

SYMANTEC ANALYSIS

3RD PHASE OF COMPILER CONSTRUCTION

1

BEYOND SYNTAX

- There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
    int a, b, c, d;
    { ... }
fee() {
    int f[3],g[0],
        h, i, j, k;
    char *p;
    fie(h,i,"ab",j, k);
    k = f * i + j;
    h = g[17];
    printf("<%s,%s>.Wn",
        p,q);
    p = 10;
}
```

What is wrong with this program?
(let me count the ways ...)

SEMANTICS

- There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
  int a, b, c, d;
  { ... }
fee() {
  int f[3],g[0],
    h, i, j, k;
  char *p;
  fie(h,i,"ab",j, k);
  k = f * i + j;
  h = g[17];
  printf("<%s,%s>.Wn",
    p,q);
  p = 10;
}
```

What is wrong with this program?

(let me count the ways ...)

- declared g[0], used g[17]
- wrong number of args to fie()
- "ab" is not an int
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

All of these are "deeper than syntax"

SEMANTIC ANALYSIS

- 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 production rules.
- Evaluation of these semantic rules may:
 - generate intermediate codes
 - put information into the symbol table
 - perform type checking
 - issue error messages
 - perform almost any activities.



ROLE OF SEMANTIC ANALYZER

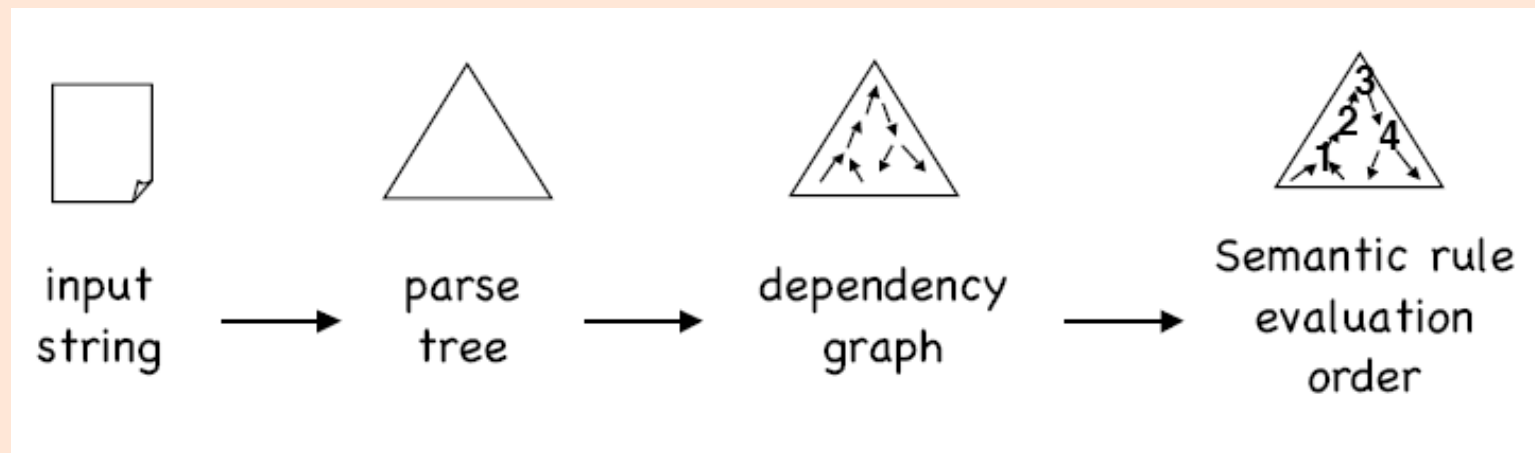
- A **semantic analyzer** checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free language used in syntax analyzers, so they are integrated with attributes (semantic rules) , resulting in
 - **Syntax-directed translation,**
 - **Attribute grammars**
- Ex:

$newval := oldval + 12$

The type of the identifier *newval* must match with type of the expression (*oldval*+12)

SYNTAX-DIRECTED TRANSLATION

- In syntax-directed translation, we attach ATTRIBUTES to grammar symbols.
- The values of the attributes are computed by SEMANTIC RULES associated with grammar productions.
- Conceptually, we have the following flow:



- In practice, however, we do everything in a single pass.

ATTRIBUTES

- Attribute values may represent

- Numbers (literal constants)
- Strings (literal constants)
- Memory locations, such as a frame index of a local variable or function argument
- A data type for type checking of expressions
- Scoping information for local declarations
- Intermediate program representations

- Attributes are two tuple value,

`<attribute name,="" attribute="" value="">`

- **There are two type of Attributes:**

- Synthesized Attributes
- Inherited Attributes

SYNTAX-DIRECTED TRANSLATION

- There are two ways to represent the semantic rules we associate with grammar symbols.
 - **SYNTAX-DIRECTED DEFINITIONS (SDDs)** do not specify the order in which semantic actions should be executed
 - **TRANSLATION SCHEMES** explicitly specify the ordering of the semantic actions.
- SDDs are higher level; translation schemes are closer to an implementation

3.1 SYNTAX-DIRECTED DEFINITIONS

SYNTAX-DIRECTED DEFINITION

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

$$b = f(c_1, c_2, \dots, c_n) \quad \text{where } f \text{ is a function,}$$

and b can be one of the followings:

- ➔ b is a **synthesized attribute** of A and c_1, c_2, \dots, c_n are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

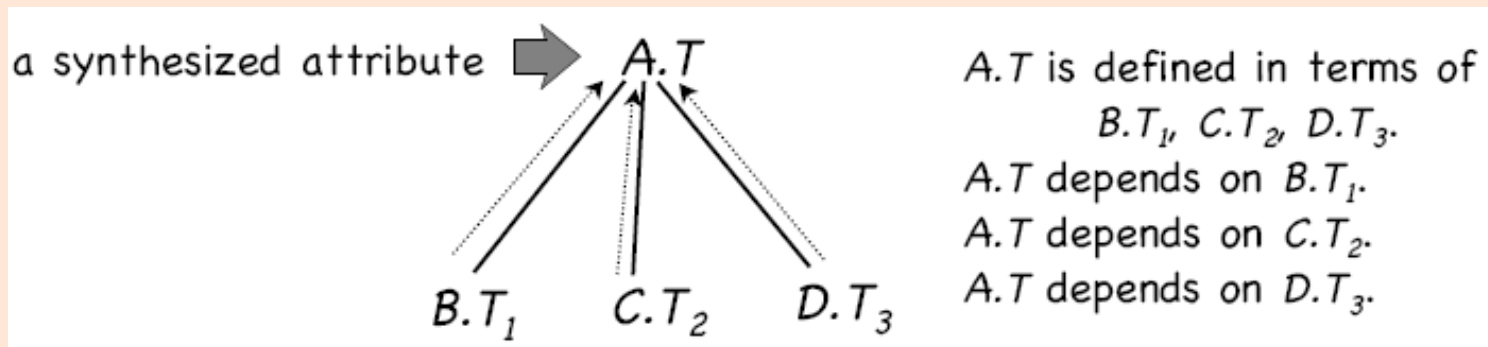
OR

- ➔ b is an **inherited attribute**, one of the grammar symbols in α (on the right side of the production), and c_1, c_2, \dots, c_n are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

in either case, we say b **DEPENDS** on c_1, c_2, \dots, c_k .

SYNTHESIZED ATTRIBUTES

- Synthesized attributes depend only on the attributes of children. They are the most common attribute type.



- Synthesized attributes never take values from their parent nodes or any sibling nodes.
- The non terminal concerned must be in the head of production.
- Terminals have synthesized attributes which are the lexical values (denoted by lexval) generated by the lexical analyzer.

EXAMPLE SDD: DESK CALCULATOR (Ex. 1)

Production

$L \rightarrow E$ **return**

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow$ **digit**

Semantic Rules

$\text{print}(E.\text{val})$

$E.\text{val} = E_1.\text{val} + T.\text{val}$

$E.\text{val} = T.\text{val}$

$T.\text{val} = T_1.\text{val} * F.\text{val}$

$T.\text{val} = F.\text{val}$

$F.\text{val} = E.\text{val}$

$F.\text{val} =$ **digit**. lexval

- Symbols E, T, and F are associated with a synthesized attribute *val*.
- Terminals do not have inherited attributes, it is assumed that their value is evaluated by the lexical analyzer.
- Here, token **digit** has a synthesized attribute *lexval*

SYNTAX-DIRECTED DEFINITION (EX. 2)

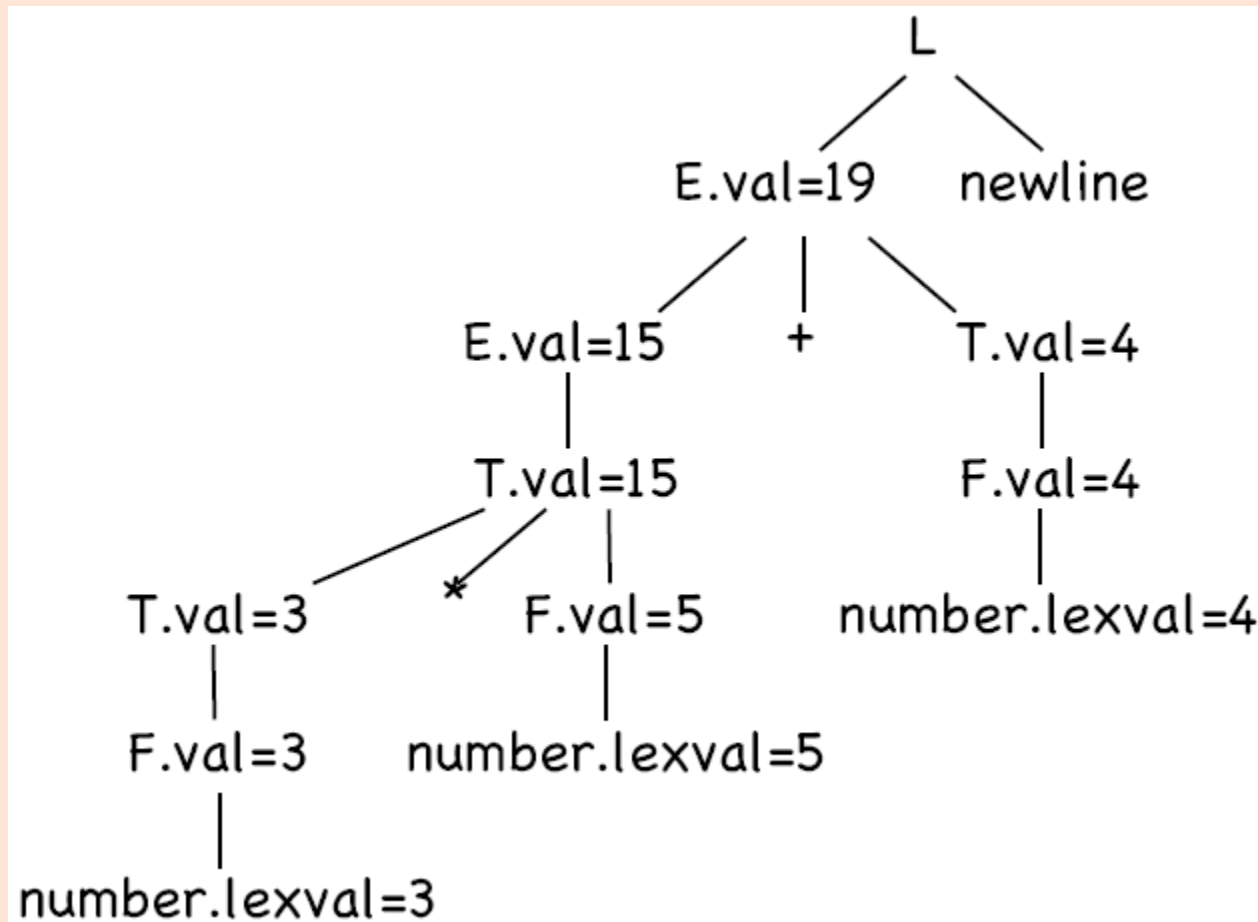
<u>Production</u>	<u>Semantic Rules</u>
$E \rightarrow E_1$	$E.loc = \text{newtemp}(), E.code = E_1.code$
$E \rightarrow E_1 + T$	$E.loc = \text{newtemp}(), E.code = E_1.code \parallel T.code \parallel \text{add } E_1.loc, T.loc, E.loc$
$E \rightarrow T$	$E.loc = T.loc, E.code = T.code$
$T \rightarrow T_1 * F$	$T.loc = \text{newtemp}(), T.code = T_1.code \parallel F.code \parallel \text{mult } T_1.loc, F.loc, T.loc$
$T \rightarrow F$	$T.loc = F.loc, T.code = F.code$
$F \rightarrow (E)$	$F.loc = E.loc, F.code = E.code$
$F \rightarrow \mathbf{id}$	$F.loc = \mathbf{id.name}, F.code = ""$

- Symbols E, T, and F are associated with synthesized attributes *loc* and *code*.
- The token **id** has a synthesized attribute *name* (it is assumed that it is evaluated by the lexical analyzer).
- It is assumed that \parallel is the string concatenation operator.

CALCULATING SYNTHESIZED ATTRIBUTES

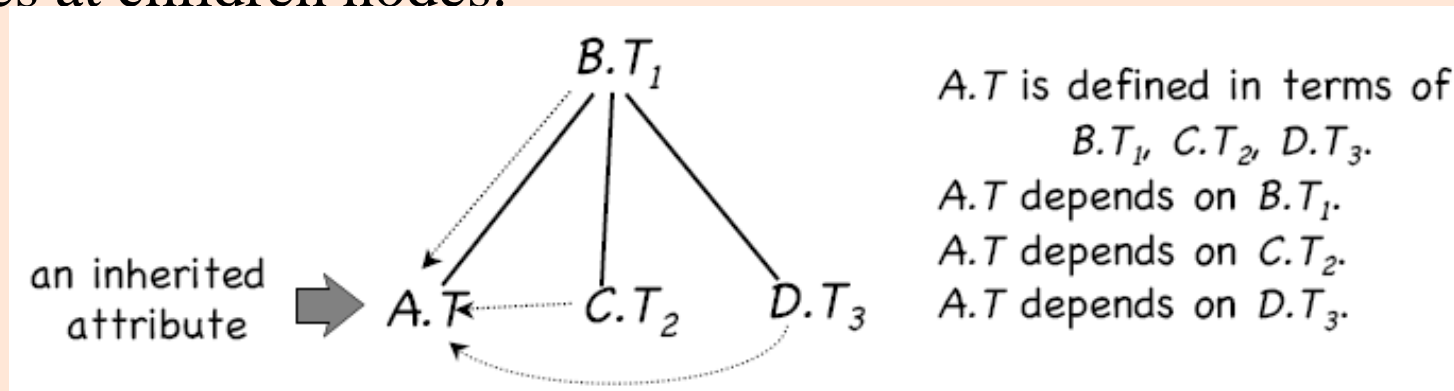
Input string: $3*5+4$ return

Annotated tree:



INHERITED ATTRIBUTES

- An inherited attribute is defined in terms of the attributes of the node's parents and/or siblings. They cannot be defined in terms of attribute values at children nodes.



- Inherited attributes are often used in compilers for passing contextual information forward, for example, the type keyword in a variable declaration statement.
- Inherited attributes are useful when the structure of a parse tree does not match the abstract syntax of the source code.

EXAMPLE OF SDD WITH AN INHERITED ATTRIBUTE

- Suppose we want to describe declarations like “real x, y, z”

	Production	Semantic Rules	
1)	$D \rightarrow T L$	$L.in := T.type$	
2)	$T \rightarrow \text{int}$	$T.type := \text{integer}$	
3)	$T \rightarrow \text{real}$	$T.type := \text{real}$	
4)	$L \rightarrow L_1, \text{id}$	$L_1.in := L.in$ $\text{addtype}(\text{id.entry}, L.in)$	addtype() is just a procedure that sets the type field in the symbol table.
5)	$L \rightarrow \text{id}$	$\text{addtype}(\text{id.entry}, L.in)$	

- Productions 4 and 5 have a rule in which a function addType is called with two arguments:
 - id.entry, a lexical value that points to a symbol-table object, and
 - L.inh, the type being assigned to every identifier on the list.
- L.in** is **inherited** since it depends on a sibling or parent. L is taking value from T ($D \rightarrow TL$)

ATTRIBUTE GRAMMAR

- A semantic rule $b=f(c_1, c_2, \dots, c_n)$ indicates that the attribute b *depends on* attributes c_1, c_2, \dots, c_n .
- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values.
- An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

SIDE EFFECTS

- **Side Effects:** Evaluation of semantic rules may generate intermediate codes, may put information into the symbol table, may perform type checking and may issue error messages. These are known as side effects.
- **Semantic Rules with Controlled Side Effects:**

In practice translation involves side effects. Attribute grammars have no side effects and allow any evaluation order consistent with dependency graph whereas translation schemes impose left-to-right evaluation and allow schematic actions to contain any program fragment.
- **Ways to Control Side Effects**
 1. Permit incidental side effects that do not disturb attribute evaluation.
 2. Impose restrictions on allowable evaluation orders, so that the same translation is produced for any allowable order

NEED FOR DEPENDENCY GRAPH

- An SDD with both inherited and synthesized attributes does not ensure any guaranteed order; even it may not have an order at all.
- For example, consider nonterminals A and B, with synthesized and inherited attributes A.s and B.i, respectively, along with the production and rules as:

Production

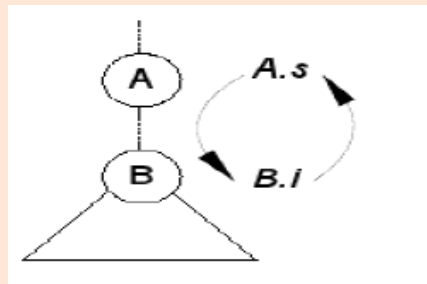
$A \rightarrow B$

Semantic Rule

$A.s = B.i$

$B.i = A.s + 1$

- 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.



The circular dependency of A.s and b.s on one another

3.2 DEPENDENCY GRAPHS

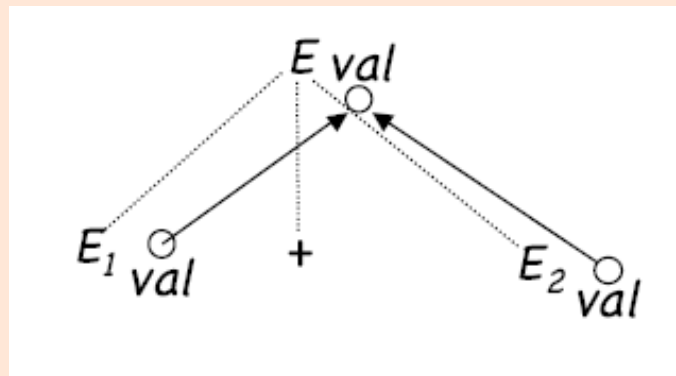
20

DEPENDENCY GRAPHS

- Dependency Graph predicts the flow of information among the attribute instances in a particular parse tree.
- The interdependencies among the inherited and synthesized attributes at the node in the parse tree is depicted by *dependency graph*.
- If an attribute b depends on attribute c, then attribute b has to be evaluated AFTER c.
- **DEPENDENCY GRAPHS** visualize:
 - Each attribute is a node
 - We add edges from the node for attribute c to the node for attribute b, if b depends on c.
 - For procedure calls, we introduce a dummy synthesized attribute that depends on the parameters of the procedure calls.

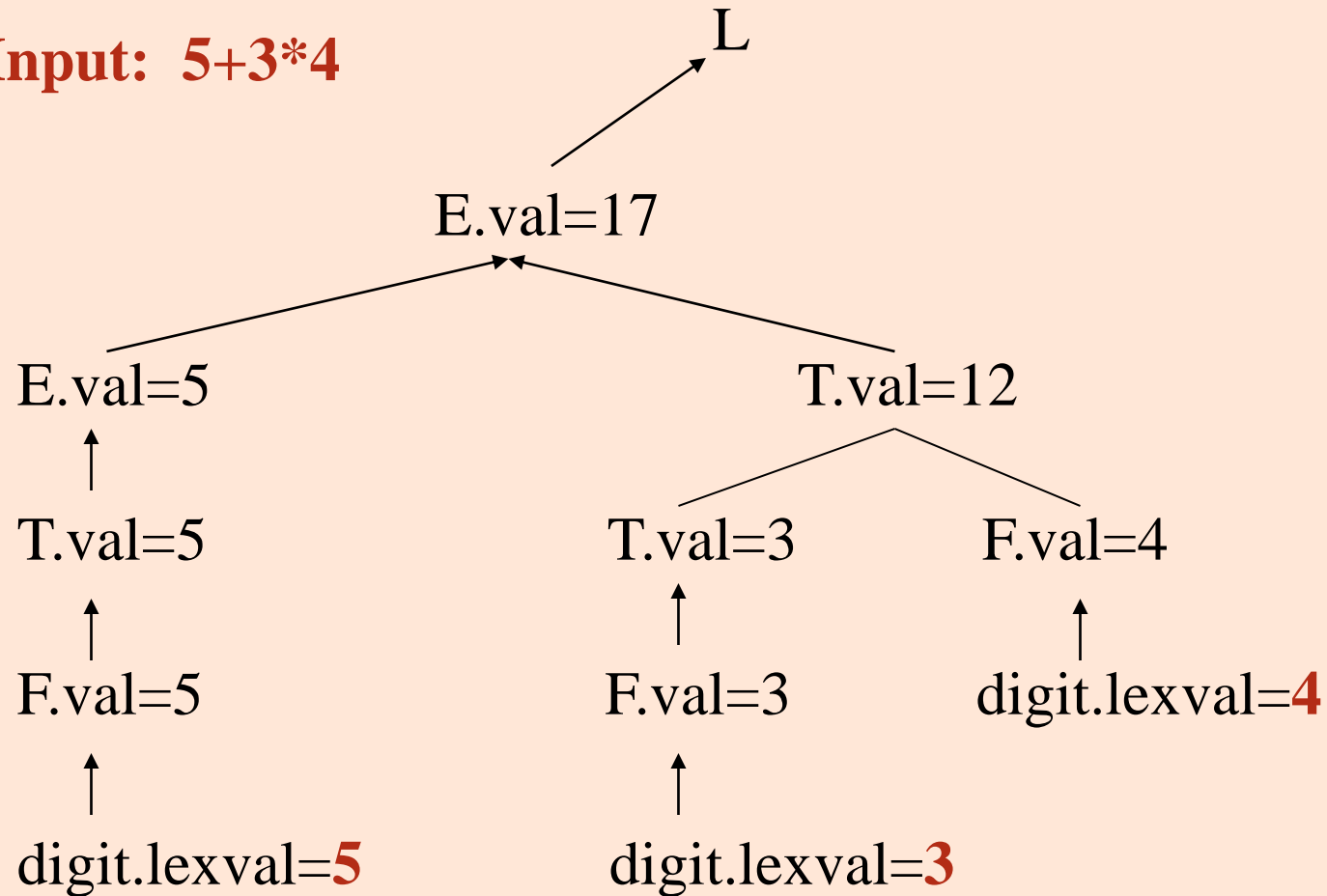
DEPENDENCY GRAPH EXAMPLE

- Production
- $E \rightarrow E_1 + E_2$
- Semantic Rule
- $E.val := E_1.val + E_2.val$
- Wherever this rule appears in the parse tree we draw:



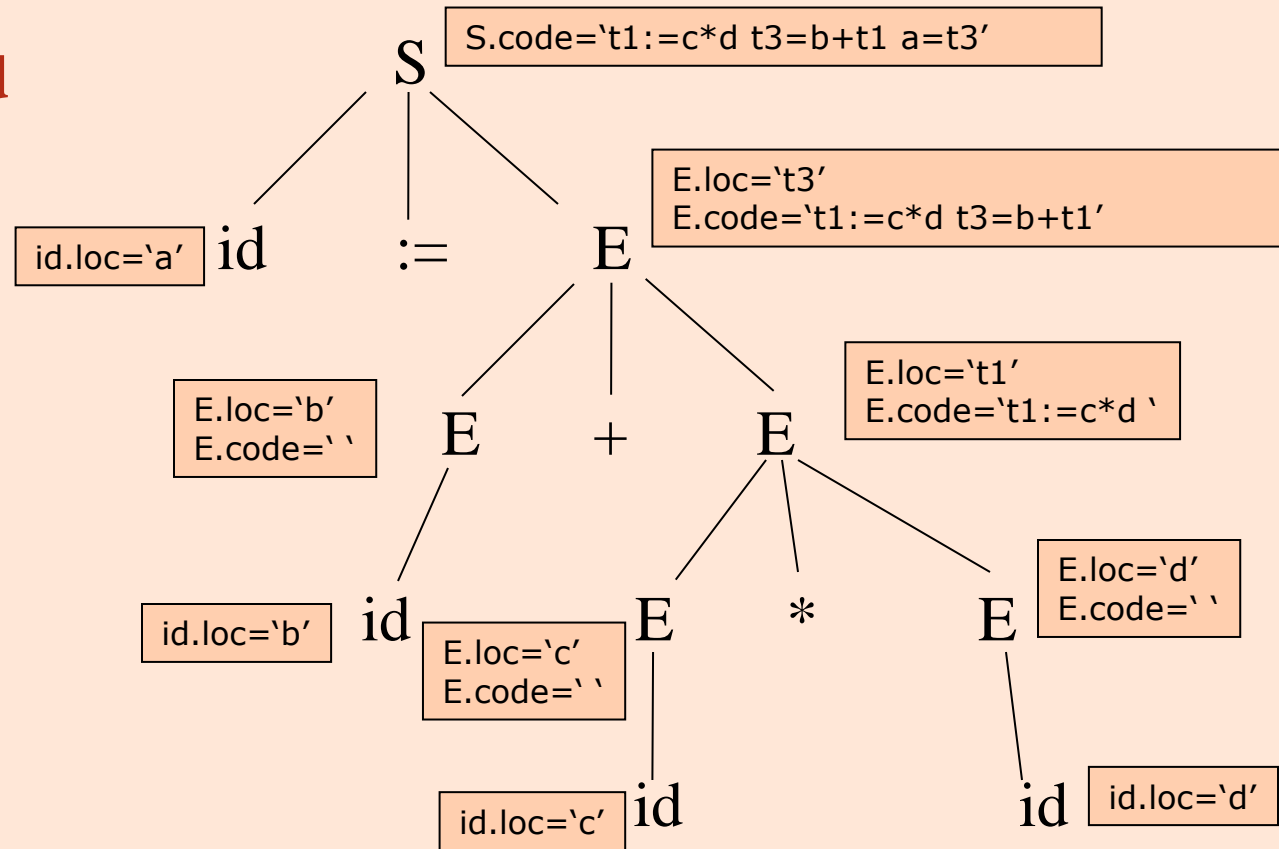
DEPENDENCY GRAPH EXAMPLE

Input: 5+3*4



DEPENDENCY GRAPH – SYNTHETIZED ATTRIBUTES

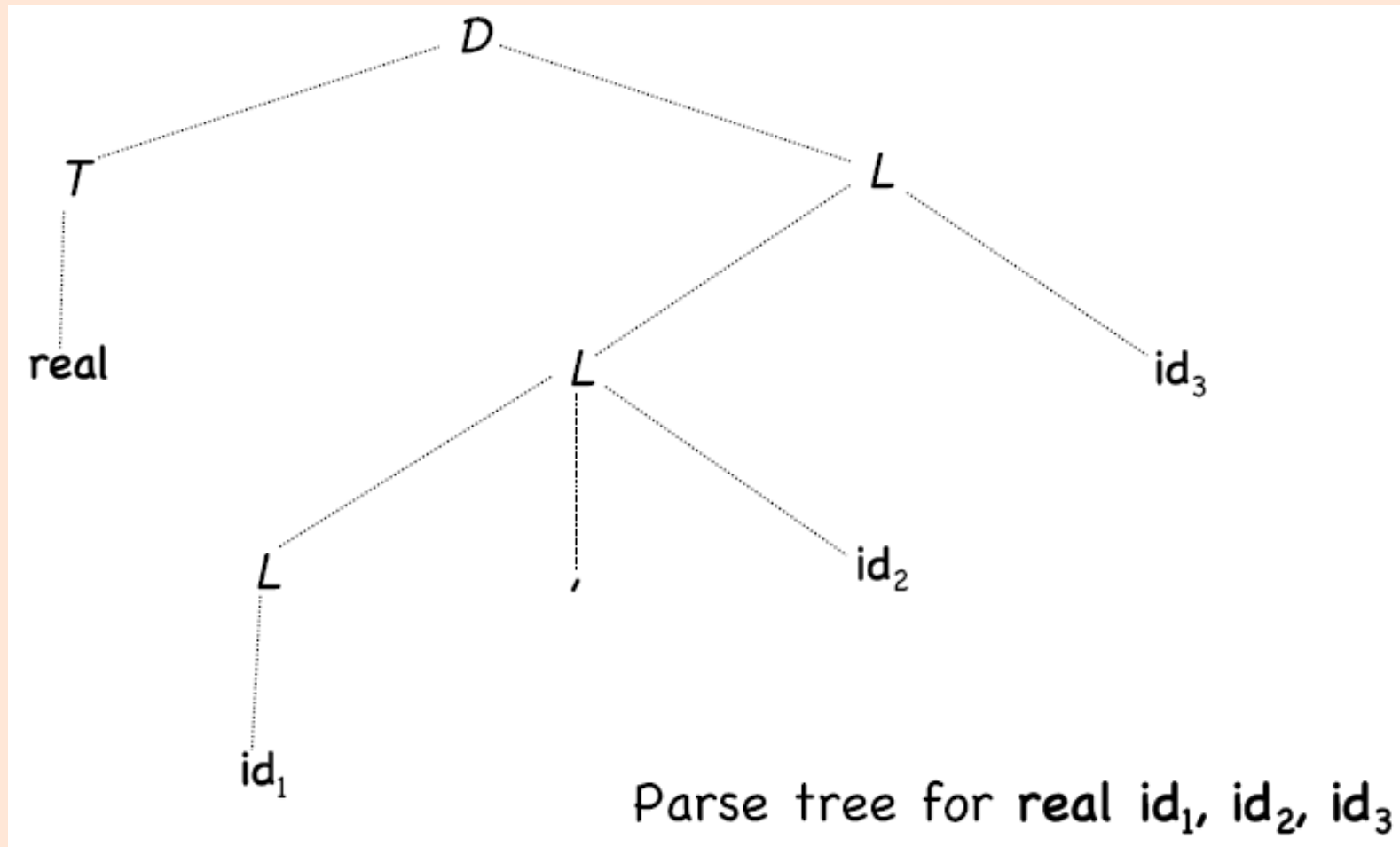
Input $a := b + c * d$



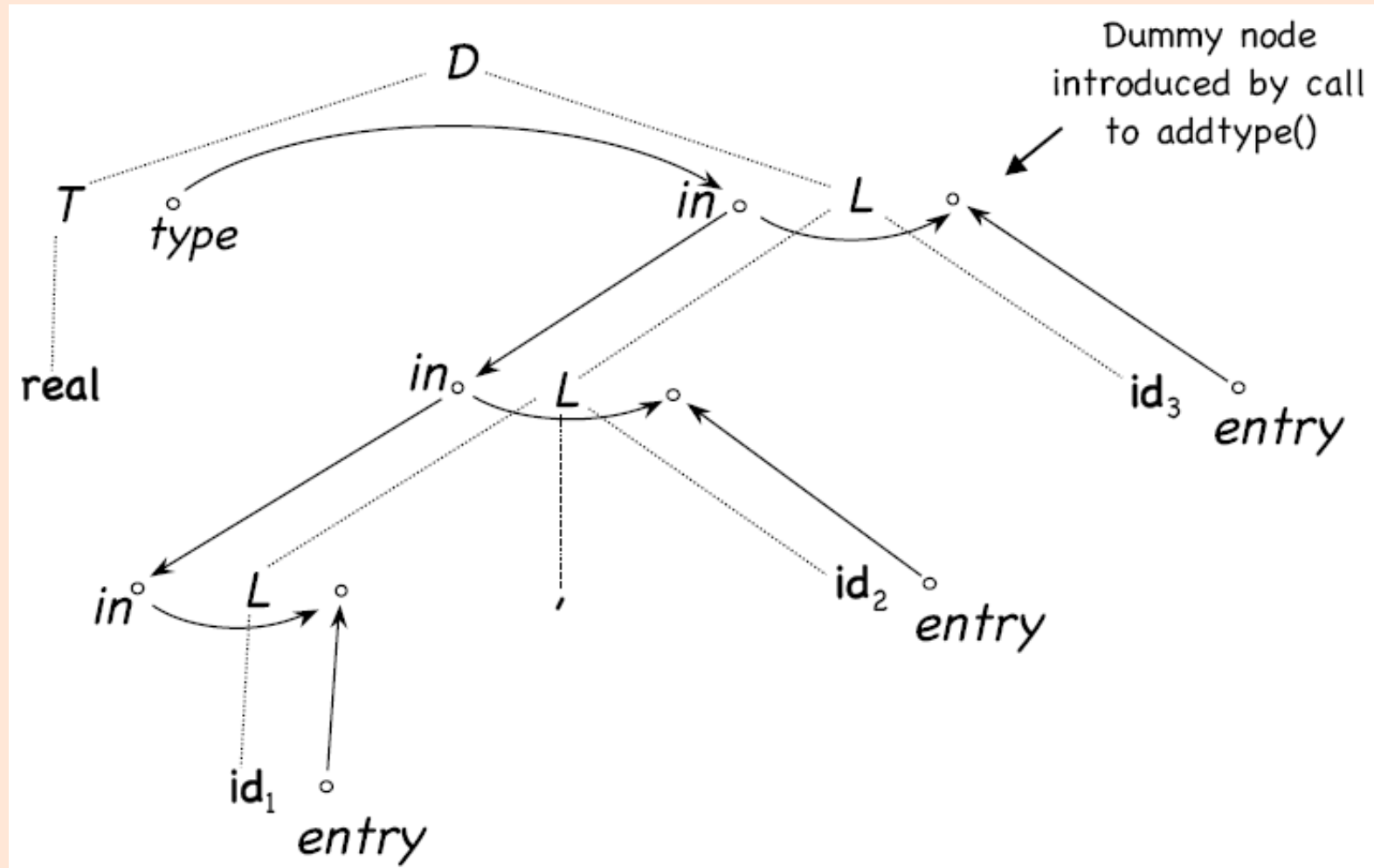
$E \rightarrow E_1 + T$
 $E \rightarrow T$
 $T \rightarrow T_1 * F$
 $T \rightarrow F$
 $F \rightarrow (E)$
 $F \rightarrow \mathbf{id}$

$E.\text{loc} = \text{newtemp}(), E.\text{code} = E_1.\text{code} \parallel T.\text{code} \parallel \text{add } E_1.\text{loc}, T.\text{loc}, E.\text{loc}$
 $E.\text{loc} = T.\text{loc}, E.\text{code} = T.\text{code}$
 $T.\text{loc} = \text{newtemp}(), T.\text{code} = T_1.\text{code} \parallel F.\text{code} \parallel \text{mult } T_1.\text{loc}, F.\text{loc}, T.\text{loc}$
 $T.\text{loc} = F.\text{loc}, T.\text{code} = F.\text{code}$
 $F.\text{loc} = E.\text{loc}, F.\text{code} = E.\text{code}$
 $F.\text{loc} = \mathbf{id}.\text{name}, F.\text{code} = \text{" "}$

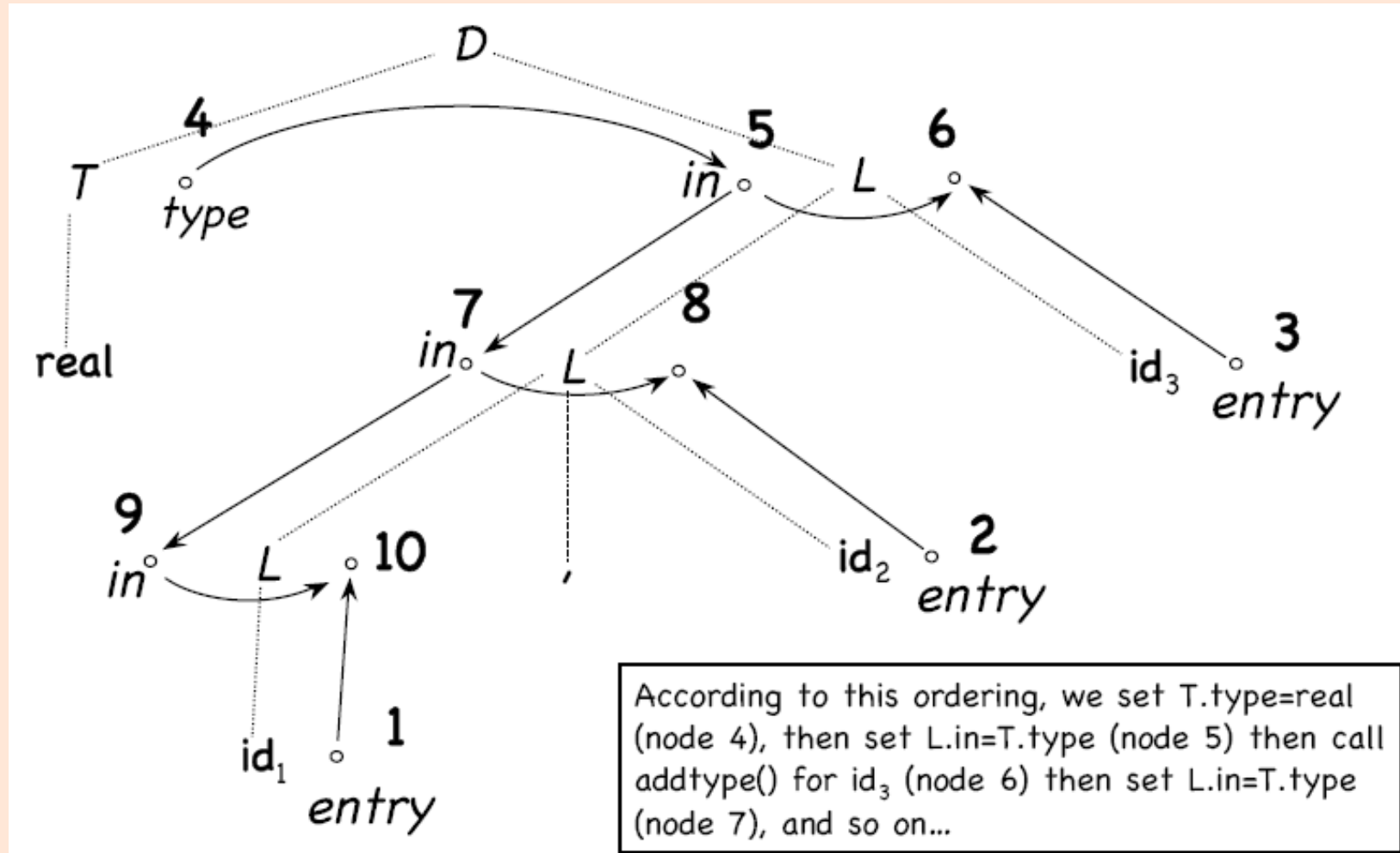
DEPENDENCY GRAPH – INHERITED ATTRIBUTES



DEPENDENCY GRAPH – INHERITED ATTRIBUTES

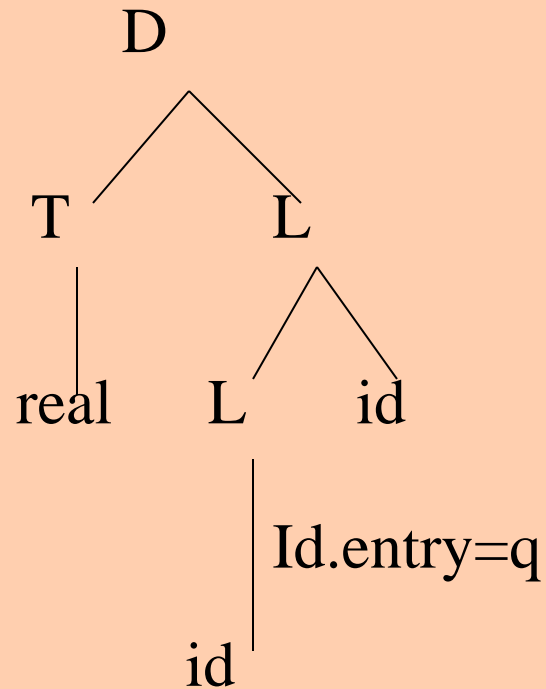


DEPENDENCY GRAPH – INHERITED ATTRIBUTES



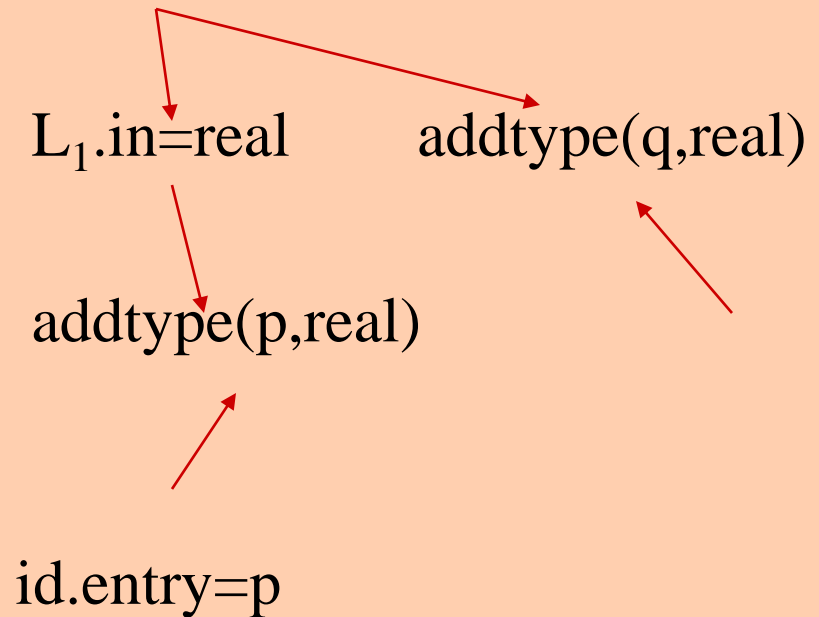
DEPENDENCY GRAPH

Input: real p q



parse tree

T.type=real \longrightarrow L.in=real



dependency graph

FINDING A VALID EVALUATION ORDER

TOPOLOGICAL SORT OF THE DEPENDENCY GRAPH

- An ordering embeds a directed graph into a linear order, and is called a *topological sort* of the graph
- A topological sort of a directed acyclic graph orders the nodes so that for any nodes a and b if there is a flow $a \rightarrow b$, a appears BEFORE b in the ordering.
- If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree. If there are no cycles, then there is always at least one topological sort.
- There are many possible topological orderings for a DAG.
- Each of the possible orderings gives a valid order for evaluation of the semantic rules.

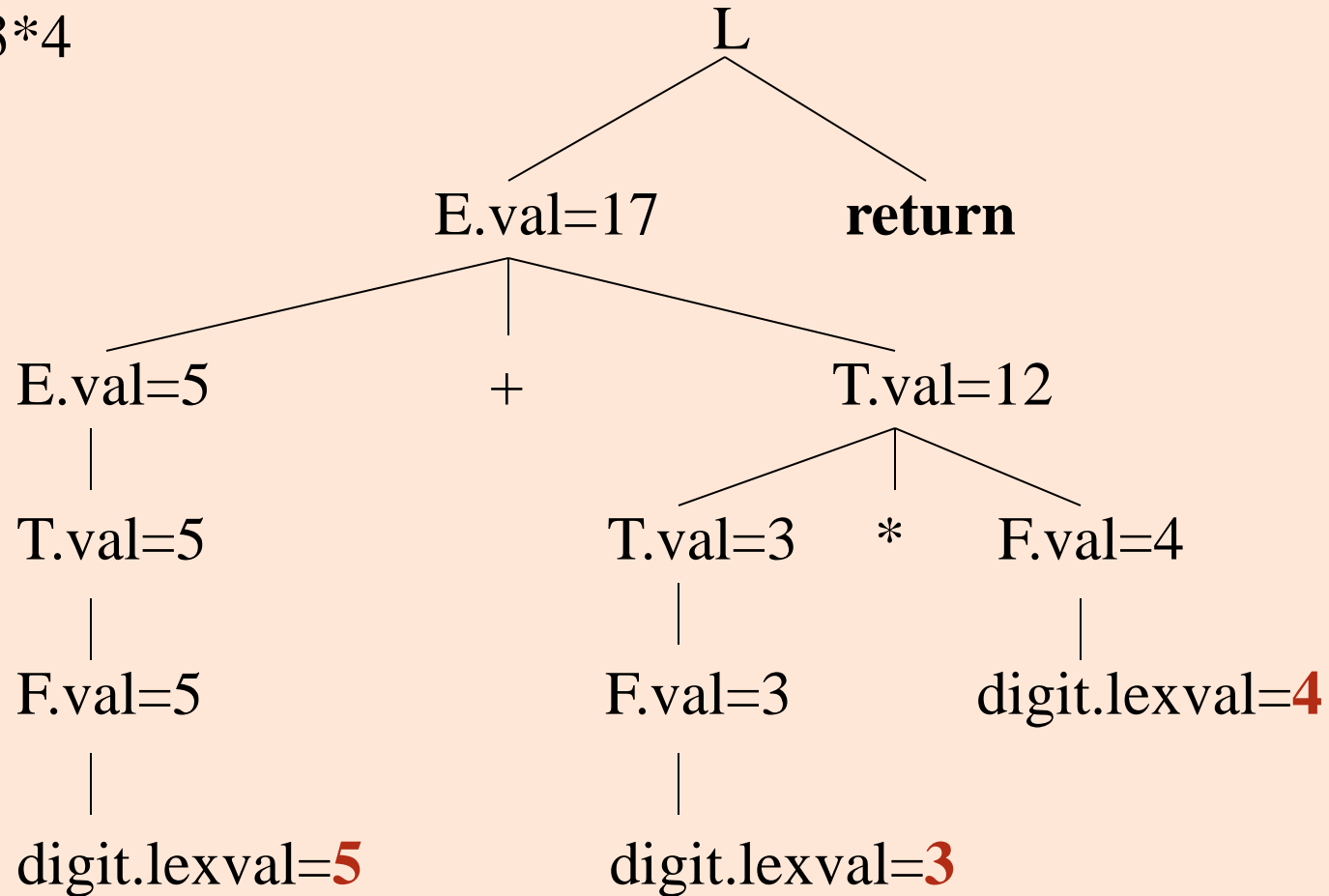
3.2 SYNTAX TREES AND PARSE TREES

ANNOTATED PARSE TREE

- 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.
- The order of these computations depends on the dependency graph induced by the semantic rules.

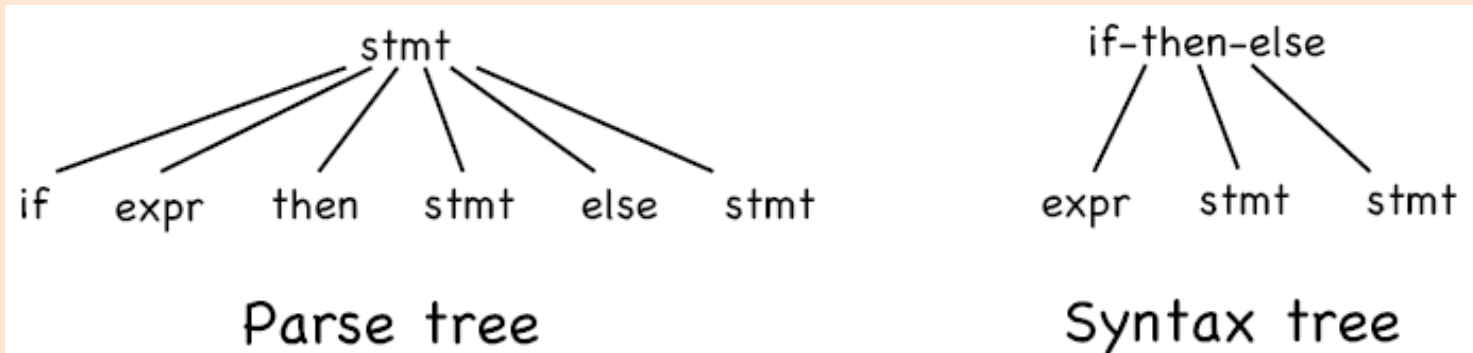
ANNOTATED PARSE TREE -- EXAMPLE

Input: 5+3*4

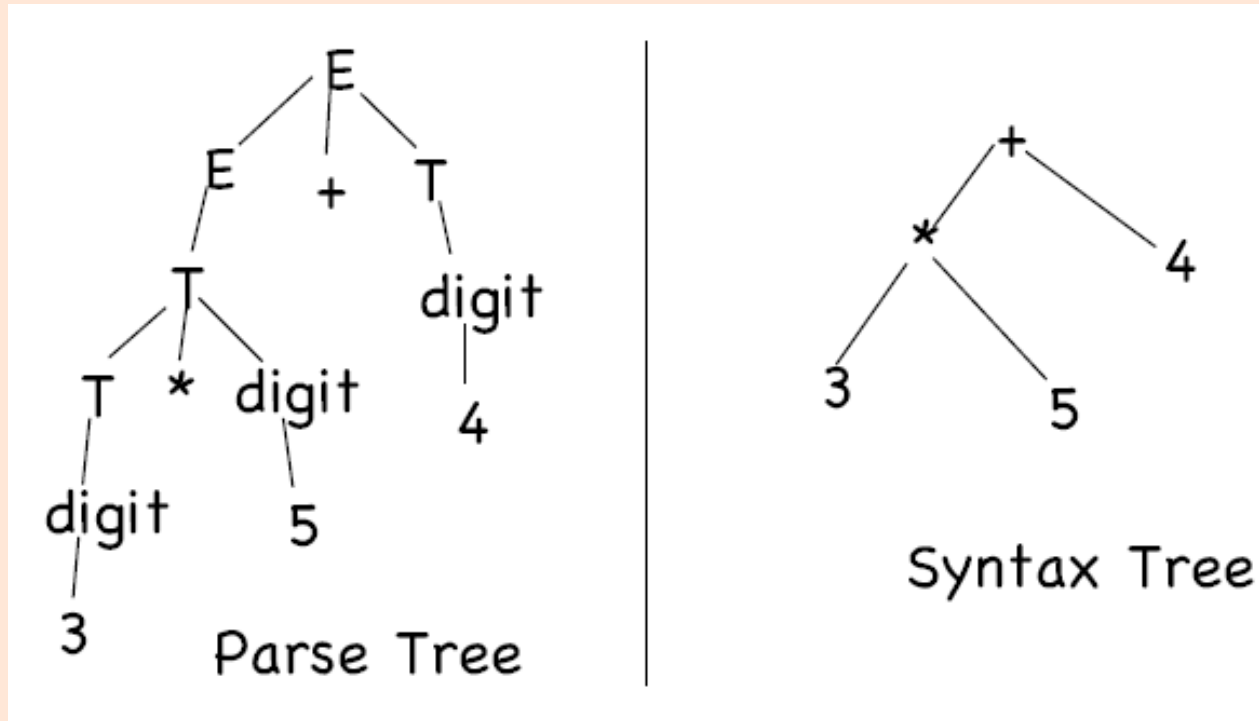


SYNTAX TREE CONSTRUCTION

- One thing SDDs are useful for is construction of SYNTAX TREES.
- A syntax tree is a condensed form of parse tree.
- Syntax trees are useful for representing programming language constructs like expressions and statements.
- They help compiler design by decoupling parsing from translation.



SYNTAX TREES



- Leaf nodes for operators and keywords are removed.
- Internal nodes corresponding to uninformative non-terminals are replaced by the more meaningful operators.

SDD FOR SYNTAX TREE CONSTRUCTION

- We need some functions to help us build the syntax tree:
 - `mknode(op,left,right)` constructs an operator node with label `op`, and two children, `left` and `right`
 - `mkleaf(id,entry)` constructs a leaf node with label `id` and a pointer to a symbol table entry
 - `mkleaf(num,val)` constructs a leaf node with label `num` and the token's numeric value `val`
- Use these functions to build a syntax tree for **a-4+c**:
 - `P1 := mkleaf(id, st_entry_for_a)`
 - `P2 := ...`

SDD FOR SYNTAX TREE CONSTRUCTION

This is an example of S-attributed definition.

Production

$E \rightarrow E_1 + T$

$E \rightarrow E_1 - T$

$E \rightarrow T$

$T \rightarrow (E)$

$T \rightarrow id$

$T \rightarrow num$

Semantic Rules

$E.nptr := mknode('+', E_1.nptr, T.nptr)$

$E.nptr := mknode('-', E_1.nptr, T.nptr)$

$E.nptr := T.nptr$

$T.nptr := E.nptr$

$T.nptr := mkleaf(id, id.entry)$

$T.nptr := mkleaf(num, num.val)$

- Try to derive the annotated parse tree for **a-4+c**.

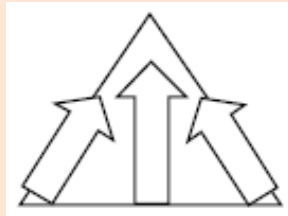
3.3 EVALUATING SDDs BOTTOM-UP

S-ATTRIBUTED DEFINITIONS

- Syntax-directed definitions are used to specify syntax-directed translations.
- To create a translator for an arbitrary syntax-directed definition can be difficult.
- Two sub-classes of the syntax-directed definitions are:
 - **S-Attributed Definitions:** only synthesized attributes used in the syntax-directed definitions.
 - **L-Attributed Definitions:** in addition to synthesized attributes, we may also use inherited attributes in a restricted fashion.
- Implementations of S-attributed Definitions are a little bit easier than implementations of L-Attributed Definitions

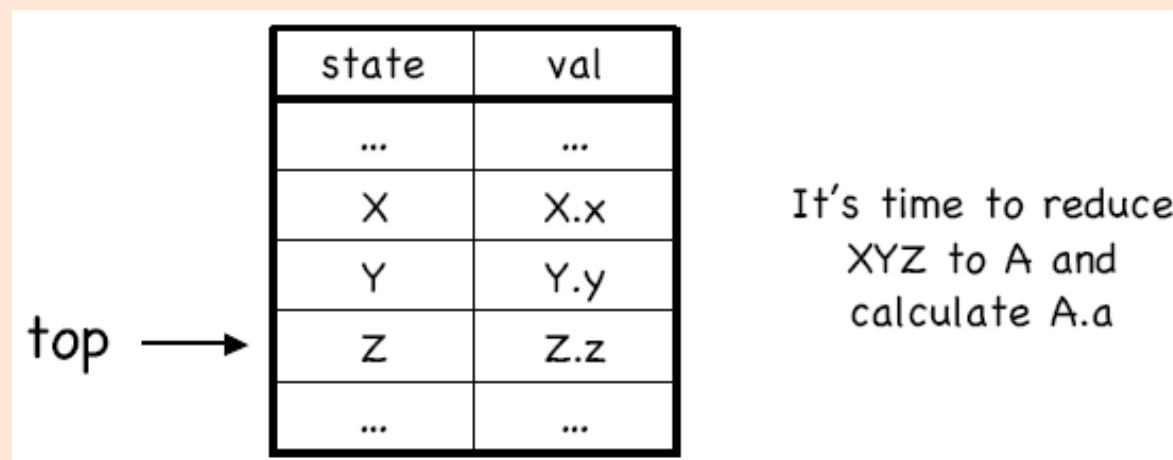
SYNTHESIZED ATTRIBUTES

- If a **SDD** has **synthesized attributes ONLY**, it is called a **S-ATTRIBUTED DEFINITION**.
- S-attributed definitions are convenient since the attributes can be calculated in a bottom-up traversal of the parse tree.



BOTTOM-UP EVALUATION OF S-ATTRIBUTED DEFINITIONS

- Put the values of the synthesized attributes of the grammar symbols into a parallel stack.
- Parser's stack stores grammar symbols AND attribute values.
- For every production $A \rightarrow XYZ$ with semantic rule $A.a := f(X.x, Y.y, Z.z)$, before XYZ is reduced to A , we should already have $X.x$, $Y.y$ and $Z.z$ on the stack. (Here all attributes are synthesized.)

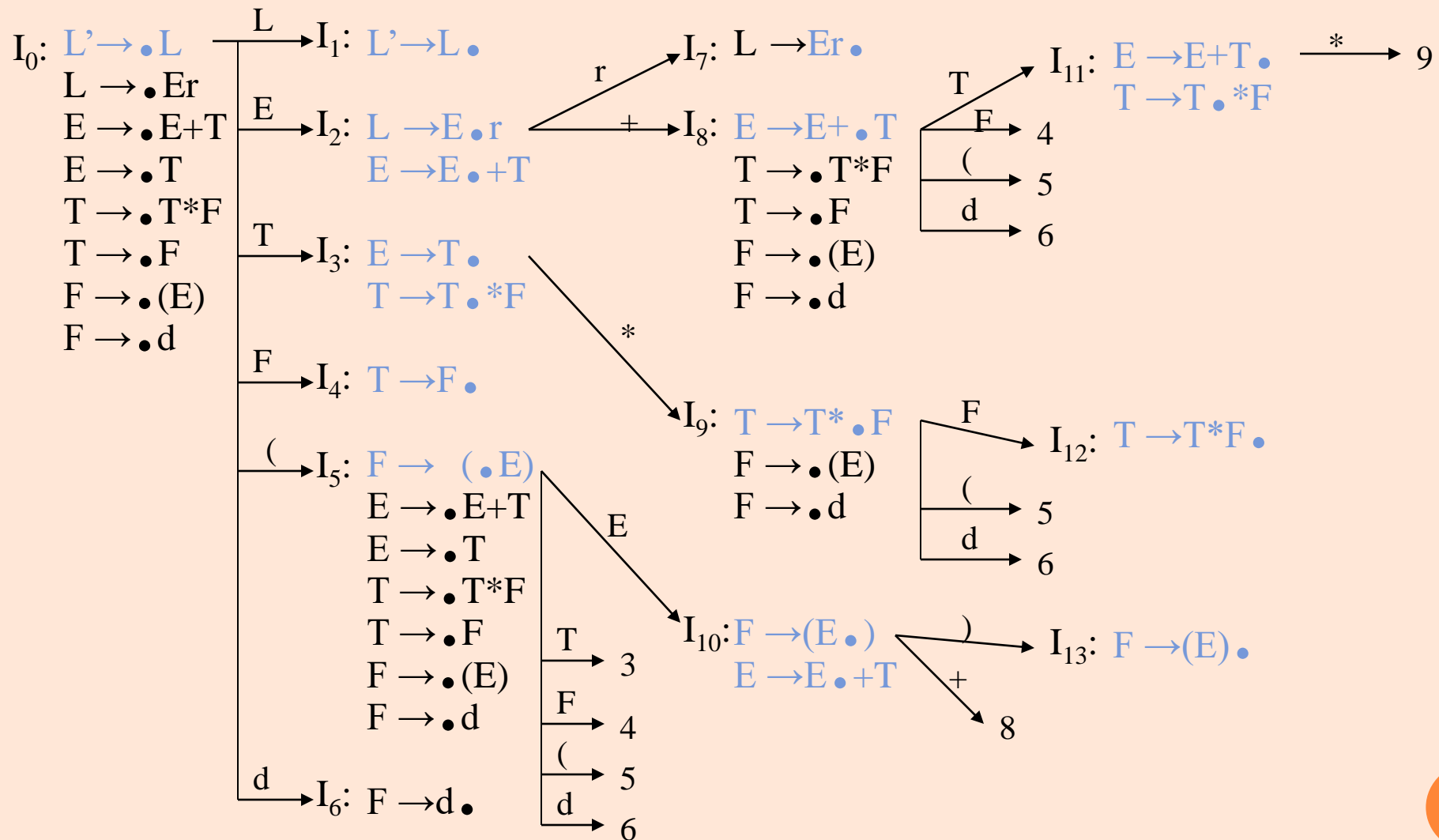


DESK CALCULATOR EXAMPLE

<u>Production</u>	<u>Semantic Rule</u>	<u>Code</u>
$L \rightarrow E \text{ return}$	$\text{print}(E.\text{val})$	$\text{print val}[\text{top}-1]$
$E \rightarrow E1 + T$	$E.\text{val} := E1.\text{val} + T.\text{val}$	$\text{val}[\text{newtop}] = \text{val}[\text{top}-2] + \text{val}[\text{top}]$
$E \rightarrow T$	$E.\text{val} := T.\text{val}$	$\text{/*newtop==top, so nothing to do*/}$
$T \rightarrow T1 * F$	$T.\text{val} := T1.\text{val} \times F.\text{val}$	$\text{val}[\text{newtop}] = \text{val}[\text{top}-2] + \text{val}[\text{top}]$
$T \rightarrow F$	$T.\text{val} := F.\text{val}$	$\text{/*newtop==top, so nothing to do*/}$
$F \rightarrow (E)$	$F.\text{val} := E.\text{val}$	$\text{val}[\text{newtop}] = \text{val}[\text{top}-1]$
$F \rightarrow \text{digit}$	$F.\text{val} := \text{digit.lexval}$	$\text{/*newtop==top, so nothing to do*/}$

- Symbol E, T and F are associated with an attribute num.
- Token **digit** the implementation of the semantic rule for a bottom-up parser.
- 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*).
- The above model is suited for a desk calculator where the purpose is to evaluate and to generate code.

CANONICAL LR(0) COLLECTION FOR THE GRAMMAR



BOTTOM-UP EVALUATION -- EXAMPLE

- At each shift of **digit**, we also push **digit.lexval** into *val-stack*.

<u>stack</u>	<u>val-stack</u>	<u>input</u>	<u>action</u>	<u>semantic rule</u>
0		5+3*4r	s6	d.lexval(5) into val-stack
0id6	5	+3*4r	F→id	F.val=id.lexval – do nothing
0F4	5	+3*4r	T→F	T.val=F.val – do nothing
0T3	5	+3*4r	E→T	E.val=T.val – do nothing
0E2	5	+3*4r	s8	push empty slot into val-stack
0E2+8	5-	3*4r	s6	d.lexval(3) into val-stack
0E2+8id6	5-3	*4r	F→id	F.val=d.lexval – do nothing
0E2+8F4	5-3	*4r	T→F	T.val=F.val – do nothing
0E2+8T11	5-3	*4r	s9	push empty slot into val-stack
0E2+8T11*9	5-3-	4r	s6	d.lexval(4) into val-stack
0E2+8T11*9id6	5-3-4	r	F→id	F.val=d.lexval – do nothing
0E2+8T11*9F12	5-3-4	r	T→T*F	T.val=T ₁ .val*F.val
0E2+8T11	5-12	r	E→E+T	E.val=E ₁ .val+T.val
0E2	17	r	s7	push empty slot into val-stack
0E2r7	17-	\$	L→Er	print(17), pop empty slot from val-stack
0L1	17	\$	acc	

TOP-DOWN EVALUATION OF S-ATTRIBUTED DEFINITIONS

<u>Productions</u>	<u>Semantic Rules</u>
$A \rightarrow B$	$\text{print}(B.n0), \text{print}(B.n1)$
$B \rightarrow 0 B_1$	$B.n0 = B_1.n0 + 1, B.n1 = B_1.n1$
$B \rightarrow 1 B_1$	$B.n0 = B_1.n0, B.n1 = B_1.n1 + 1$
$B \rightarrow \varepsilon$	$B.n0 = 0, B.n1 = 0$

where B has two synthesized attributes (n0 and n1).

TOP-DOWN EVALUATION OF S-ATTRIBUTED DEFINITIONS

- Remember that: In a recursive predicate parser, each non-terminal corresponds to a procedure.

```
procedure A() {  
    call B();  
}
```

$A \rightarrow B$

```
procedure B() {  
    if (currtoken=0) { consume 0; call B(); }  
    else if (currtoken=1) { consume 1; call B(); }  
    else if (currtoken=$) {} // $ is end-marker  
    else error("unexpected token");  
}
```

$B \rightarrow 0 B$

$B \rightarrow 1 B$

$B \rightarrow \epsilon$

TOP-DOWN EVALUATION (OF S-ATTRIBUTED DEFINITIONS)

```
procedure A() {
```

```
  int n0,n1;
```

```
  call B(&n0,&n1);
```

```
  print(n0); print(n1);
```

```
}
```

```
procedure B(int *n0, int *n1) {
```

```
  if (currtoken=0)
```

```
    {int a,b; consume 0; call B(&a,&b); *n0=a+1; *n1=b;}
```

```
  else if (currtoken=1)
```

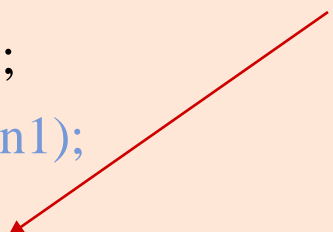
```
    { int a,b; consume 1; call B(&a,&b); *n0=a; *n1=b+1; }
```

```
  else if (currtoken=$) { *n0=0; *n1=0; } // $ is end-marker
```


```
  else error("unexpected token");
```

```
}
```

Synthesized attributes of non-terminal B
are the output parameters of procedure B.



All the semantic rules can be evaluated
at the end of parsing of production rules



L-ATTRIBUTED DEFINITIONS

- S-attributed definitions only allow synthesized attributes.
- We saw earlier that inherited attributes are useful.
- But we prefer definitions that can be evaluated in one pass.
- L-ATTRIBUTED definitions are the set of SDDs whose attributes can be evaluated in a DEPTH-FIRST traversal of the parse tree.

L-ATTRIBUTED DEFINITIONS

- A syntax-directed definition is **L-attributed** if each inherited attribute of X_j , where $1 \leq j \leq n$, on the right side of $A \rightarrow X_1 X_2 \dots X_n$ depends only on:
 1. The attributes of the symbols X_1, \dots, X_{j-1} to the left of X_j in the production and
 2. the inherited attribute of A
- Every S-attributed definition is L-attributed, the restrictions only apply to the inherited attributes (not to synthesized attributes).

L-ATTRIBUTED GRAMMAR

- Informally – dependency-graph edges may go from left to right, not other way around.
- Given production $A \rightarrow X_1 X_2 \cdots X_n$, inherited attributes of X_j depend only on:
 - inherited attributes of A
 - arbitrary attributes of $X_1, X_2, \cdots X_{j-1}$
- Synthesized attributes of A depend only on its inherited attributes and arbitrary RHS attributes
- Synthesized attributes of an action depends only on its inherited attributes
 - i.e., evaluation order: $\text{Inh}(A), \text{Inh}(X_1), \text{Syn}(X_1), \dots, \text{Inh}(X_n), \text{Syn}(X_n), \text{Syn}(A)$
 - This is precisely the order of evaluation for an LL parser

L-ATTRIBUTED DEFINITIONS

Is the following SDD L-attributed?

Production

$A \rightarrow LM$

Semantic Rules

$L.i := l(A.i)$

$M.i := m(L.s)$

$A.s := f(M.s)$

$A \rightarrow QR$

$R.i := r(A.i)$

$Q.i := q(R.s)$

$A.s := f(Q.s)$

A DEFINITION WHICH IS NOT L-ATTRIBUTED

Productions

$A \rightarrow L M$

$A \rightarrow Q R$

Semantic Rules

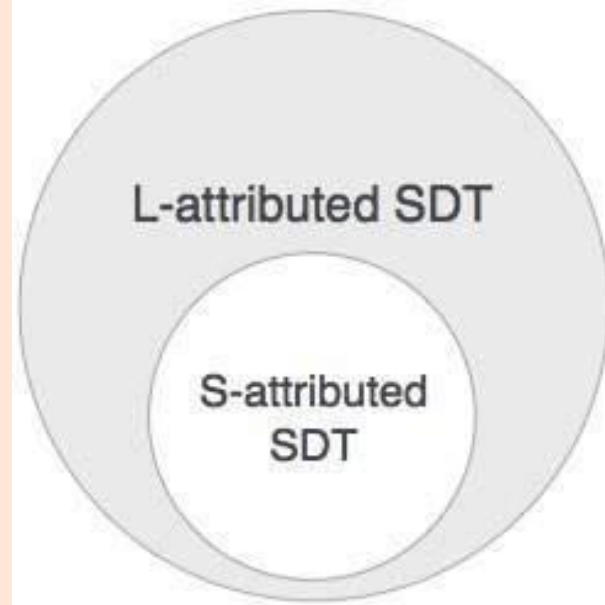
$L.in = l(A.i), M.in = m(L.s), A.s = f(M.s)$

$R.in = r(A.in), Q.in = q(R.s), A.s = f(Q.s)$

- This syntax-directed definition is not L-attributed because the semantic rule $Q.in = q(R.s)$ violates the restrictions of L-attributed definitions.
- When $Q.in$ must be evaluated before we enter to Q because it is an inherited attribute.
- But the value of $Q.in$ depends on $R.s$ which will be available after we return from R . So, we are not be able to evaluate the value of $Q.in$ before we enter to Q .

COMPARISON BETWEEN L-ATTRIBUTED AND S-ATTRIBUTED SDT

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

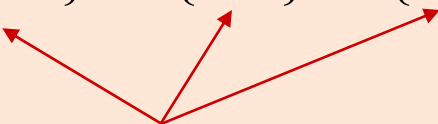


3.4 TRANSLATION SCHEMES

TRANSLATION SCHEMES

- In a syntax-directed definition, we do not say anything about the evaluation times of the semantic rules (when the semantic rules associated with a production should be evaluated?).
- 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 productions.

○ *Ex:* $A \rightarrow \{ \dots \} X \{ \dots \} Y \{ \dots \}$



Semantic Actions

Translation schemes are closer to a real implementation because they specify when, during the parse, attributes should be computed.

TRANSLATION SCHEMES

- When designing a translation scheme, some restrictions should be observed to ensure that an attribute value is available when a semantic action refers to that attribute.
- These restrictions (motivated by L-attributed definitions) ensure that a semantic action does not refer to an attribute that has not yet computed.
- In translation schemes, we use *semantic action* terminology instead of *semantic rule* terminology used in syntax-directed definitions.
- The position of the semantic action on the right side indicates when that semantic action will be evaluated.

TURNING A SDD INTO A TRANSLATION SCHEME

- For a translation scheme to work, it must be the case that an attribute is computed BEFORE it is used.
- If the SDD is S-attributed, it is easy to create the translation scheme implementing it:

Production

$T \rightarrow T1 * F$

Semantic Rule

$T.val := T1.val \times F.val$

- Translation scheme:

$T \rightarrow T1 * F \{ T.val = T1.val * F.val \}$

- Each associated semantic rule in a S-attributed syntax-directed definition will be inserted as a semantic action into the end of the right side of the associated production.
- That is, we just turn the semantic rule into an action and add at the far right hand side. This DOES NOT WORK for inherited attributes!

A TRANSLATION SCHEME EXAMPLE

- A simple translation scheme that converts infix expressions to the corresponding postfix expressions.

$E \rightarrow T R$

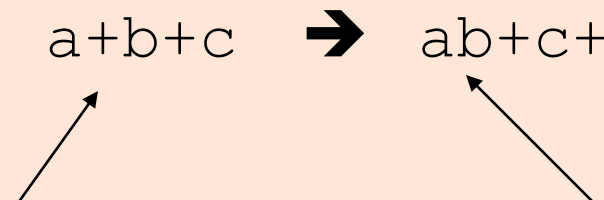
$R \rightarrow + T \{ \text{print}(\text{"+"}) \} R_1$

$R \rightarrow \varepsilon$

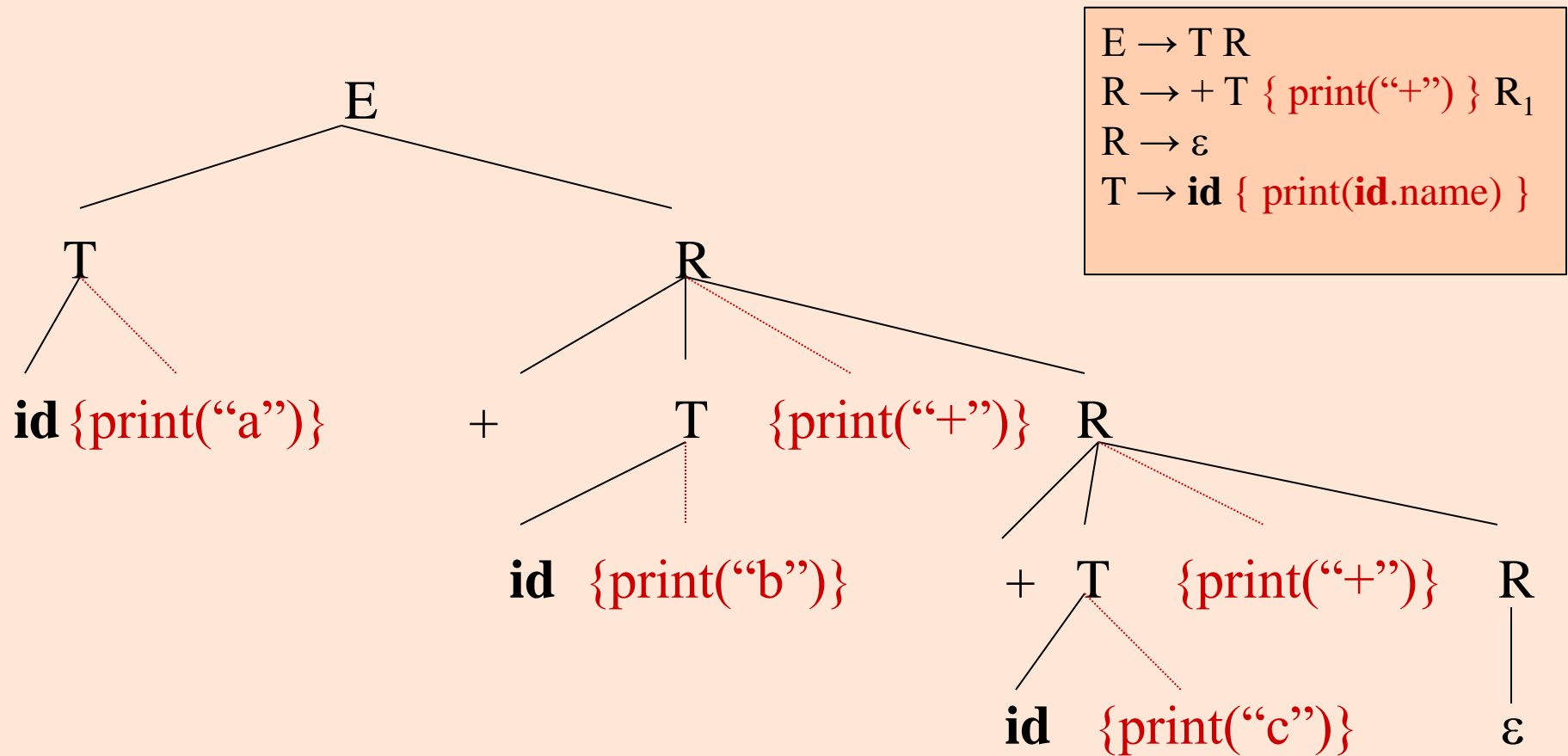
$T \rightarrow \text{id} \{ \text{print}(\text{id.name}) \}$

$a+b+c \rightarrow ab+c+$

infix expression postfix expression



A TRANSLATION SCHEME EXAMPLE (CONT.)



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

COMPARISON BETWEEN SDD AND SDT

- SDD: Specifies the values of attributes by associating semantic rules with the productions.
- SDD is easier to read; easy for specification.
- SDT scheme: embeds program fragments (also called semantic actions) within production bodies. The position of the action defines the order in which the action is executed (in the middle of production or end).
- SDT scheme: can be more efficient; easy for implementation.

EXAMPLE OF SDD AND SDT

productions	SDD	SDTS
expr \rightarrow list + term	expr.t = list.t term.t "+"	expr \rightarrow list + term printf{"+"})}
expr \rightarrow list - term	expr.t = list.t term.t "-"	expr \rightarrow list + term printf{"-"})}
expr \rightarrow term	expr.t = term.t	expr \rightarrow term
term \rightarrow 0	term.t = "0"	term \rightarrow 0 printf{"0"})}
term \rightarrow 1	term.t = "1"	term \rightarrow 1 printf{"1"})}
...
term \rightarrow 9	term.t = "9"	term \rightarrow 9 printf{"0"})}

SDD and SDT for Infix to Postfix Notation



INHERITED ATTRIBUTES IN TRANSLATION SCHEMES

With inherited attributes, the translation scheme designer needs to follow three rules:

1. An inherited attribute for a symbol on the RHS **MUST** be computed in an action **BEFORE** the occurrence of the symbol.
2. An action **MUST NOT** refer to the synthesized attribute of a symbol to the right of the action.
3. A synthesized attribute for the LHS nonterminal can **ONLY** be computed in an action **FOLLOWING** the symbols for all the attributes it references.

With a 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).

EXAMPLE

This translation scheme does NOT follow the rules:

$$\begin{array}{ll} S \rightarrow A_1 A_2 & \{ A_1.in = 1; A_2.in = 2 \} \\ A \rightarrow a & \{ \text{print}(A.in) \} \end{array}$$

If we traverse the parse tree depth first, $A_1.in$ has not been set when referred to in the action $\text{print}(A.in)$

$$\begin{array}{l} S \rightarrow \{ A_1.in = 1 \} A_1 \{ A_2.in = 2 \} A_2 \\ A \rightarrow a \{ \text{print}(A.in) \} \end{array}$$

TOP-DOWN TRANSLATION

- We will look at the implementation of L-attributed definitions during predictive parsing.
- Instead of the syntax-directed translations, we will work with translation schemes.
- We will see how to evaluate inherited attributes (in L-attributed definitions) during recursive predictive parsing.
- We will also look at what happens to attributes during the left-recursion elimination in the left-recursive grammars.

TOP-DOWN TRANSLATION

- For each non-terminal A , construct a function that
 - Has a formal parameter for each inherited attribute of A
 - Returns the values of the synthesized attributes of A
- The code associated with each production does the following:
 - Save the s-attribute of each token X into a variable $X.x$
 - Generate an assignment $B.s = \text{parse } B(B.i_1, B.i_2, \dots, B.i_k)$ for each non-terminal B , where $B.i_1, \dots, B.i_k$ are values for the L attributes of B and $B.s$ is a variable to store s-attributes of B .
 - Copy the code for each action, replacing references to attributes by the corresponding variables

A TRANSLATION SCHEME WITH INHERITED ATTRIBUTES

$D \rightarrow T \text{ id } \{ \text{addtype}(\text{id.entry}, T.\text{type}), L.\text{in} = T.\text{type} \} L$

$T \rightarrow \text{int } \{ T.\text{type} = \text{integer} \}$

$T \rightarrow \text{real } \{ T.\text{type} = \text{real} \}$

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

$L \rightarrow \varepsilon$

- This is a translation scheme for an L-attributed definitions.

PREDICTIVE PARSING (OF INHERITED ATTRIBUTES)

```
procedure D() {  
    int Ttype,Lin,identry;  
    call T(&Ttype); consume(id,&identry);  
    addtype(identry,Ttype); Lin=Ttype;  
    call L(Lin);  
}  
procedure T(int *Ttype) {  
    if (currtoken is int) { consume(int); *Ttype=TYPEINT; }  
    else if (currtoken is real) { consume(real); *Ttype=TYPEREAL; }  
    else { error("unexpected type"); }  
}  
procedure L(int Lin) {  
    if (currtoken is id) { int L1in,identry; consume(id,&identry);  
                           addtype(identry,Lin); L1in=Lin; call L(L1in); }  
    else if (currtoken is endmarker) { }  
    else { error("unexpected token"); }  
}
```

a synthesized attribute (an output parameter)

an inherited attribute (an input parameter)

ELIMINATING LEFT RECURSION FROM TRANSLATION SCHEME

- A translation scheme with a left recursive grammar.

$$E \rightarrow E_1 + T \{ E.val = E_1.val + T.val \}$$
$$E \rightarrow E_1 - T \{ E.val = E_1.val - T.val \}$$
$$E \rightarrow T \{ E.val = T.val \}$$
$$T \rightarrow T_1 * F \{ T.val = T_1.val * F.val \}$$
$$T \rightarrow F \{ T.val = F.val \}$$
$$F \rightarrow (E) \{ F.val = E.val \}$$
$$F \rightarrow \mathbf{digit} \{ F.val = \mathbf{digit.lexval} \}$$

- 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 (IN GENERAL)

$$\begin{array}{l} A \rightarrow A \alpha \mid \beta \\ \Downarrow \\ A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \varepsilon \end{array}$$

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

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

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

\Downarrow eliminate left recursion

inherited attribute of the new non-terminal

synthesized attribute of the new non-terminal

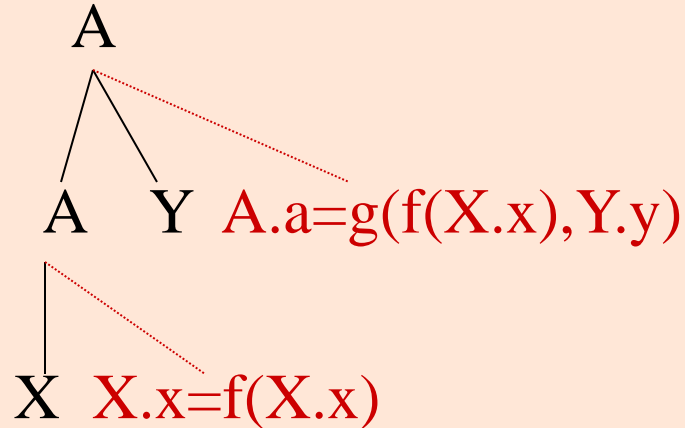
$$A \rightarrow X \{ R.in = f(X.x) \} R \{ A.a = R.syn \}$$

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

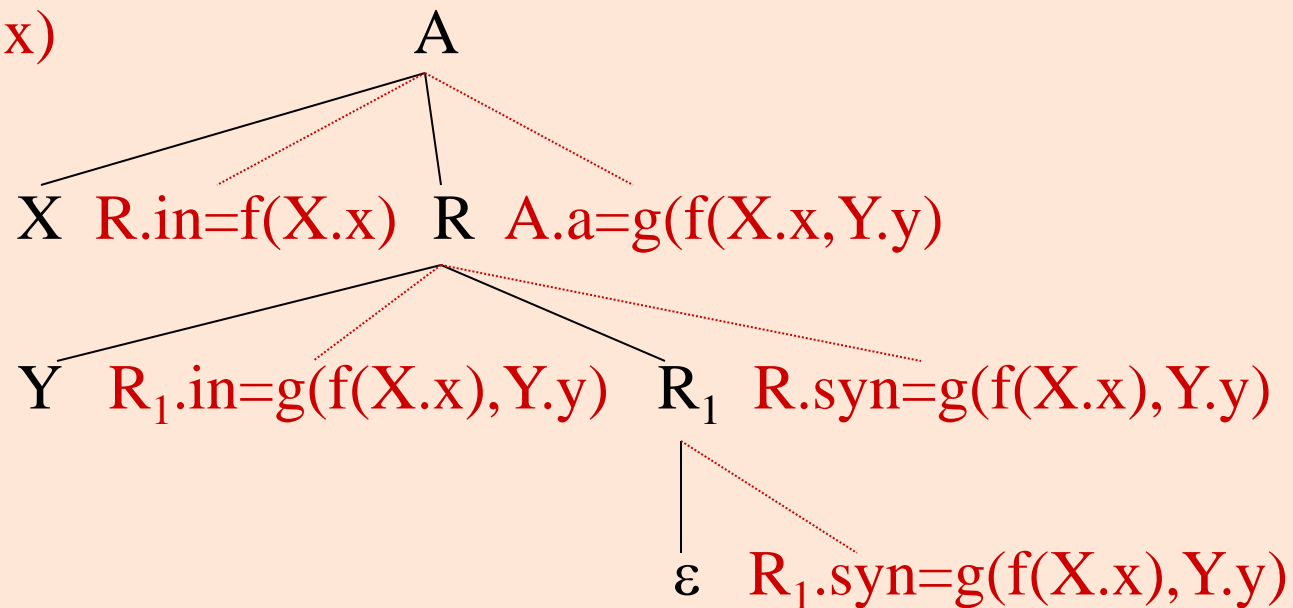
$$R \rightarrow \varepsilon \{ R.syn = R.in \}$$

EVALUATING ATTRIBUTES

Parse tree of left recursive grammar



Parse tree of non-left-recursive grammar



ELIMINATING LEFT RECURSION (CONT.)

inherited attribute

synthesized attribute

$E \rightarrow T \{ A.in = T.val \} A \{ E.val = A.syn \}$

$A \rightarrow + T \{ A_1.in = A.in + T.val \} A_1 \{ A.syn = A_1.syn \}$

$A \rightarrow - T \{ A_1.in = A.in - T.val \} A_1 \{ A.syn = A_1.syn \}$

$A \rightarrow \epsilon \{ A.syn = A.in \}$

$T \rightarrow F \{ B.in = F.val \} B \{ T.val = B.syn \}$

$B \rightarrow * F \{ B_1.in = B.in * F.val \} B_1 \{ B.syn = B_1.syn \}$

$B \rightarrow \epsilon \{ B.syn = B.in \}$

$F \rightarrow (E) \{ F.val = E.val \}$

$F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

TRANSLATION SCHEME - INTERMEDIATE CODE GENERATION

$E \rightarrow T \{ A.in = T.loc \} A \{ E.loc = A.loc \}$

$A \rightarrow + T \{ A_1.in = \text{newtemp}(); \text{emit}(\text{add}, A.in, T.loc, A_1.in) \}$
 $A_1 \{ A.loc = A_1.loc \}$

$A \rightarrow \varepsilon \{ A.loc = A.in \}$

$T \rightarrow F \{ B.in = F.loc \} B \{ T.loc = B.loc \}$

$B \rightarrow * F \{ B_1.in = \text{newtemp}(); \text{emit}(\text{mult}, B.in, F.loc, B_1.in) \}$
 $B_1 \{ B.loc = B_1.loc \}$

$B \rightarrow \varepsilon \{ B.loc = B.in \}$

$F \rightarrow (E) \{ F.loc = E.loc \}$

$F \rightarrow \mathbf{id} \{ F.loc = \mathbf{id.name} \}$

PREDICTIVE PARSING – INTERMEDIATE CODE GENERATION

```
procedure E(char **Eloc) {  
    char *Ain, *Tloc, *Aloc;  
    call T(&Tloc); Ain=Tloc;  
    call A(Ain,&Aloc); *Eloc=Aloc;  
}  
procedure A(char *Ain, char **Aloc) {  
    if (currtok is +) {  
        char *A1in, *Tloc, *A1loc;  
        consume(+); call T(&Tloc); A1in=newtemp(); emit("add",Ain,Tloc,A1in);  
        call A(A1in,&A1loc); *Aloc=A1loc;  
    }  
    else { *Aloc = Ain }  
}
```


PREDICTIVE PARSING (CONT.)

```
procedure T(char **Tloc) {  
    char *Bin, *Floc, *Bloc;  
    call F(&Floc); Bin=Floc;  
    call B(Bin,&Bloc); *Tloc=Bloc;  
}  
procedure B(char *Bin, char **Bloc) {  
    if (currtok is *) {  
        char *B1in, *Floc, *B1loc;  
        consume(+); call F(&Floc); B1in=newtemp(); emit("mult",Bin,Floc,B1in);  
        call B(B1in,&B1loc); Bloc=B1loc;  
    }  
    else { *Bloc = Bin }  
}  
procedure F(char **Floc) {  
    if (currtok is "(") { char *Eloc; consume("("); call E(&Eloc); consume(")"); *Floc=Eloc }  
    else { char *idname; consume(id,&idname); *Floc=idname }  
}
```

BOTTOM-UP EVALUATION OF INHERITED ATTRIBUTES

- The first step is to convert the SDD to a valid translation scheme.
- Then a few “tricks” have to be applied to the translation scheme.
- It is possible, with the right tricks, to do one-pass bottom-up attribute evaluation for ALL LL(1) grammars and MOST LR(1) grammars, if the SDD is L-attributed.

REMOVING EMBEDDING SEMANTIC ACTIONS

- In bottom-up evaluation scheme, the semantic actions are evaluated during the reductions.
- During the bottom-up evaluation of S-attributed definitions, we have a parallel stack to hold synthesized attributes.
- *Problem:* where are we going to hold inherited attributes?
- *A Solution:*

We will convert our grammar to an equivalent grammar to guarantee the followings:

- All embedding semantic actions in our translation scheme will be moved into the end of the production rules.
- All inherited attributes will be copied into the synthesized attributes (most of the time synthesized attributes of new non-terminals).
- Thus, we will evaluate all semantic actions during reductions, and we find a place to store an inherited attribute.

REMOVING EMBEDDING SEMANTIC ACTIONS

To transform our translation scheme into an equivalent translation scheme:

1. Remove an embedding semantic action S_i , put a new non-terminal M_i instead of that semantic action.
2. Put that semantic action S_i into the end of a new production rule $M_i \rightarrow \varepsilon$ for that non-terminal M_i .
3. That semantic action S_i will be evaluated when this new production rule is reduced.
4. The evaluation order of the semantic rules are not changed by this transformation.

REMOVING EMBEDDING SEMANTIC ACTIONS

$$A \rightarrow \{S_1\} X_1 \{S_2\} X_2 \dots \{S_n\} X_n$$

\Downarrow remove embedding semantic actions

$$A \rightarrow M_1 X_1 M_2 X_2 \dots M_n X_n$$

$$M_1 \rightarrow_{\varepsilon} \{S_1\}$$

$$M_2 \rightarrow_{\varepsilon} \{S_2\}$$

.

.

$$M_n \rightarrow_{\varepsilon} \{S_n\}$$

REMOVING EMBEDDING SEMANTIC ACTIONS

$E \rightarrow T R$

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

$R \rightarrow \varepsilon$

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

\Downarrow remove embedding semantic actions

$E \rightarrow T R$

$R \rightarrow + T M R_1$

$R \rightarrow \varepsilon$

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

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

TRANSLATION WITH INHERITED ATTRIBUTES

- Let us assume that every non-terminal A has an inherited attribute $A.i$, and every symbol X has a synthesized attribute $X.s$ in our grammar.
- For every production rule $A \rightarrow X_1 X_2 \dots X_n$,
 - introduce new marker non-terminals M_1, M_2, \dots, M_n and
 - replace this production rule with $A \rightarrow M_1 X_1 M_2 X_2 \dots M_n X_n$
 - the synthesized attribute of X_i will not be changed.
 - the inherited attribute of X_i will be copied into the synthesized attribute of M_i by the new semantic action added at the end of the new production rule $M_i \rightarrow \epsilon$.
 - Now, the inherited attribute of X_i can be found in the synthesized attribute of M_i (which is immediately available in the stack).

$$A \rightarrow \{B.i=f_1(\dots)\} B \{C.i=f_2(\dots)\} C \{A.s=f_3(\dots)\}$$
$$\Downarrow$$
$$A \rightarrow \{M_1.i=f_1(\dots)\} M_1 \{B.i=M_1.s\} B \{M_2.i=f_2(\dots)\} M_2 \{C.i=M_2.s\} C \{A.s=f_3(\dots)\}$$
$$M_1 \rightarrow \epsilon \{M_1.s=M_1.i\}$$
$$M_2 \rightarrow \epsilon \{M_2.s=M_2.i\}$$

TRANSLATION WITH INHERITED ATTRIBUTES

$$\begin{array}{lll}
 S \rightarrow \{A.i=1\} & A & \{S.s=k(A.i,A.s)\} \\
 A \rightarrow \{B.i=f(A.i)\} & B & \{C.i=g(A.i,B.i,B.s)\} \quad C \{A.s= h(A.i,B.i,B.s,C.i,C.s)\} \\
 B \rightarrow b & \{B.s=m(B.i,b.s)\} & \\
 C \rightarrow c & \{C.s=n(C.i,c.s)\} &
 \end{array}$$


$$\begin{array}{llll}
 S \rightarrow \{M_1.i=1\} & M_1 & \{A.i=M_1.s\} & A \quad \{S.s=k(M_1.s,A.s)\} \\
 A \rightarrow \{M_2.i=f(A.i)\} & & M_2 & \{B.i=M_2.s\} \quad B \\
 & \{M_3.i=g(A.i,M_2.s,B.s)\} & M_3 & \{C.i=M_3.s\} \quad C \\
 & \{A.s= h(A.i, M_2.s,B.s, M_3.s,C.s)\} & & \\
 B \rightarrow b & \{B.s=m(B.i,b.s)\} & & \\
 C \rightarrow c & \{C.s=n(C.i,c.s)\} & & \\
 M_1 \rightarrow \epsilon & \{M_1.s=M_1.i\} & & \\
 M_2 \rightarrow \epsilon & \{M_2.s=M_2.i\} & & \\
 M_3 \rightarrow \epsilon & \{M_3.s=M_3.i\} & &
 \end{array}$$

ACTUAL TRANSLATION SCHEME

$$S \rightarrow \{M_1.i=1\} M_1 \{A.i=M_1.s\} A \{S.s=k(M_1.s,A.s)\}$$

$$A \rightarrow \{M_2.i=f(A.i)\} M_2 \{B.i=M_2.s\} B \{M_3.i=g(A.i,M_2.s,B.s)\} M_3 \{C.i=M_3.s\} C \{A.s= h(A.i, M_2.s,B.s, M_3.s,C.s)\}$$

$$B \rightarrow b \{B.s=m(B.i,b.s)\}$$

$$C \rightarrow c \{C.s=n(C.i,c.s)\}$$

$$M_1 \rightarrow \varepsilon \{M_1.s= M_1.i\}$$

$$M_2 \rightarrow \varepsilon \{M_2.s=M_2.i\}$$

$$M_3 \rightarrow \varepsilon \{M_3.s=M_3.i\}$$

$$S \rightarrow M_1 A \quad \{ s[ntop]=k(s[top-1],s[top]) \}$$

$$M_1 \rightarrow \varepsilon \quad \{ s[ntop]=1 \}$$

$$A \rightarrow M_2 B M_3 C \quad \{ s[ntop]=h(s[top-4],s[top-3],s[top-2],s[top-1],s[top]) \}$$

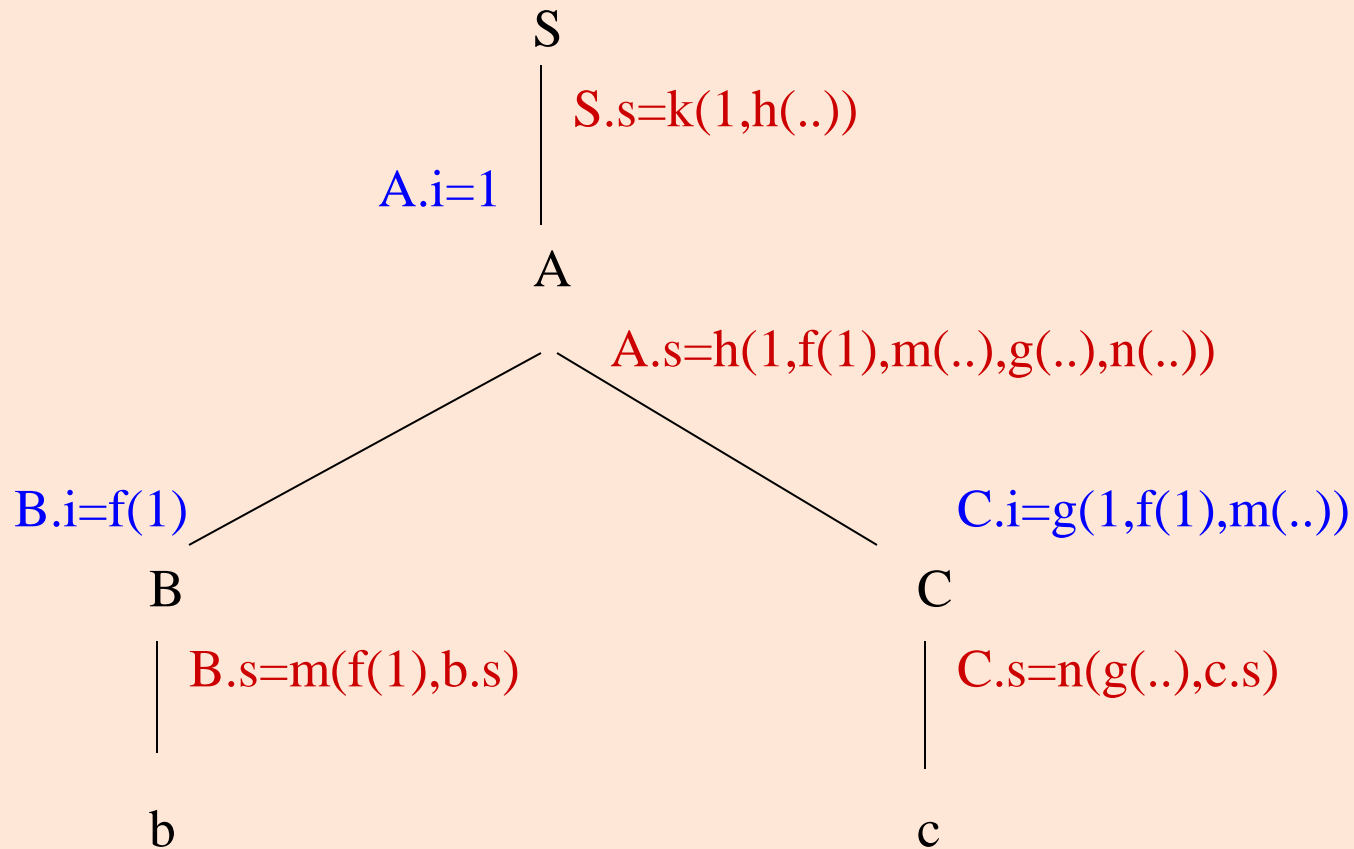
$$M_2 \rightarrow \varepsilon \quad \{ s[ntop]=f(s[top]) \}$$

$$M_3 \rightarrow \varepsilon \quad \{ s[ntop]=g(s[top-2],s[top-1],s[top]) \}$$

$$B \rightarrow b \quad \{ s[ntop]=m(s[top-1],s[top]) \}$$

$$C \rightarrow c \quad \{ s[ntop]=n(s[top-1],s[top]) \}$$

EVALUATION OF ATTRIBUTES



EVALUATION OF ATTRIBUTES

<u>stack</u>	<u>input</u>	<u>s-attribute stack</u>
	bc\$	
M ₁	bc\$	1
M ₁ M ₂	bc\$	1 f(1)
M ₁ M ₂ b	c\$	1 f(1) b.s
M ₁ M ₂ B	c\$	1 f(1) m(f(1),b.s)
M ₁ M ₂ B M ₃	c\$	1 f(1) m(f(1),b.s) g(1,f(1),m(f(1),b.s))
M ₁ M ₂ B M ₃ c	\$	1 f(1) m(f(1),b.s) g(1,f(1),m(f(1),b.s)) c.s
M ₁ M ₂ B M ₃ C	\$	1 f(1) m(f(1),b.s) g(1,f(1),m(f(1),b.s)) n(g(..),c.s)
M ₁ A	\$	1 h(f(1),m(..),g(..),n(..))
S	\$	k(1,h(..))