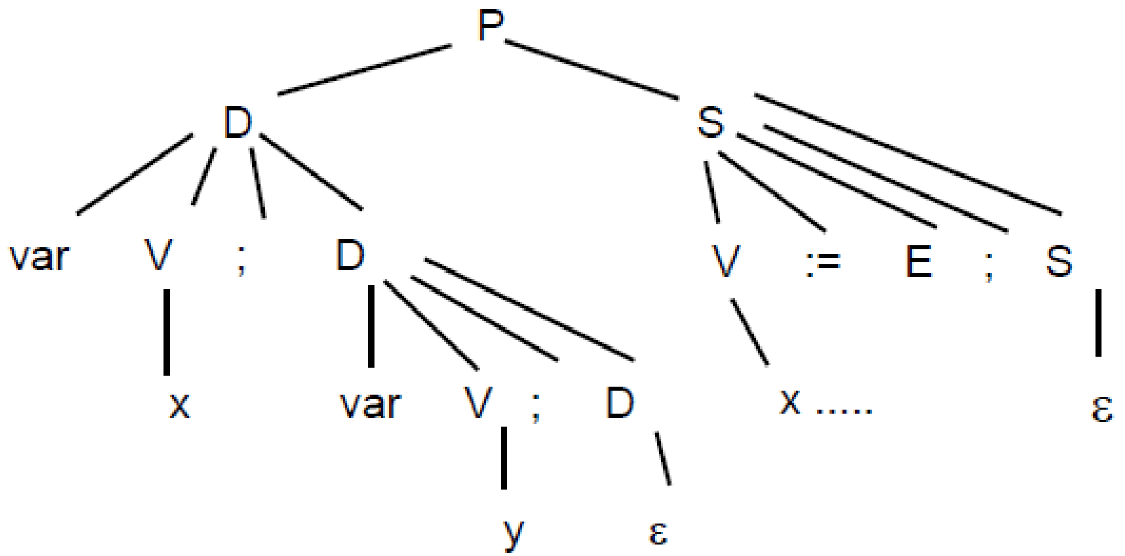
- Now we add two attributes to this grammar, ***name*** and ***dl***, for the name of a variable and the list of declarations.
- Each time a new variable is declared, a synthesized attribute for its ***name*** is attached to it.
- That name is added to a list of variables declared so far in the synthesized attribute ***dl*** that is created from the declaration block.
- The list of variables is then passed as an inherited attribute to the statements following the declarations for use in checking that variables are declared before use.

P	–>	DS	`{ S.dl = D.dl }`
D_1	–>	var V; D_2	`{` D_1`.dl = addlist(V.name,` D_2`.dl) }`
	|	ε	`{` D_1`.dl = NULL }`
S_1	–>	V := E; S_2	`{ check(V.name,` S_1`.dl);` S_2`.dl =` S_1`.dl }`
	|	ε	
V	–>	x	`{ V.name = 'x' }`
	|	y	`{ V.name = 'y' }`
	|	z	`{ V.name = 'z' }`

Evaluating an SDD at the Nodes of a Parse Tree

```
var x;
var y;

x := ...;
y := ...;
```



Evaluating an SDD at the Nodes of a Parse Tree

```
typedef struct _attribute {
   char *name;
   struct _attribute *list;
} attribute;

   P -> DS              { $2.list = $1.list }
   D -> var V; D        { $$.list = add_to_list($2.name, $4.list) }
      | ε               { $$.list = NULL }
   S -> V := E; S       { check($1.name, $$.list); $5.list = $$.list }
      | ε
   V -> x               { $$.name = 'x' }
      | y               { $$.name = 'y' }
      | z               { $$.name = 'z' }
```

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.

- It depicts the flow of information among the attribute on stances in a particular parse tree.

Evaluation Orders for SDD's

- Each attribute is associated to a node.

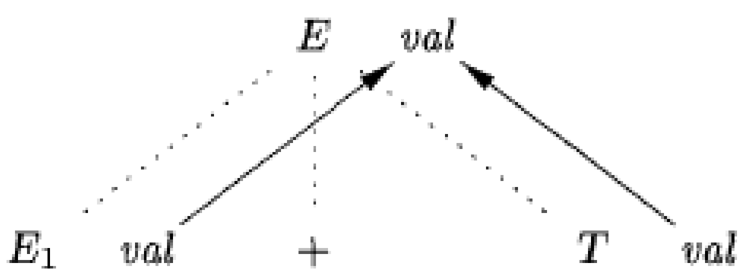
- If a semantic rule associated with a production p defines the value of **synthesized attribute** $A.b$ in terms of the value of $X.c$, then graph has an edge from $X.c$ *to* $A.b$

- If a semantic rule associated with a production p defines the value of **inherited attribute** $B.c$ in terms of value of $X.a$, then graph has an edge from $X.a$ *to* $B.c$

Evaluation Orders for SDD's

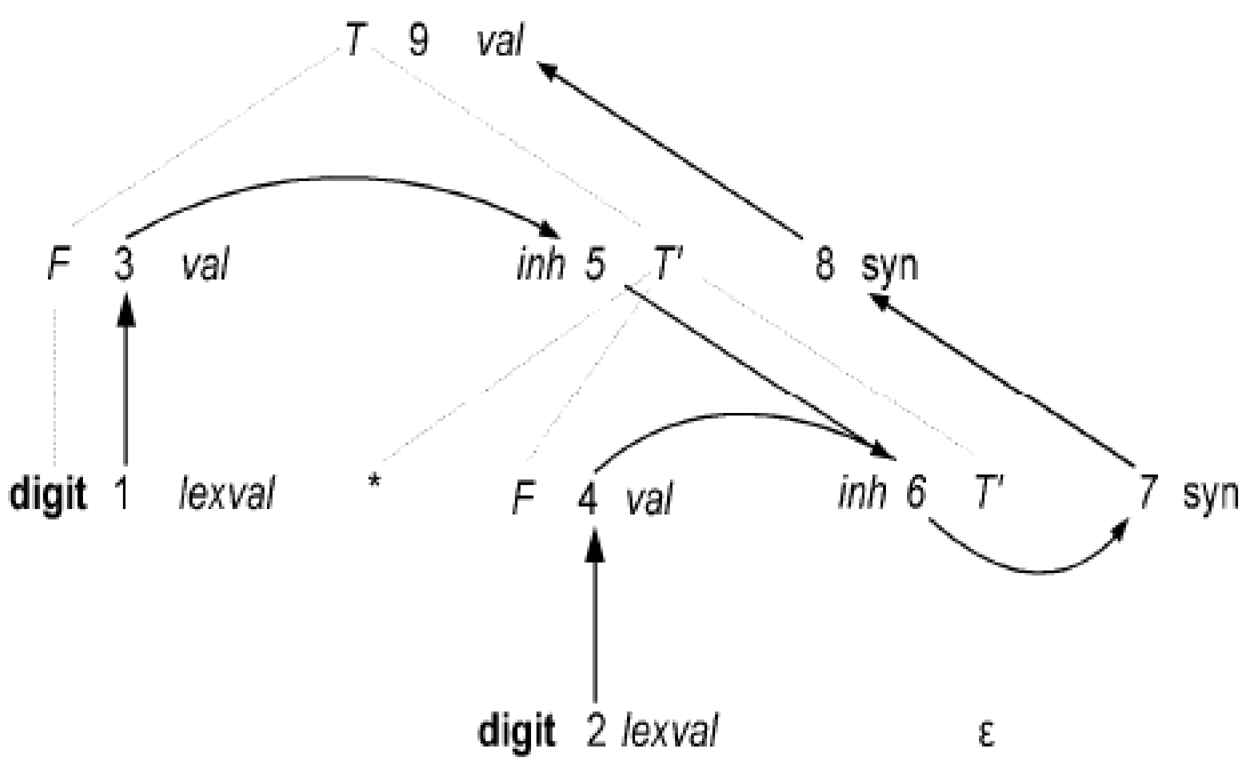
PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

- At every node N labeled E with children correspond to the body of production,
  - The synthesized attribute *val* at N is computed using the values of *val* at the two children, labeled E and T



Evaluation Orders for SDD's

Dependency graph for the annotated parse tree for 3*5



SDD For Simple Type Declarations

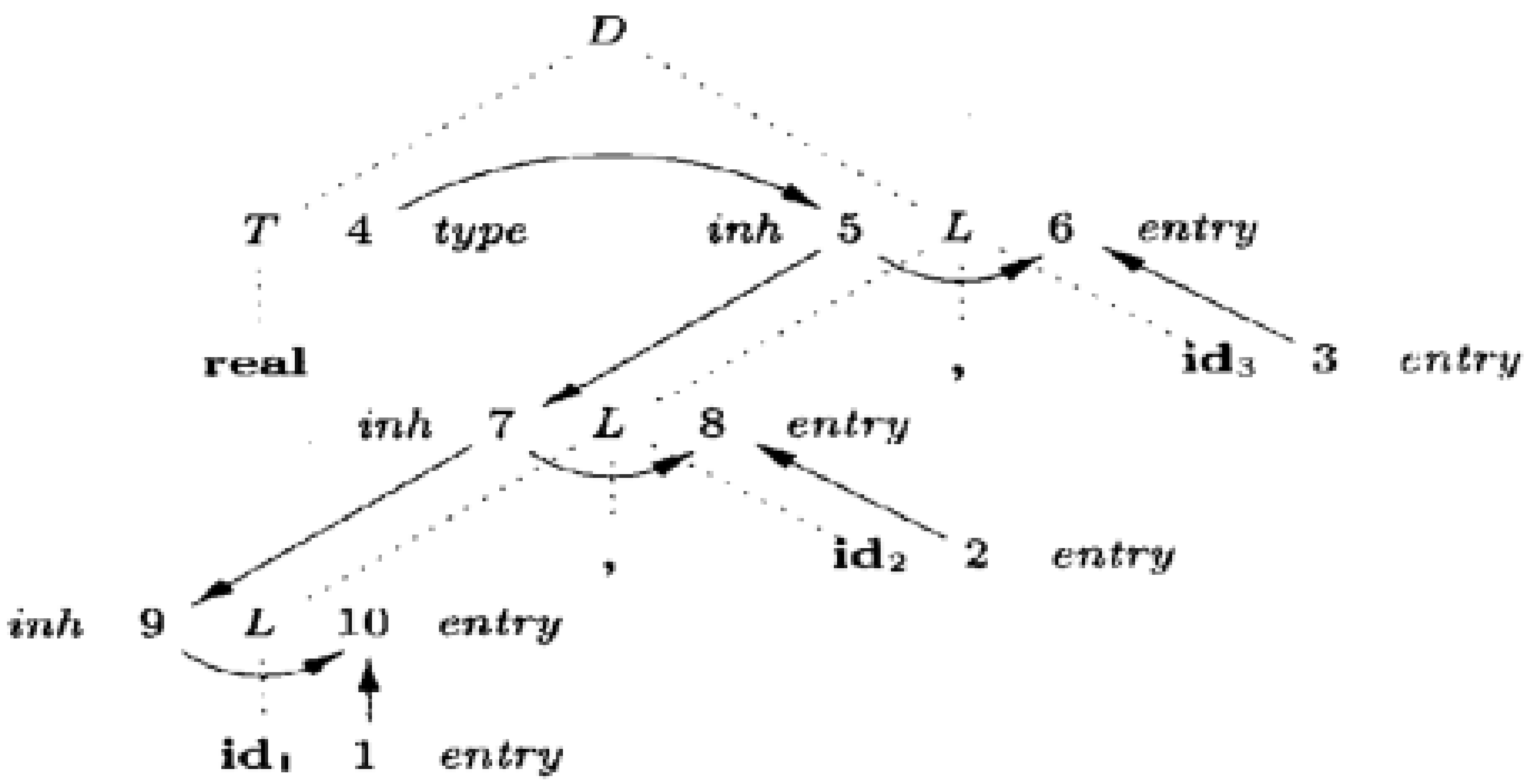
Production	Semantic Rules
1) $D \rightarrow T\ L$	$L.inh = T.type$
2) $T \rightarrow$ **int**	$T.type$ = integer
3) $T \rightarrow$ **float**	$T.type$ = float
4) $L \rightarrow L_1$, **id**	$L_1.inh = L.inh$ *addType* (**id**.*entry*, *L.inh*)
5) $L \rightarrow$ **id**	*addType* (**id**.*entry*, *L.inh*)

• The purpose of **L.inh** is to pass the declared type down the list of identifiers, so that it can be the appropriate symbol-table entries.
• Productions 2 and 3 each evaluate the synthesized attribute **T.type**, giving it the appropriate value, integer or float.
• Productions 4 and 5 also have a rule in which a function **addType** is called with two arguments:
1. **id.entry**, a lexical value that points to a symbol-table object, and
2. **L.inh**, the type being assigned to every identifier on the list.
• The function addType properly installs the type L.inh as the type of the represented identifier.

Dependency Graph For Simple Type Declarations

A dependency graph for the input string

float id1 , id 2, id3



Any Question?