SEMANTIC ANALYSIS

Semantic Analysis

- Parser builds abstract syntax tree
- Now, compiler needs to extract semantic information and check constraints
- It can sometimes be done during the parse, but often easier to organize as a separate pass
- It is the last phase in front end

Semantic Analysis

- Extract types and other information from the program
- Check language rules that go beyond the grammar
- Assign storage locations
- Key data structure: Symbol Tables – For each identifier in the program, record its attributes (kind, type, etc.)

Semantic Analysis

- Lexical analysis: detecting illegal tokens (e.g., string constant too long, illegal characters in the string)
- Syntax analysis: structure errors, no parse tree for the string according to given grammar (brackets don’t match)

Semantic Analysis

- Semantic analysis (dependent on the language): e.g., specifying wrong types (int a = b+c (c cannot be string)), buffer overflows, divide by zero
  - *Scoping: e.g., all variables must be declared*
  - *Typing: certain operations only can be performed on the certain types of variables; e.g., actual parameter types have to match formal parameter types*

Typical Semantic Errors

- The following are some of the semantics errors that the semantic analyzer is expected to encounter:
  - *Type mismatch*
  - *Undeclared variable*
  - *Reserved identifier misuse*
  - *Multiple declaration of variable in a scope*
  - *Accessing an out of scope variable*
  - *Actual and formal parameter mismatch*

Typical Semantic Errors

- The type of the right-side expression of an assignment statement should match the type of the left-side, and the left-side needs to be a properly declared and assignable identifier (i.e. not some sort of constant)
- The parameters of a function should match the arguments of a function call in both number and type

Typical Semantic Errors

- The language may require that identifiers are unique, disallowing a global variable and function of the same name
- The operands to multiplication operation will need to be of numeric type, perhaps even the exact same type depending on the strictness of the language

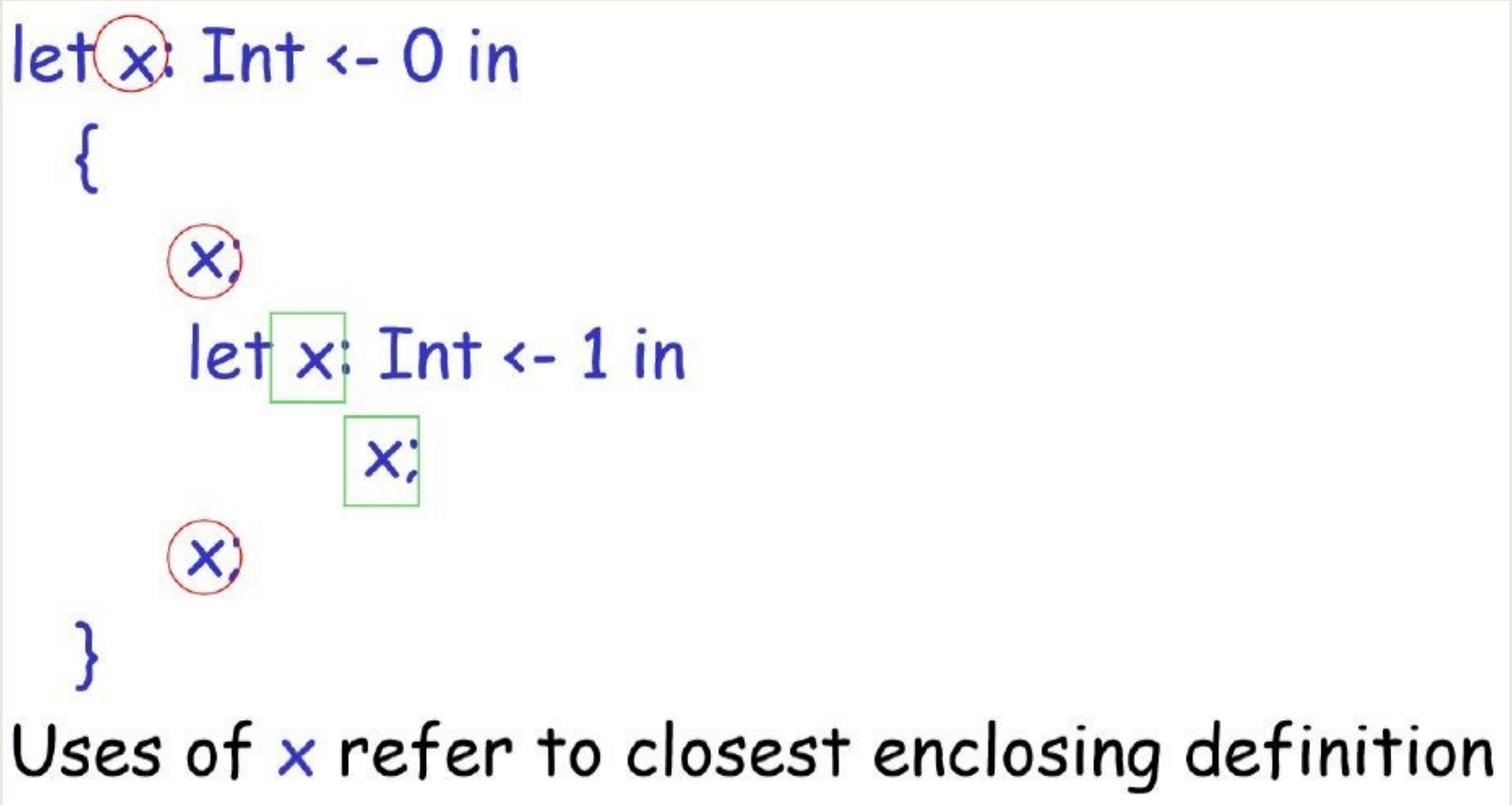
Scoping

- Match identifiers' declaration with uses
- Visibility of an entity
- Binding between declaration and uses
- The **scope** of an identifier is the portion of a program in which that identifier is accessible
- The same identifier may refer to different things in different parts of the program: Different scopes for same name don’t overlap
- An identifier may have restricted scope

Static vs Dynamic Scope

- Most languages have static scope: Scope depends only on the program text, not runtime behavior (e.g., Java, C)
- A few languages are dynamically scoped: Scope depends on execution of the program (e.g., Lisp, SNOBOL, Perl allow dynamic scope through certain keywords) – context specific scoping
- Dynamic scoping means that when a symbol is referenced, the compiler/interpreter will walk up the symbol-table stack to find the correct instance of the variable to use

Static Scope



Static scoping

```
1  const int b = 5;
2  int foo()
3  {
4     int a = b + 5;
5     return a;
6  }
7
8  int bar()
9  {
10    int b = 2;
11    return foo();
12 }
13
14 int main()
15 {
16    foo(); // returns 10
17    bar(); // returns 10
18    return 0;
19 }
```

Dynamic scoping

```
1  const int b = 5;
2  int foo()
3  {
4     int a = b + 5;
5     return a;
6  }
7
8  int bar()
9  {
10    int b = 2;
11    return foo();
12 }
13
14 int main()
15 {
16    foo(); // returns 10
17    bar(); // returns 7
18    return 0;
19 }
```

Syntax-Directed Translation

- Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent
- Values of these attributes are evaluated by the **semantic rules** associated with the grammar productions
- The way of associating attributes to each of the symbols in the grammar that provides us the order of translation, allowing some level of independence is called **syntax directed translation**

Syntax-Directed Translation

- Evaluation of these semantic rules:
  - *may generate intermediate codes*
  - *may put information into the symbol table*
  - *may perform type checking*
  - *may issue error messages*
- An attribute may hold a string, a number, a memory location, a complex record

Syntax-Directed Translation

- When we associate semantic rules with productions, we use two notations:
  - *Syntax-Directed Definitions*
  - *Translation Schemes*
- **Syntax-Directed Definitions** are high level specifications for translations. They hide many implementation details and free the user from having to explicitly specify the order in which translation takes place.

Syntax-Directed Translation

- **Translation Schemes** indicate the order in which semantic rules are to be evaluated (using a dependency graph), so they allow some implementation details to be shown.

Syntax-Directed Definitions

- Syntax-directed definitions (or attribute grammar) bind a set of semantic rules to productions
- They are the high level specification for the translation schemes i.e. they hide the implementation details and do not necessitate the consideration of translations order
- It is the generalization of parse tree where each terminal and nonterminal has attributes associated to it whose values are determined by the semantic rules

Syntax-Directed Definitions

- A syntax-directed definition is a generalization of a context-free grammar in which:
  - *Each grammar symbol is associated with a set of attributes.*
  - *This set of attributes for a grammar symbol is partitioned into two subsets called* ***synthesized*** *and* ***inherited*** *attributes.*
  - *Each production rule is associated with a set of semantic rules*

Syntax-Directed Definitions

- Semantic rules set up dependencies between attributes which can be represented by a dependency graph
- This dependency graph determines the evaluation order of these semantic rules
- If the value of the attribute only depends upon its children then it is **synthesized attribute**
- The syntax directed definition with only synthesized attribute is called **S-attributed definition** or **S- attributed grammar**

Syntax-Directed Definitions

- If the value of the attribute depends upon its parent or siblings then it is **inherited attribute**
- 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**) of the parse tree

- In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

$$b = f(c_1, c_2, \ldots, c_n)$$

where f is a function, $c_1, c_2, \ldots, c_n$ are attributes of the grammar symbols in the production $A \rightarrow \alpha$ and either

  - *b is a synthesized attribute of A*
  - *b is an inherited attribute of one of the grammar symbols in α*

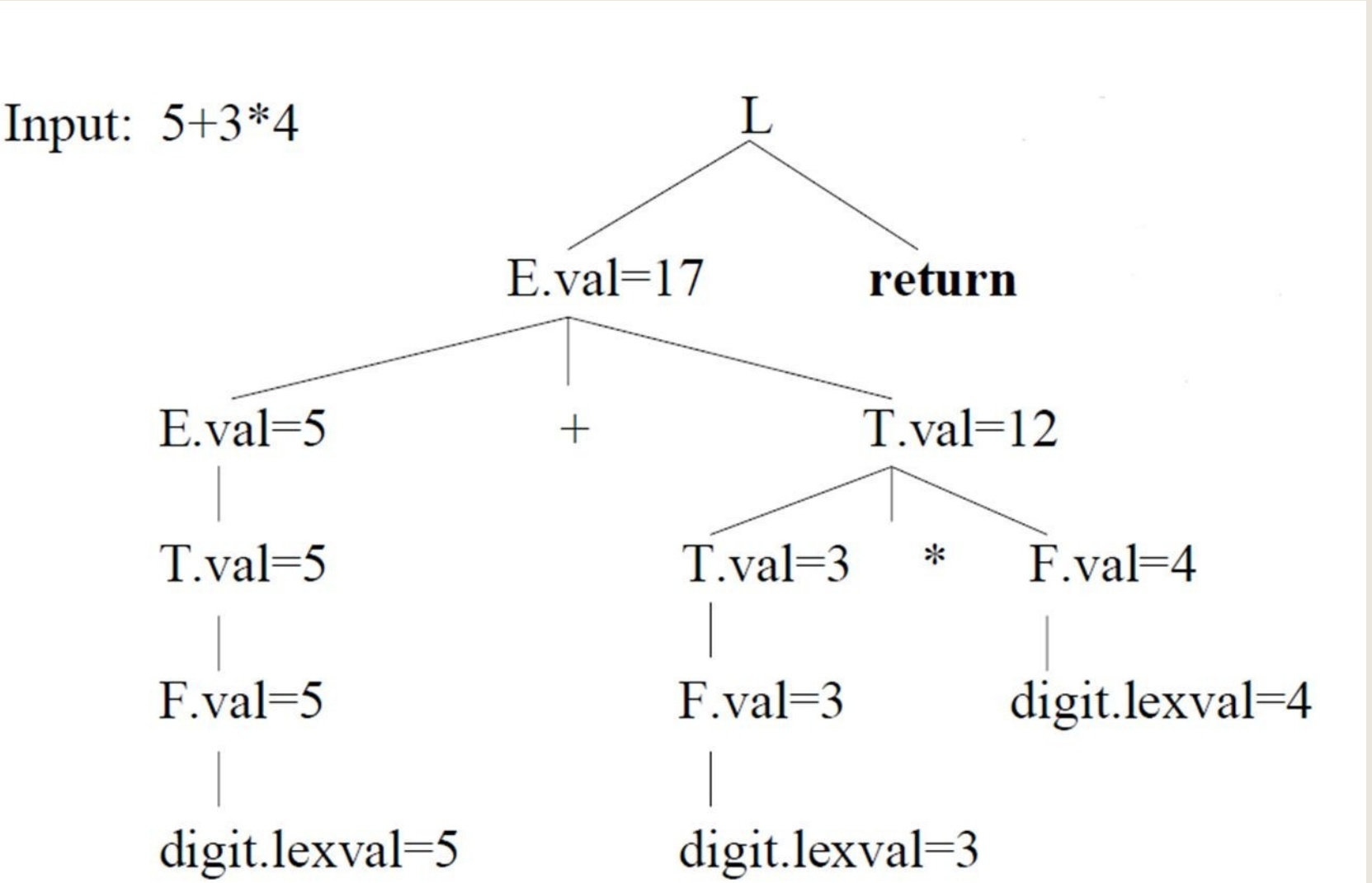
- So, a semantic rule $b = f(c_1, c_2, \ldots, c_n)$ indicates that the attribute b depends on attributes $c_1, c_2, \ldots, c$

- Example :

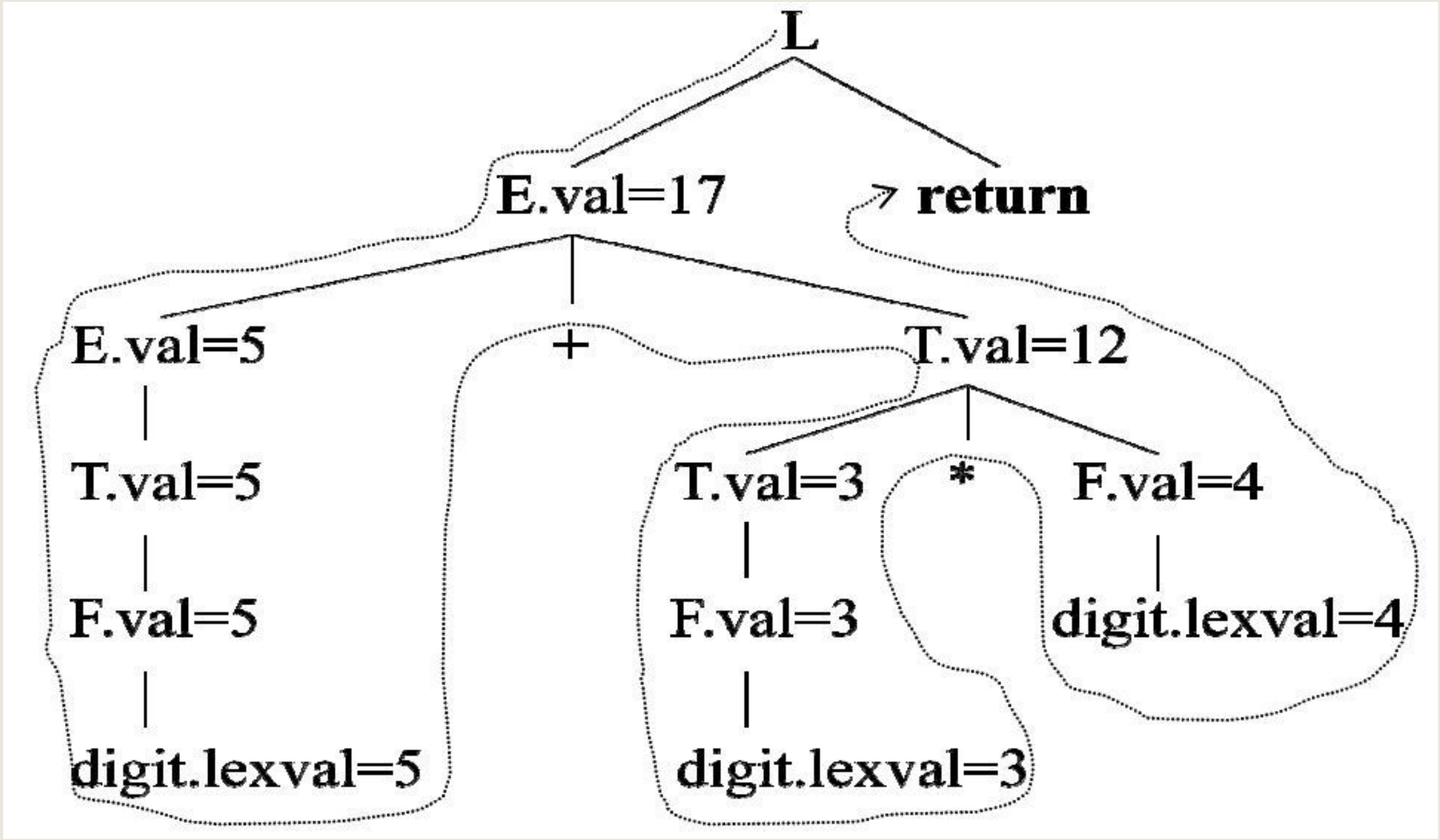
Production	Semantic Rules
$L \rightarrow E$ **return**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	F.val = **digit**.lexval

All attributes in this example are of the synthesized type. Symbols E, T, and F are associated with a synthesized attribute **val**. The token digit has a synthesized attribute **lexval** (it is assumed that it is evaluated by the lexical analyzer).

Annotated Parse Tree



Computation starts from leaf node with associated production and corresponding semantic rule. Output: The value, printed at the root of tree, is the value of **E.val** at the first child of the root.

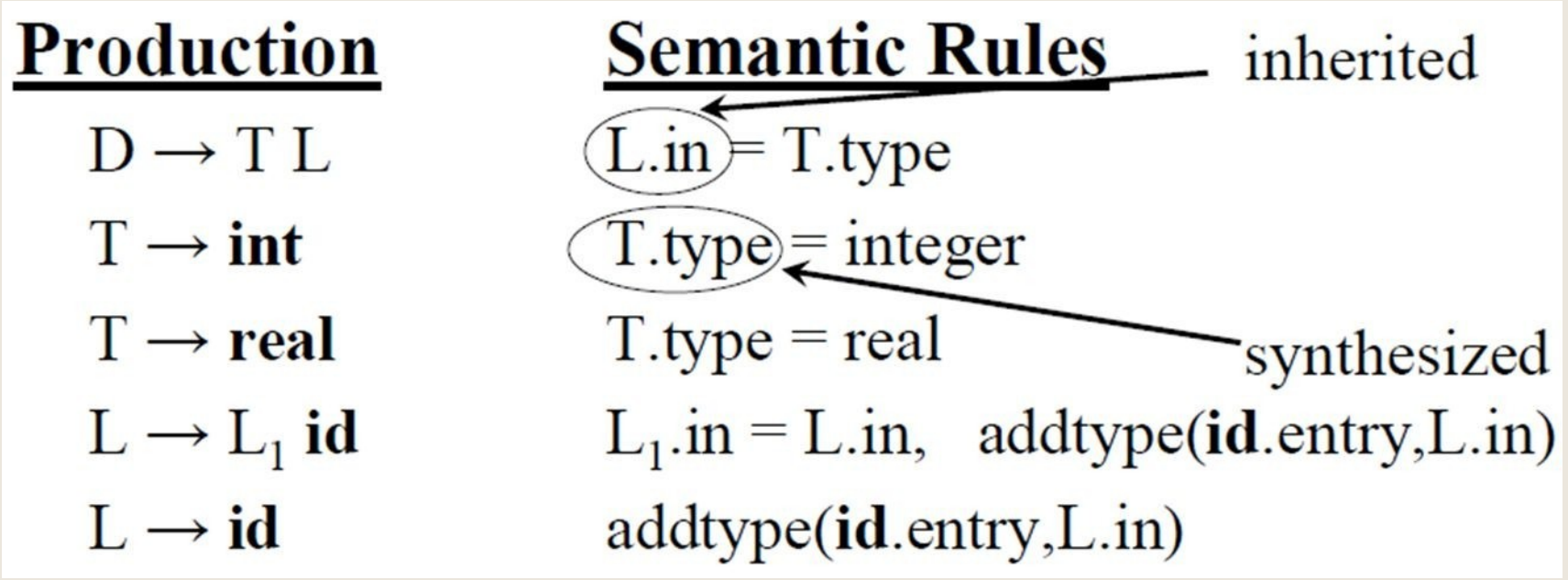

L
E.val=17
return
E.val=5
+
T.val=12
T.val=5
T.val=3
*
F.val=4
F.val=5
F.val=3
digit.lexval=4
digit.lexval=5
digit.lexval=3

Inherited Attributes

- An inherited attribute at any node is defined based on the attributes at the parent and/or siblings of the node
- They are useful for describing context sensitive behavior of grammar symbols
- For example, an inherited attribute can be used to keep track of whether an identifier appears at the left or right side of an assignment operator
- This can be used to decide whether the address or the value of the identifier is needed

Inherited Attributes

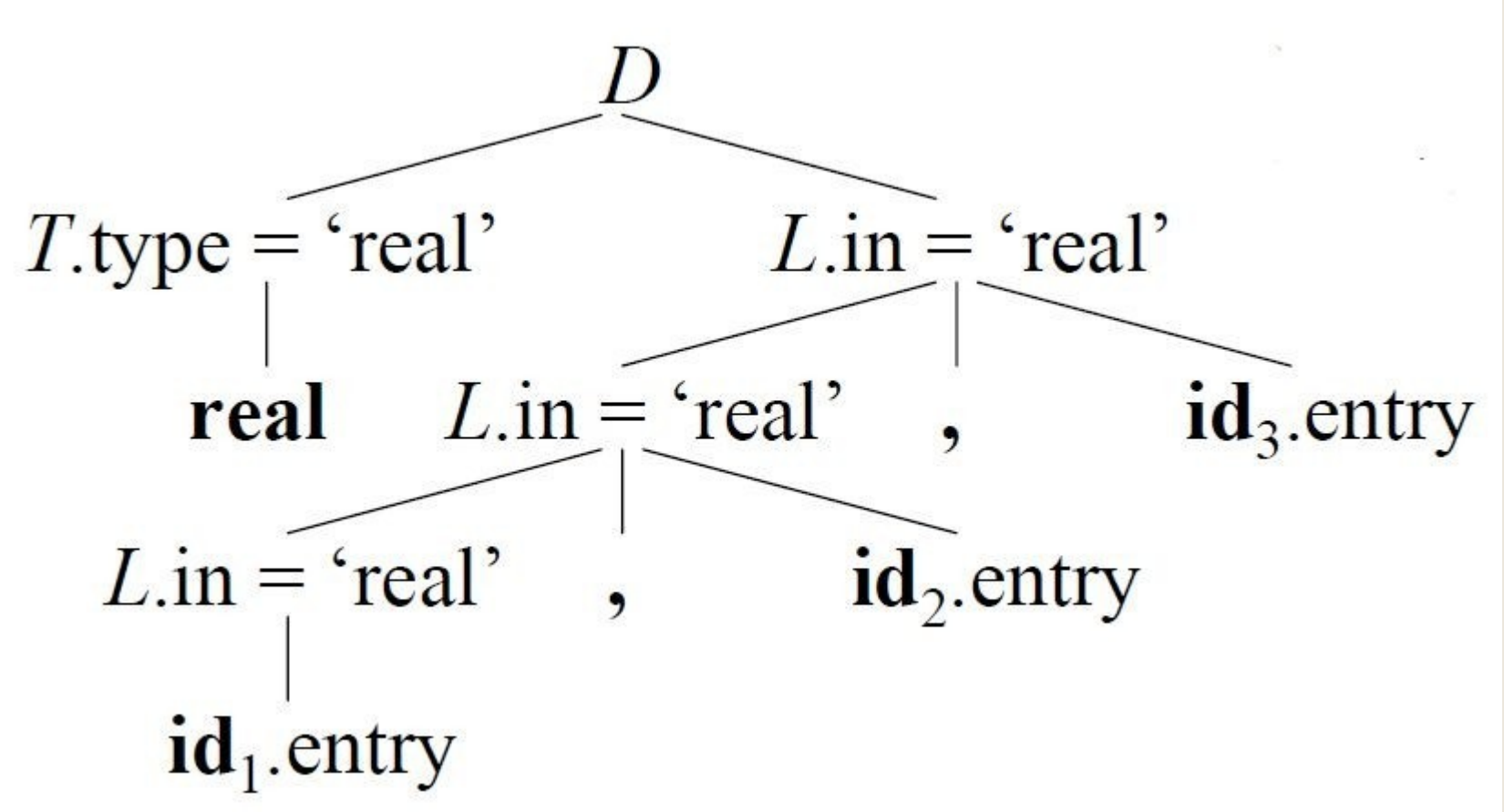
- Example



Symbol T is associated with a synthesized attribute ***type***

Symbol L is associated with an inherited attribute ***in***

Example (Annotated Parse Tree): Input : real id1, id2, id3



The value of **L.in** at the three L-nodes gives the type of identifiers **id1**, **id2** & **id3**. These values are determined by computing the value of attribute **T.type** at the left child of the root and then evaluating **L.in** in top-down at the three L-nodes in the right subtree of the root

Dependency Graph

- In order to correctly evaluate attributes of syntax tree nodes, a dependency graph is useful
- A dependency graph is a directed graph that contains attributes as nodes and dependencies across attributes as edges

Algorithm for dependency graph construction

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

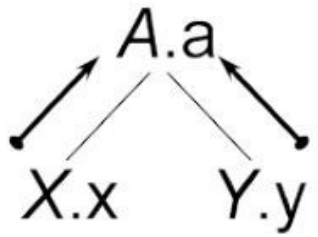
for each semantic rule $\mathbf{b = f(c_1, c_2, \ldots, c_k)}$ associated with the production used at **n** do

for **i** = 1 to **k** do

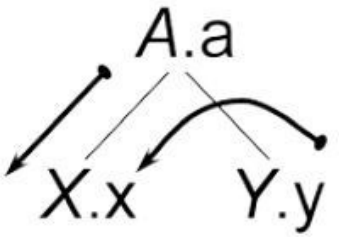
Construct an edge from **c** to the node for **b**

Acyclic Dependency Graphs for Parse Trees

$A \rightarrow X\ Y$



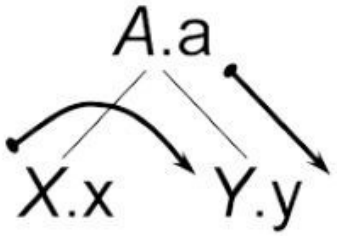
$A.a := f(X.x,\ Y.y)$



$X.x := f(A.a,\ Y.y)$

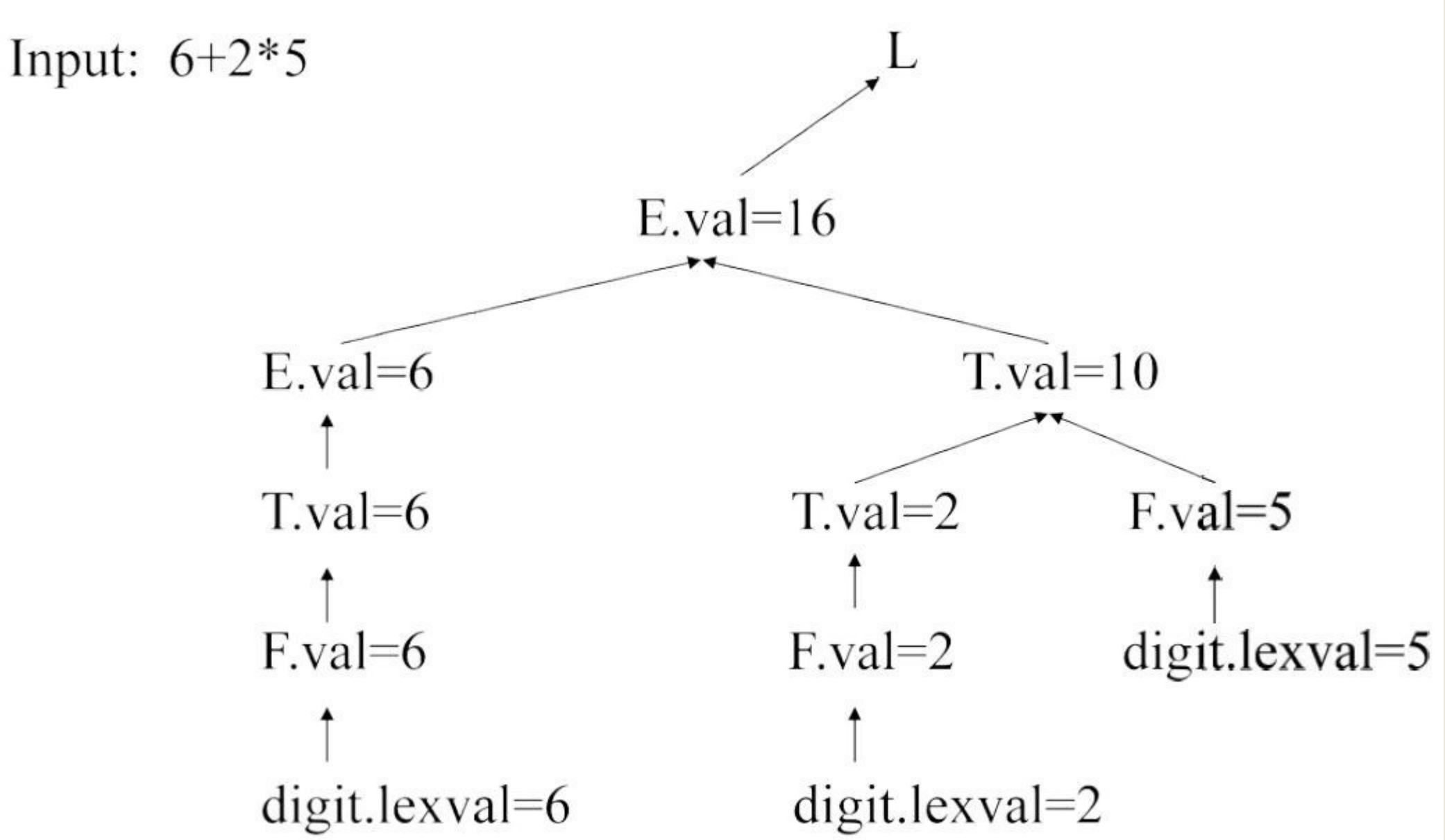
Direction of

value dependence

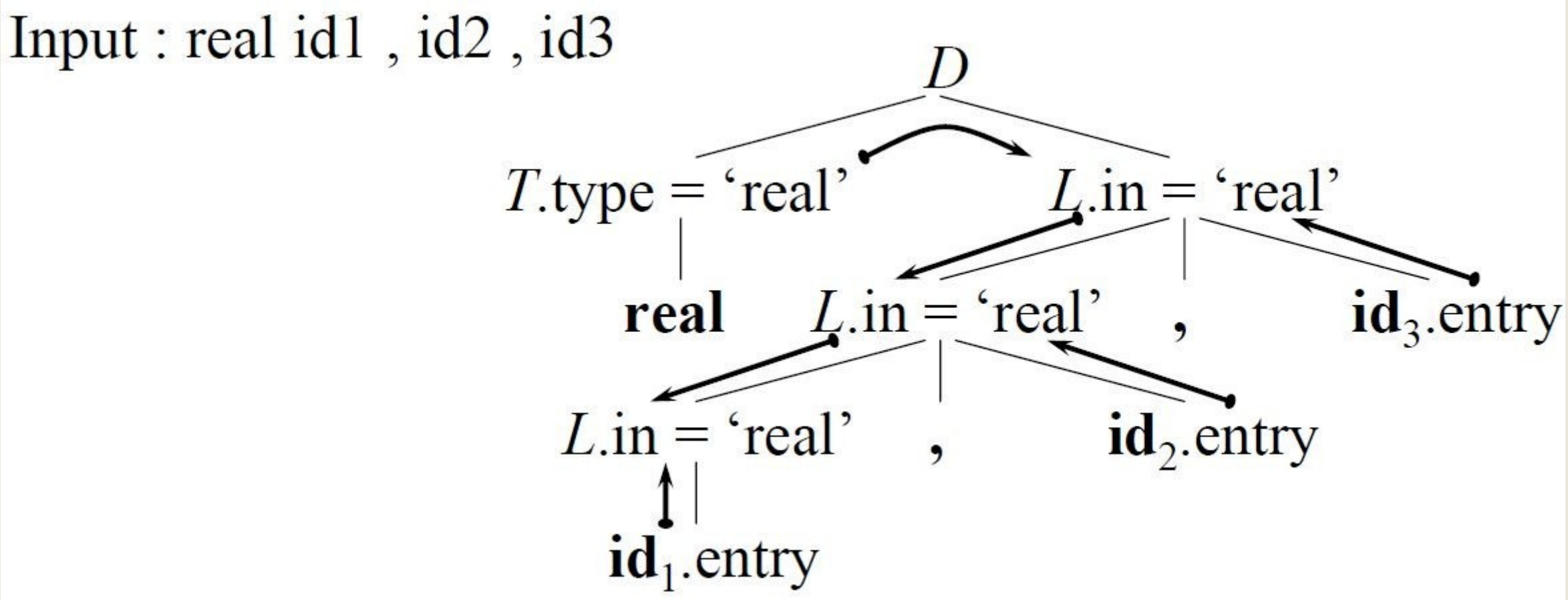


$Y.y := f(A.a,\ X.x)$

Example (Dependency Graph for Annotated Parse Tree for 6 + 2 * 5)



Example (Annotated Parse Tree with Dependency Graph for real id1, id2, id3)



S–Attributed Definitions

- A syntax-directed definition that uses synthesized attributes exclusively is called an S–attributed definition (or S–attributed grammar)
- A parse tree of an S–attributed definition is annotated by evaluating the semantic rules for the attribute at each node bottom-up

Bottom-up Evaluation of S–Attributed Definitions

- An S–attributed grammar can be translated bottom-up when the grammar is being parsed using an LR parser
- The bottom up parser uses a stack to hold information about subtrees that have been parsed
- Let us suppose that the stack is implemented by a pair of arrays **state** and **val**
- If the i^{th} state symbol is A, then **val[i]** will hold the value of the attribute associated with the parse tree node corresponding to A

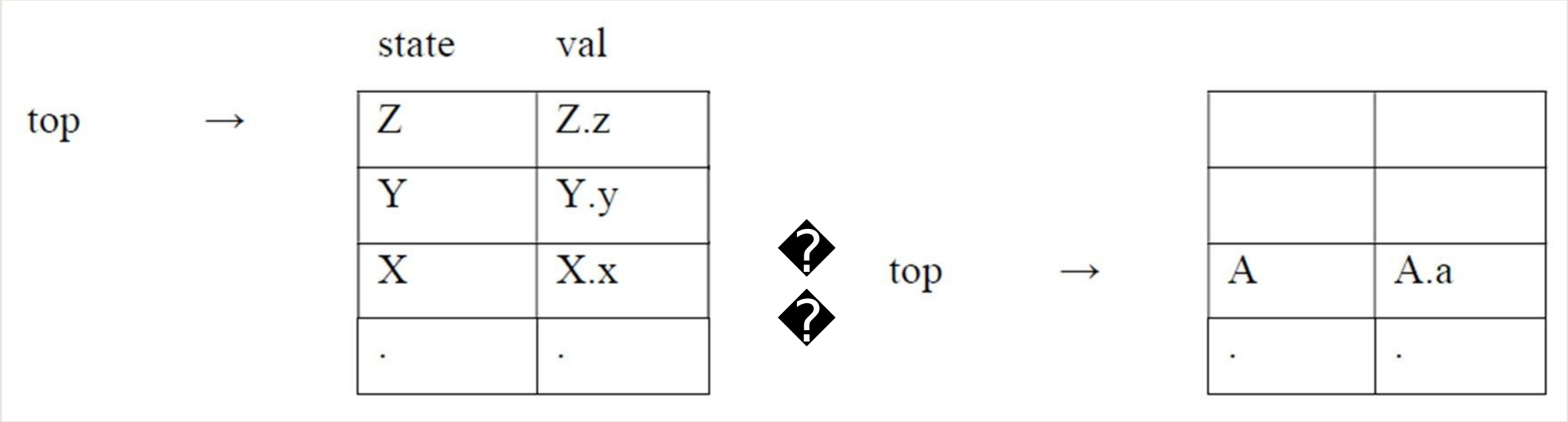
Bottom-up Evaluation of S–Attributed Definitions

- The current top of the stack is indicated by the pointer **top**
- We assume that synthesized attributes are evaluated just before each reduction
- For example, A → XYZ, A.a = f(X.x, Y.y ,Z.z) where all attributes are synthesized
- Before XYZ is reduced to A, the value of the attribute Z.z is in val[top], that of Y.y in val[top–1], and that of X.x in val[top–2]

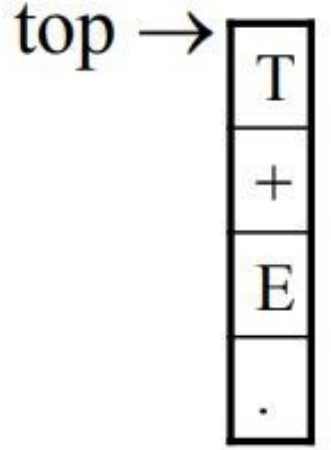
Bottom-up Evaluation of S–Attributed Definitions

- If a symbol has no attribute, then the corresponding entry in the **val** array is undefined
- After the reduction, top is decremented by 2, the state covering A is put in state[top] i.e. where X was, and the value of the synthesized attribute A.a is put in val[top]

Bottom-up Evaluation of S–Attributed Definitions



Production	Semantic Rules
L → E	print(val[top-1])
E → E1 + T	val[top] = val[top-2] + val[top]
E → T	
T → T1 * F	val[top] = val[top-2] * val[top]
T → F	
F → (E)	val[top] = val[top-1]
F → **digit**	



- At each shift of **digit**, we also push **digit.lexval** into *val-stack*.
- At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).

Stack	val-stack	Input	Action	Semantic Rule
$		6+2*5n$	shift	d.lexval(6) into val-stack
$6	6	+2*5n$	F→d	F.val=d.lexval – do nothing
$F	6	+2*5n$	T→F	T.val=F.val – do nothing
$T	6	+2*5n$	E→T	E.val=T.val – do nothing
$E	6	+2*5n$	shift	push empty slot into val-stack
$E+	6 _	2*5n$	shift	d.lexval(2) into val-stack
$E+2	6 _ 2	*5n$	F→d	F.val=d.lexval – do nothing
$E+F	6 _ 2	*5n$	T→F	T.val=F.val – do nothing
$E+T	6 _ 2	*5n$	shift	push empty slot into val-stack
$E+T*	6 _ 2 _	5n$	shift	d.lexval(5) into val-stack
$E+T*5	6 _ 2 _ 5	n$	F→d	F.val=d.lexval – do nothing
$E+T*F	6 _ 2 _ 5	n$	T→T*F	T.val=T1.val*F.val
$E+T	6 _ 10	n$	E→E+T	E.val=E1.val*T.val
$E	16	n$	shift	push empty slot into val-stack
$En	16 _	$	L→En	print(16), pop empty slot from val-stack
$L	16	$	accept	

L–Attributed Definition

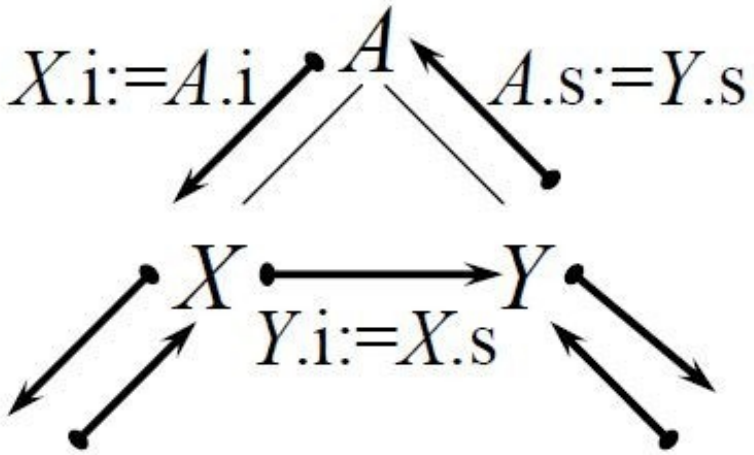
- A syntax-directed definition is L–attributed if each inherited attribute of X_j on the right side of $A \rightarrow X_1 X_2 \ldots X_n$ depends only on
  - *The attributes of the symbols $X_1, X_2, \ldots, X_{j-1}$*
  - *The inherited attributes of A*
- L–attributed definitions allow for a natural order of evaluating attributes: depth-first and left to right
- Every S-attributed syntax-directed definition is also L–attributed and the above rule applies only for inherited attributes

Depth First Evaluation of L–Attributed Definitions

```
Procedure dfvisit(n: node);      //depth-first
    evaluation  for each child m of n, from left to
    right do
    evaluate inherited attributes of m;
        dfvisit(m);
evaluate synthesized attributes of n
```

Depth First Evaluation of L–Attributed Definitions

$A \rightarrow X\,Y$



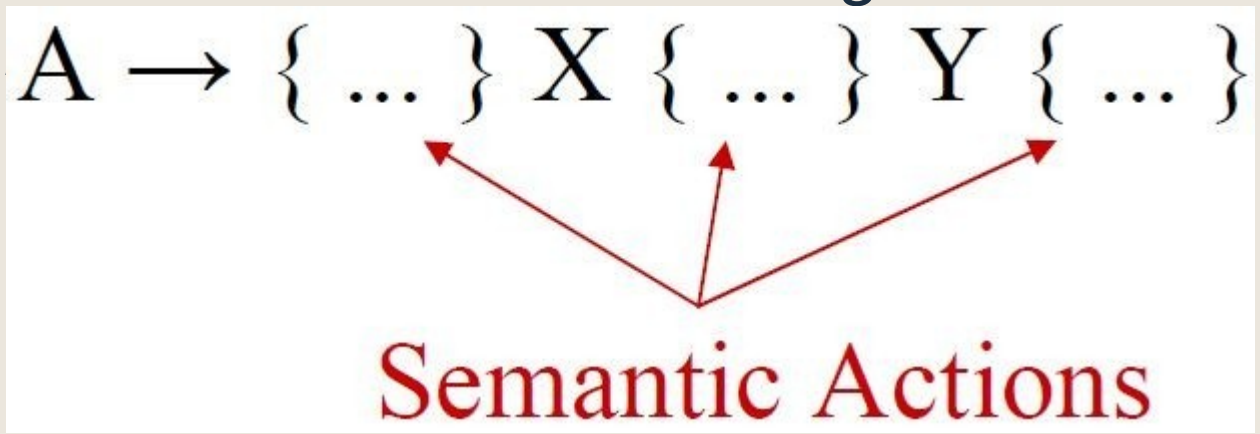
$X.\mathrm{i} := A.\mathrm{i}$

$Y.\mathrm{i} := X.\mathrm{s}$

$A.\mathrm{s} := Y.\mathrm{s}$

Translation Schemes

- A translation scheme is a context-free grammar in which:
  - *attributes are associated with the grammar symbols and*
  - *semantic actions enclosed between braces { } are inserted within the right sides of p*



Translation Schemes

- In translation schemes, we use the term **semantic action** instead of **semantic rule** used in
syntax-directed definitions
- The position of the semantic action on the right side indicates when that semantic action will be evaluated

- Example: A simple translation scheme that converts infix expression to the corresponding postfix expression

$E \rightarrow T\ R$

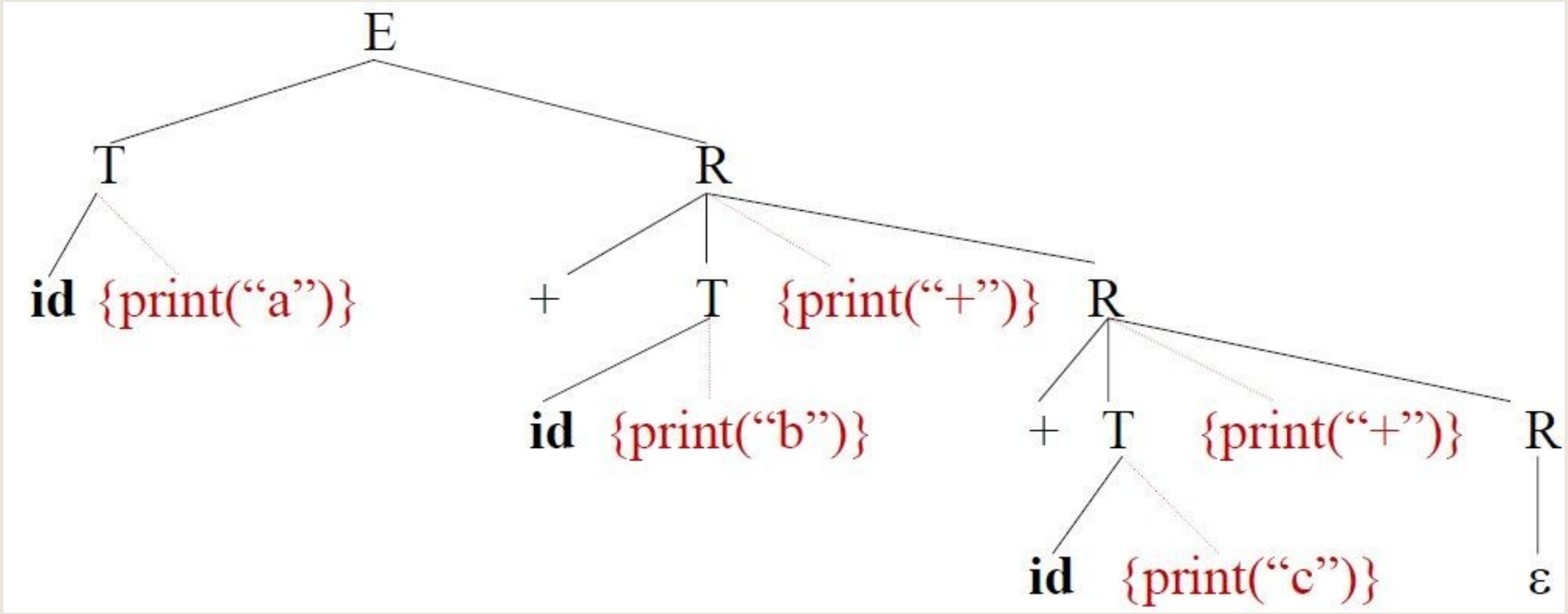
$R \rightarrow +\ T\ \{\ print(\text{“+”})\ \}\ R_1$

$R \rightarrow \varepsilon$

$T \rightarrow \mathbf{id}\ \{\ print(\mathbf{id}.name)\ \}$

a+b+c ➔ ab+c+

infix expression postfix expression



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

Translation Scheme Requirements

- If a translation scheme has to contain both synthesized and inherited attributes, we have to observe the following rules:

1. An inherited attribute of a symbol on the right side of a production must be computed in a semantic action before that symbol.
2. A semantic action must not refer to a synthesized attribute of a symbol to the right of that semantic action.
3. A synthesized attribute for the non-terminal on the left can only be computed after all attributes it references have been computed (we normally put this semantic action at

Translation Scheme Requirements

- With an L–attributed syntax-directed definition, it is always possible to construct a corresponding translation scheme which satisfies these three conditions
- This may not be possible for a general syntax-directed translation

Production	Semantic Rule
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := 'integer'
$T \rightarrow$ **real**	T.type := 'real'
$L \rightarrow L_1$, **id**	L_1.in := L.in; *addtype*(**id**.entry, L.in)
$L \rightarrow$ **id**	*addtype*(**id**.entry, L.in)

Translation Scheme

$D \rightarrow T$ { L.in := T.type } L

$T \rightarrow$ **int** { T.type := 'integer' }

$T \rightarrow$ **real** { T.type := 'real' }

$L \rightarrow$ { L_1.in := L.in } L_1 , **id** { *addtype*(**id**.entry, L.in) }

$L \rightarrow$ **id** { *addtype*(**id**.entry, L.in) }

Top-Down Translation

- L–attributed definitions can be evaluated in a top-down fashion (predictive parsing) with translation schemes
- Since no grammar with left recursion can be parsed deterministically top-down, we used left-recursion elimination algorithm
- The algorithm for elimination of left recursion is extended to evaluate action and attribute

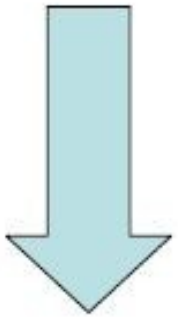
Eliminating Left Recursion from a Translation Scheme

- We can implement translation of L–attributed definition for top down parser by using the method that is similar to the removal of left recursion from the grammar with the following rule:

$A \rightarrow A_1\ Y\ \{ A.a = g(A_1.a, Y.y) \}$

$A \rightarrow X\ \{ A.a = f(X.x) \}$

a left recursive grammar with synthesized attributes (a,y,x).



eliminate left recursion

inherited attribute of the new non-terminal

synthesized attribute of the new non-terminal

$A \rightarrow X\ \{ R.in = f(X.x) \}\ R\ \{ A.a = R.syn \}$

$R \rightarrow Y\ \{ R_1.in = g(R.in, Y.y) \}\ R_1\ \{ R.syn = R_1.syn \}$

$R \rightarrow \varepsilon\ \{ R.syn = R.in \}$

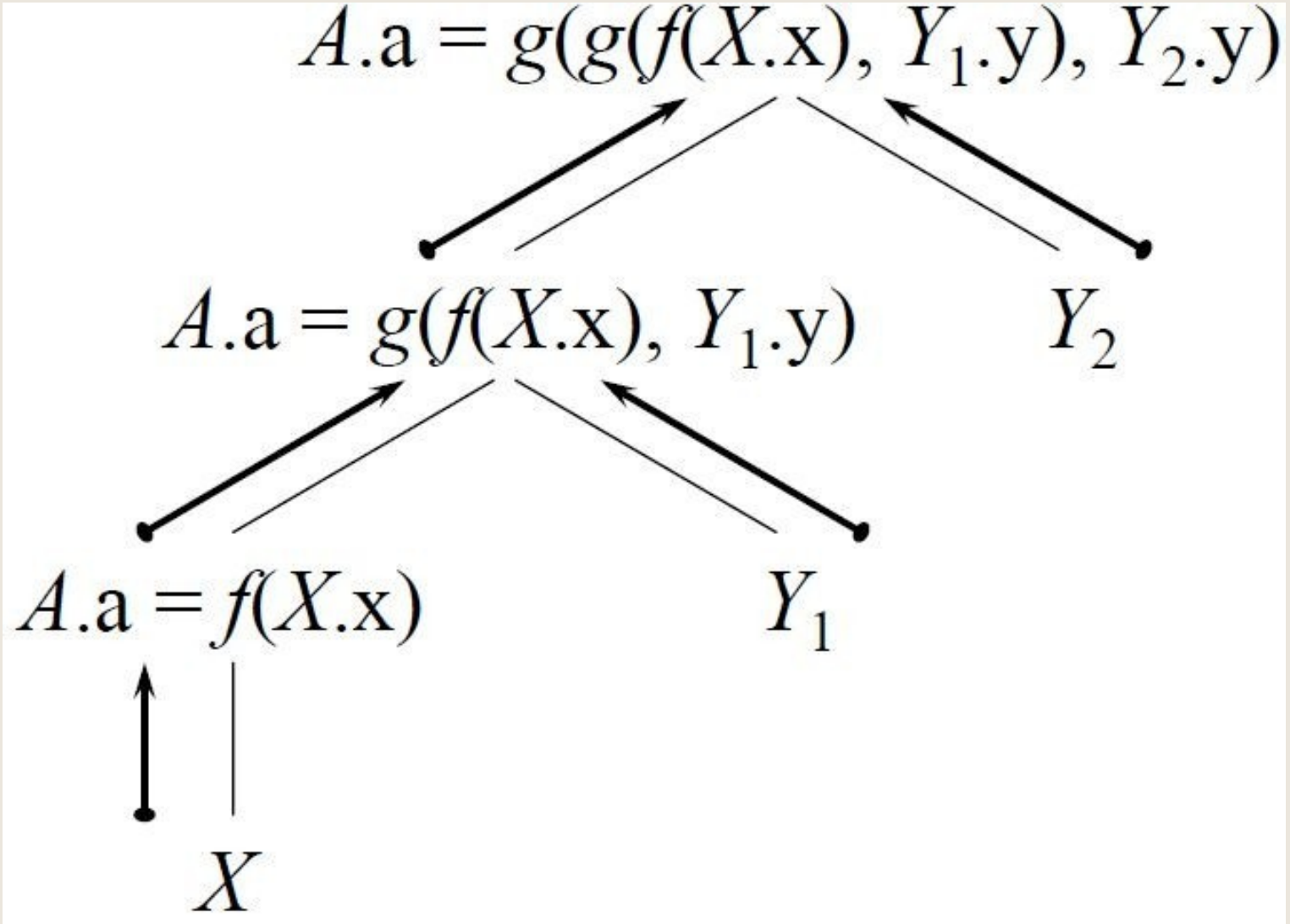
When we eliminate the left recursion from the grammar (to get a suitable grammar for the top-down parsing) we also have to change semantic actions

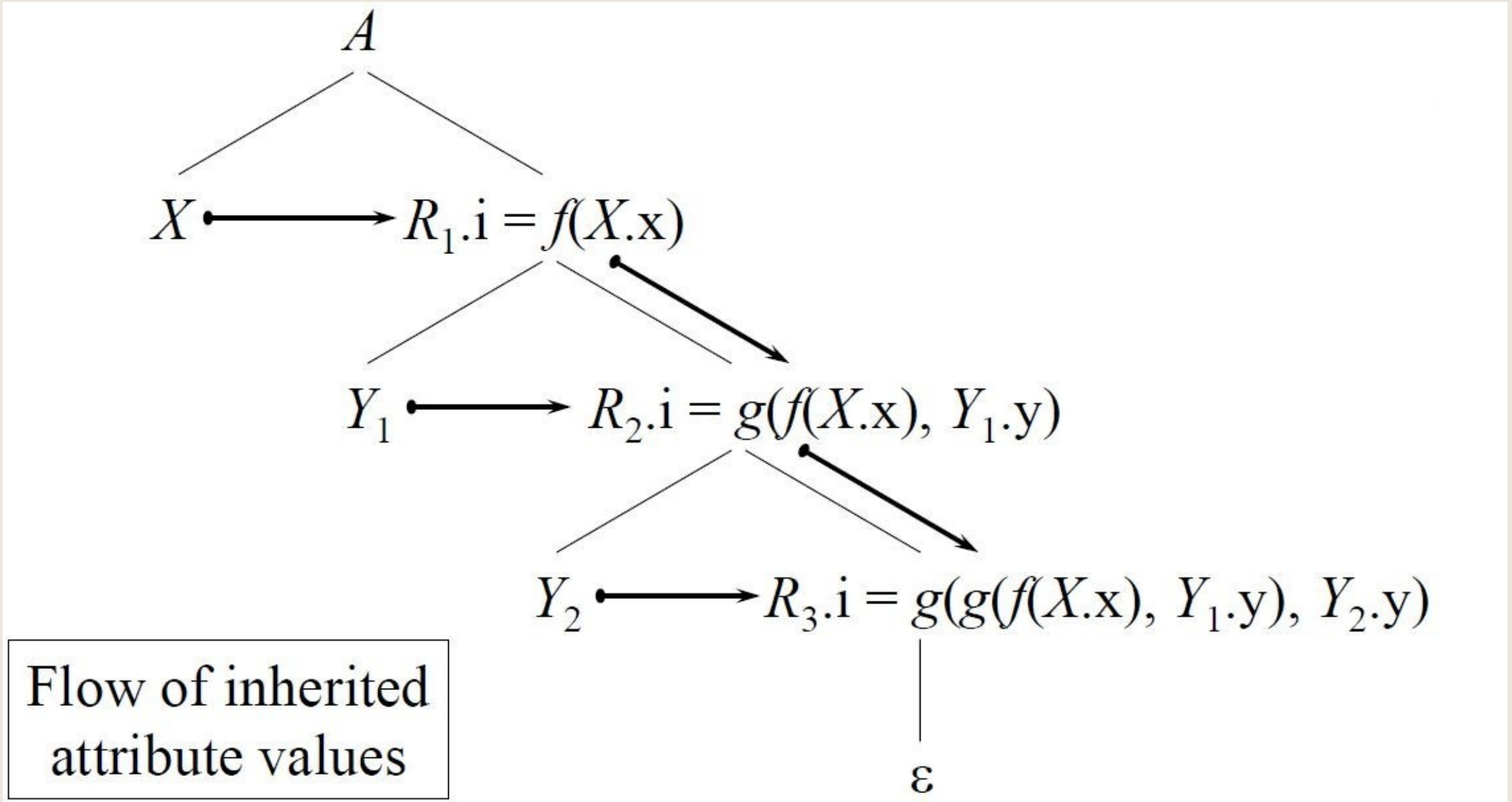
Eliminating Left Recursion from a Translation Scheme

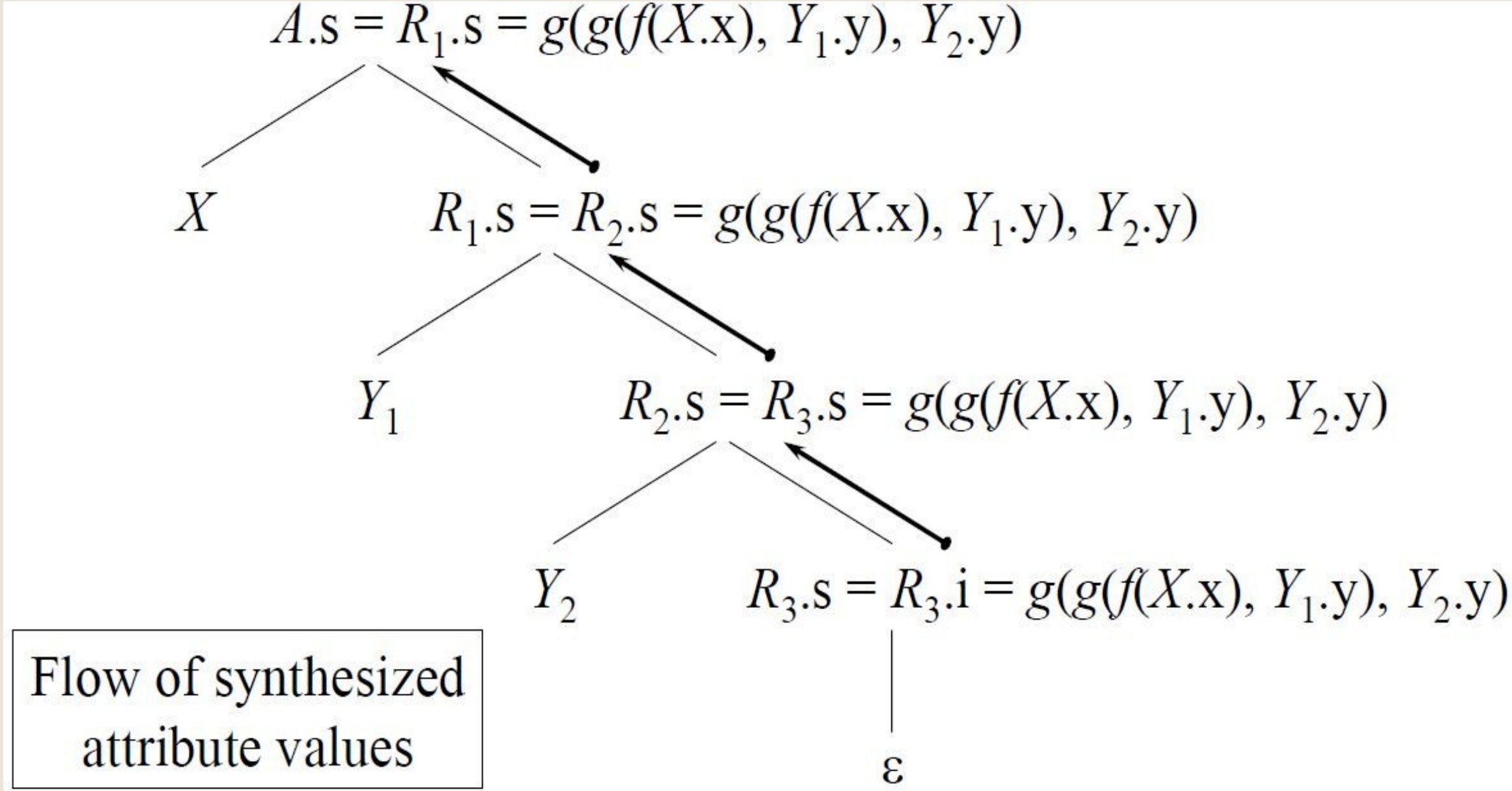
- The inherited attribute **R.i** is evaluated immediately before a use of **R** in the body, while the synthesized attributes **A.a** and **R.s** are evaluated at the ends of the productions
- Thus, whatever values are needed to compute these attributes will be available from what has been computed to the left

Evaluation Example

Evaluation of string **XYY**




A
X
R_1.i = f(X.x)
Y_1
R_2.i = g(f(X.x), Y_1.y)
Y_2
R_3.i = g(g(f(X.x), Y_1.y), Y_2.y)
ε
Flow of inherited attribute values


$A.s = R_1.s = g(g(f(X.x), Y_1.y), Y_2.y)$
X
$R_1.s = R_2.s = g(g(f(X.x), Y_1.y), Y_2.y)$
Y_1
$R_2.s = R_3.s = g(g(f(X.x), Y_1.y), Y_2.y)$
Y_2
$R_3.s = R_3.i = g(g(f(X.x), Y_1.y), Y_2.y)$
ε
Flow of synthesized attribute values

Bottom-Up Evaluation of Inherited Attributes

- L–attributed definitions are a simple subclass of inherited attributes that can be effectively implemented in most bottom-up parsers
- If we are able to identify the position of inherited attribute in the stack, then simple assignment of the value stored in the stack may be used to implement the L-attributed definition in bottom-up manner

Bottom-Up Evaluation of Inherited Attributes

- The method used in bottom-up evaluation of S-Attributed Definitions is extended with conditions:
  - *All embedding semantic actions in translation scheme should be in the end of the production rules*
  - *All inherited attributes should be copied into the synthesized attributes (most of the time synthesized attributes of new non-terminals)*
  - *Evaluate all semantic actions during reductions*

Removing Embedding Semantic Actions

- Insert marker non-terminals to remove embedded actions from translation schemes; that is, $A \rightarrow X\{ \text{ actions } \}Y$ is rewritten with marker non-terminal N as

$$A \rightarrow X\ N\ Y$$

$$N \rightarrow \varepsilon\ \{ \text{ actions } \}$$

- Problem: inserting a marker nonterminal may introduce a conflict in the parse table

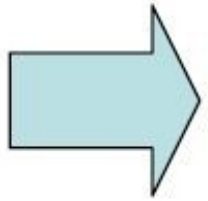
Removing Embedding Semantic Actions

$E \rightarrow T\ R$

$R \rightarrow +\ T\ \{\ \text{print}(\text{“+”})\ \}\ R1$

$R \rightarrow \varepsilon$

$T \rightarrow \textbf{id}\ \{\ \text{print}(\textbf{id}.\text{name})\ \}$



$E \rightarrow T\ R$

$R \rightarrow +\ T\ M\ R1$

$R \rightarrow \varepsilon$

$T \rightarrow \textbf{id}\ \{\ \text{print}(\textbf{id}.\text{name})\ \}$

$M \rightarrow \varepsilon\ \{\ \text{print}(\text{“+”})\ \}$

Example of Bottom-Up Evaluation

Consider the following grammar:

E → T {L.type = T.val} L *i.e. {print val[top] = val[top – 1]}*

L → L, id {settype(id.entry, L.type)} *i.e. {settype (val[top] = val[top – 3])}*

L → id {settype(id.entry, L.type)} *i.e. {settype (val[top] = val[top – 1])}*

T → int{T.val = int} *i.e. {val[top] = int}*

T → float{T.val = float} i.e. *{val[top] = float}*

Example of Bottom-Up Evaluation

Consider type declaration of variables in C: **float a,b;**

Given a string of the form **float id,id** a bottom-up evaluation can be traced as follows:

Input	State	Value (Stack)	Action
float id,id$	$		
id,id$	$float	float	shift
id,id$	$T	float	T → float (reduce)
,id$	$T id	float _	shift
,id$	$T L	float float	L → id
id$	$T L,	float float _	shift
$	$T L, id	float float _ _	shift
$	$T L	float float	L → L, id
$	$E	float	E → T L

Applications of Syntax-Directed Translation

- Syntax-directed translations are used for the construction of syntax trees
- The syntax-directed translation techniques are also applied for type checking and intermediate code generation
- SDT is used for executing arithmetic expressions
- SDT is used in the conversion from infix expression to postfix/prefix expression
- SDT is used in counting number of reductions
- It is also used for storing information into symbol