Syntax-Directed Translation

Basic Idea

- Guided by context-free grammar
- Associate information with a programming language construction by attaching attributes to the grammar symbols representing the construction
- Values for attributes are computed by “semantic rules ”associated with the grammar productions

Two Notations for Associating Semantic Rules with Productions

- Syntax-directed definitions
  - High-level specifications for translations
  - Hide implementation details
  - No need to specify explicitly the order in which translation take place

Two Notations for Associating Semantic Rules with Productions

- Translation schemes
  - Indicate the order in which semantic rules are to be evaluated
  - Allow some implementation details to be shown

Conceptual View of Syntax-Directed Translation

- Notes:
  - Evaluation of the semantic rules may generate code, save information in a symbol table, issue error messages, or perform any other activities
  - Special cases of syntax-directed definitions can be implemented in a single pass by evaluating semantic rules during parsing, without explicitly constructing a parse tree or a graph showing dependencies between attributes

Syntax-Directed Definitions

- Syntax-directed definition
  - A generalization of a context-free grammar in which each grammar symbol has an associated set of attributes
- Attribute
  - Represent anything we choose: a string, a number, a type, a memory location, etc.
- The value of an attribute at a parse-tree node is defined by a semantic rule associated with the production used at that node

Synthesized and Inherited Attributes

- In a syntax-directed definition, each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form $b := f(c_1, c_2, \ldots c_k)$, where f is a function, and either
  - b is a synthesized attribute of A, and $c_1, c_2, \ldots c_k$ are attributes belonging to the grammar symbols of the production, or
  - b is an inherited attribute of one of the grammar symbols on the right side of the production, and $c_1, c_2, \ldots c_k$ are attributes belonging to the grammar symbols of the production

Synthesized and Inherited Attributes

- Synthesized attribute
  - The value of a synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree
- Inherited attribute
  - The value of an inherited attribute is computed from the values of attributes at the siblings and parent of that node

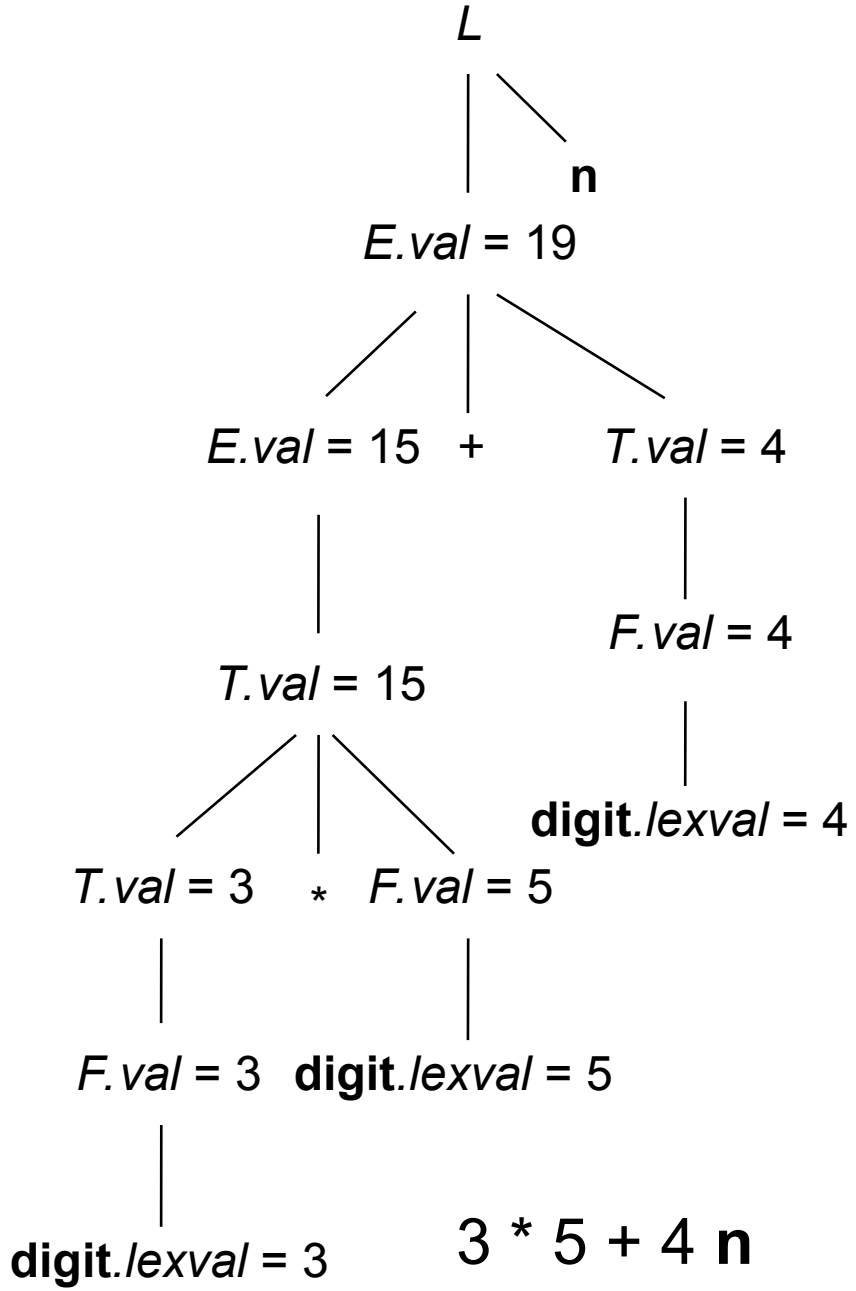
Annotated Parse Tree

- A parse tree showing the values of attributes at each node
- The process of computing the attribute values at the nodes is called annotating or decorating the parse tree
- Values for attributes of terminals are usually supplied by the lexical analyzer

S-Attributed Definition

A syntax-directed definition that uses synthesized attributes exclusively

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

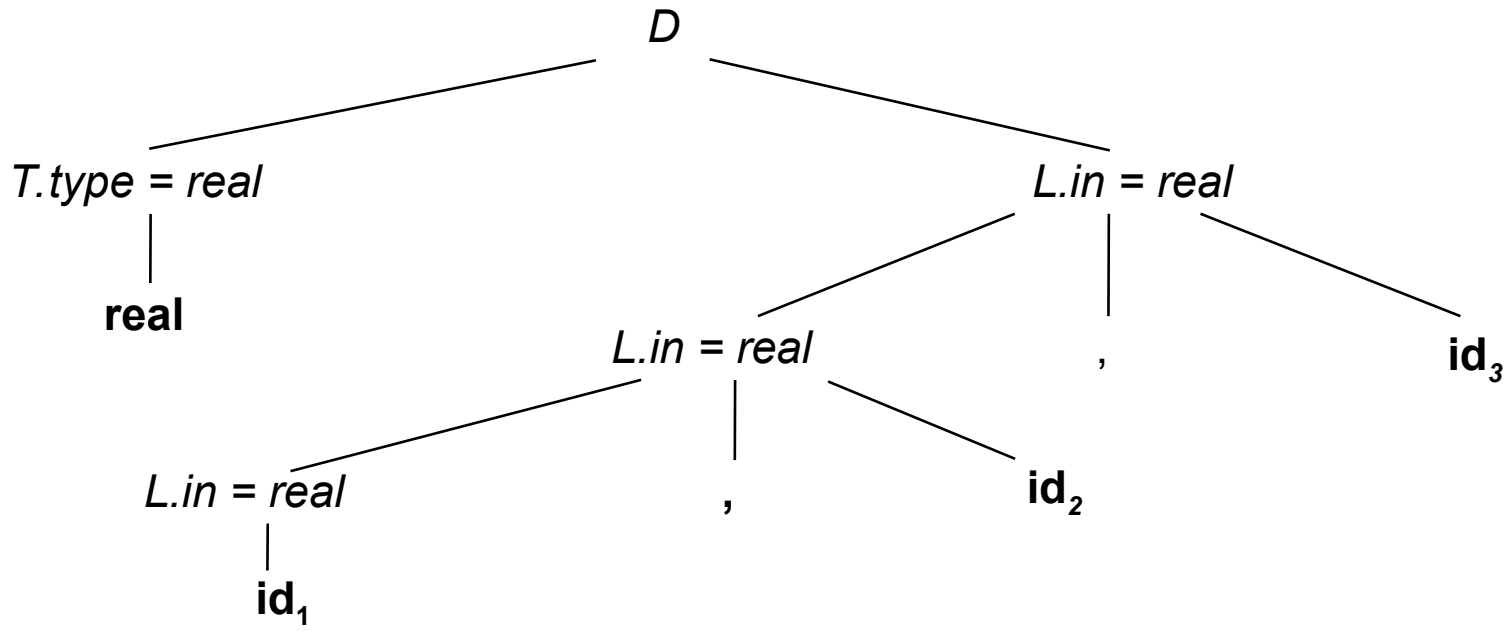


Inherited Attributes

- Expressing the dependence of a programming language construction on the context in which it appears
  - By keeping track of the context whether an identifier appears on the left or right side of an assignment context we can decide whether the address or the value of the identifier is needed
  - Although it is always possible to rewrite a syntax-directed definition to use only synthesized attributes, it is often natural to use syntax-directed definition with inherited attributes

Example of Non-S-Attributed Definition

Production	Semantic Rules
D→ *T L*	*L.in := T.type*
T→ **int**	*T.type := integer*
T→ **real**	*T.type := real*
L→ L_1 , **id**	L_1.in := L.in *addtype*(**id***.entry, L.in*)
L→ id	*addtype*(**id***.entry, L.in*)



Attribute Grammar

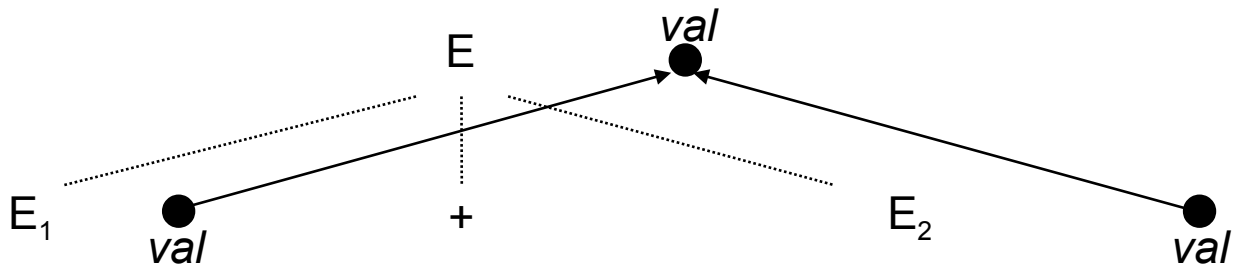
- A syntax-directed definition in which the functions in semantic rules cannot have side effects

Dependency Graph

- A directed graph that represents dependencies between attributes set up by the semantic rules
- Notes:
  - From the dependency graph, we can derive an evaluation order for the semantic rules
  - Evaluation of the semantic rules defines the values of the attributes at the nodes in the parse tree
  - A semantic rule may have side effects, e,g, printing a value or updating a global variable
  - For procedure calls with side effects we introduce a dummy synthesized attribute

Dependency Graph

```
for each node n in the parse tree do
        for each attribute a of the grammar symbol at n do
                construct a node in the dependency graph for a;
for each node n in the parse tree do
        for each semantic rule b:=f(c1,c2,...ck) associated with the production used at n do
                for i := 1 to k do
                        construct and edge from the node for ci to the node for bi
```

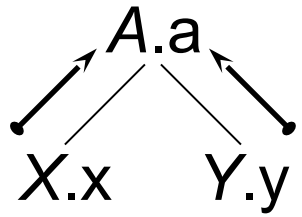


$E \rightarrow E_1 + E_2$ $\quad E.val := E_1.val + E_2.val$

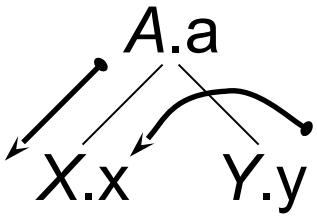
- The three nodes of the dependency graph marked by • represent the synthesized attributes E.val, E_1.val, and E_2.val at the corresponding nodes in the parse tree
- The dotted lines represent the parse tree and are not part of the dependency graph

Acyclic Dependency Graphs for Parse Trees

$A \rightarrow X\ Y$



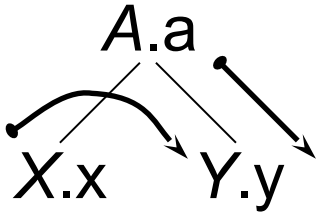
$A.a := f(X.x,\ Y.y)$



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

Direction of

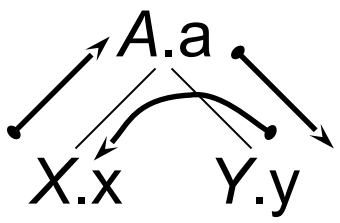
value dependence



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

Dependency Graphs with Cycles?

- Edges in the dependence graph show the evaluation order for attribute values
- Dependency graphs cannot be cyclic



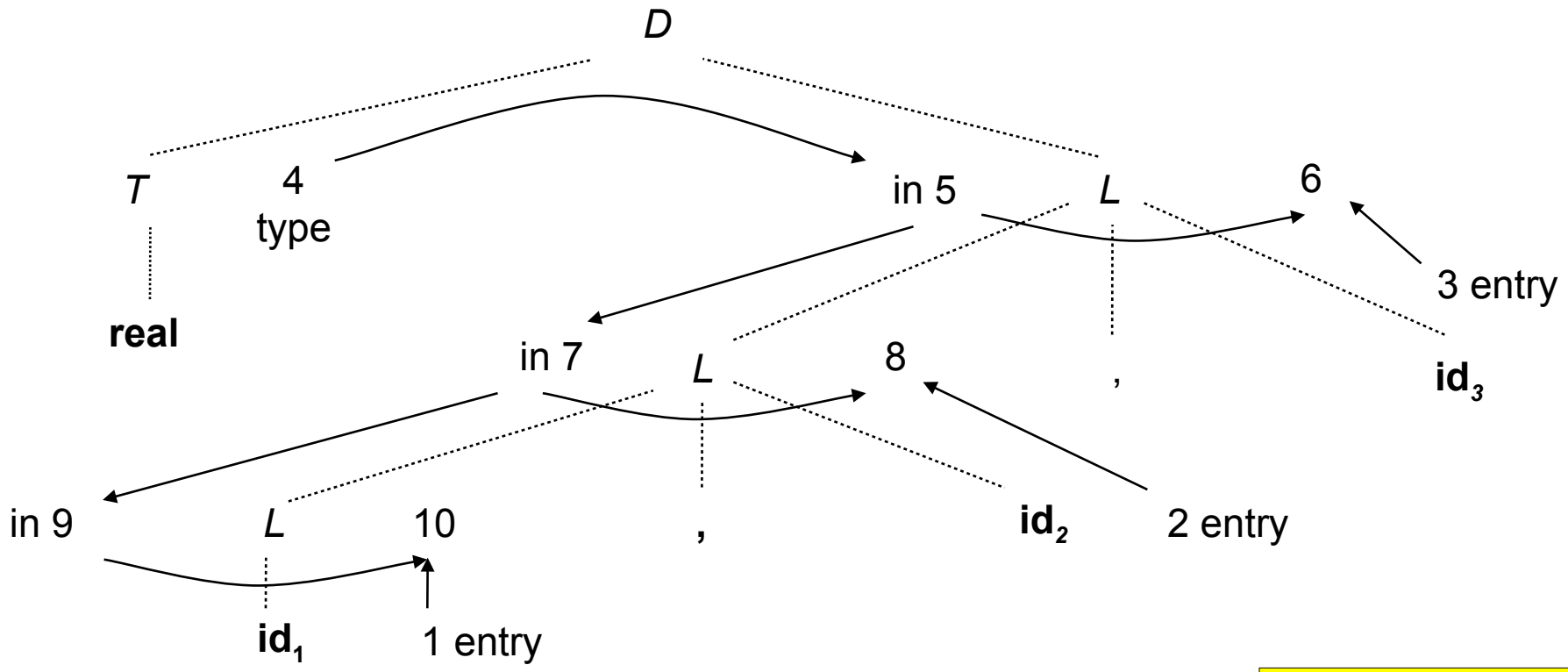
$A.a := f(X.x)$
$X.x := f(Y.y)$
$Y.y := f(A.a)$

Error: cyclic dependence

Evaluation Order

- Basic idea in the topological sort of a graph
- Topological sort of a graph
  - Any ordering $m_1, m_2, \ldots, m_k$ of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes
  - From a topological sort of a dependency graph, we obtain an evaluation order for the semantic rules

Dependency Graph



Production	Semantic Rules
$D \rightarrow T\ L$	*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*)

a_4:=*real;*
a_5:=a_4;
addtype(**id**$_3$.*entry,* a_5);
a_7:=a_5;
addtype(**id**$_2$.*entry,* a_7);
a_9:=a_7;
addtype(**id**$_1$.*entry,* a_9);

Evaluation Order

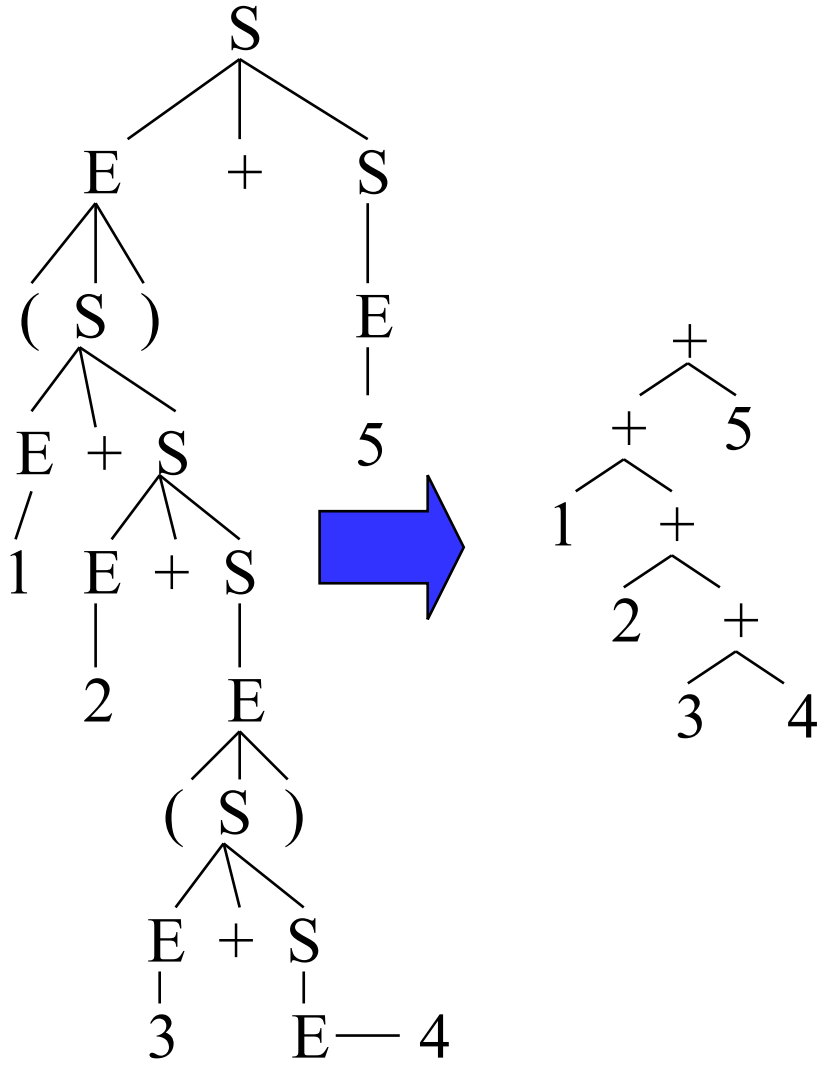
- Parse-tree methods
  - Obtain an evaluation order from a topological sort of the dependency graph constructed from the parse tree for each input
    - It will fail if the dependency graph for the particular parse tree has a cycle
- Rule-based methods
  - At compiler-construction time, the semantic rules associated with productions are analyzed
    - For each production, the order is predetermined
- Oblivious methods
  - An evaluation order is chosen without considering the semantic rules
    - It restricts the class of syntax-directed definitions that can be implemented

Evaluation Order

- Rule-based and oblivious methods need not explicitly construct the dependency graph at compile time, so they can be more efficient in their use of compiler time and space

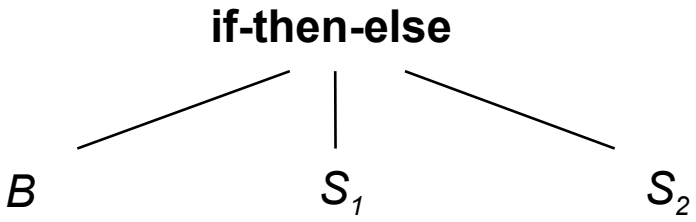
Abstract Syntax Tree (AST)

- Derivation = sequence of applied productions
- S => E+S => 1+S => 1+E => 1+2
- Parse tree = graph representation of a derivation
- Doesn't capture the order of applying the productions
- AST discards unnecessary information from the parse tree
- In an AST, operators and keywords do not appear as leaves, but rather are associated with the interior node that would be parent of those leaves in the parse tree

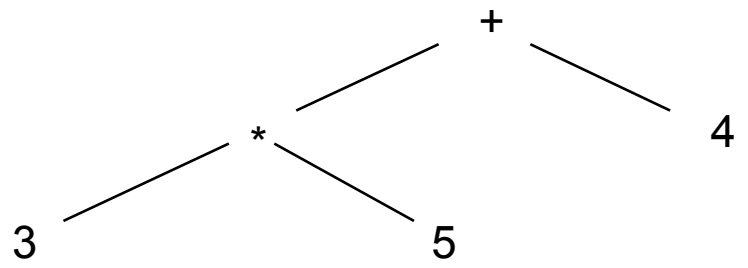


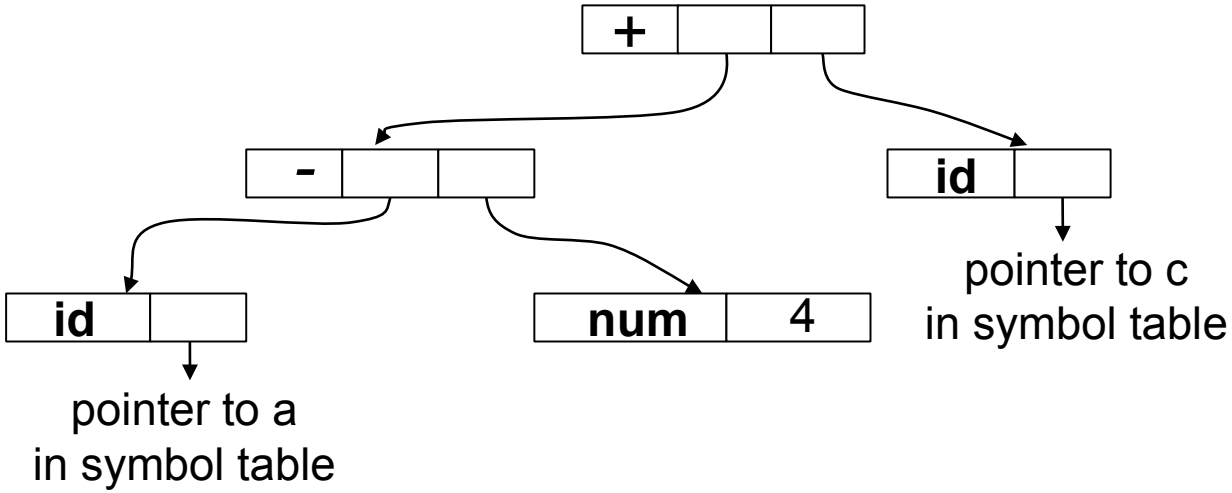
Contruction of Syntax Trees

$S \rightarrow$ **if** B **then** S_1 **else** S_2



3 * 5 + 4



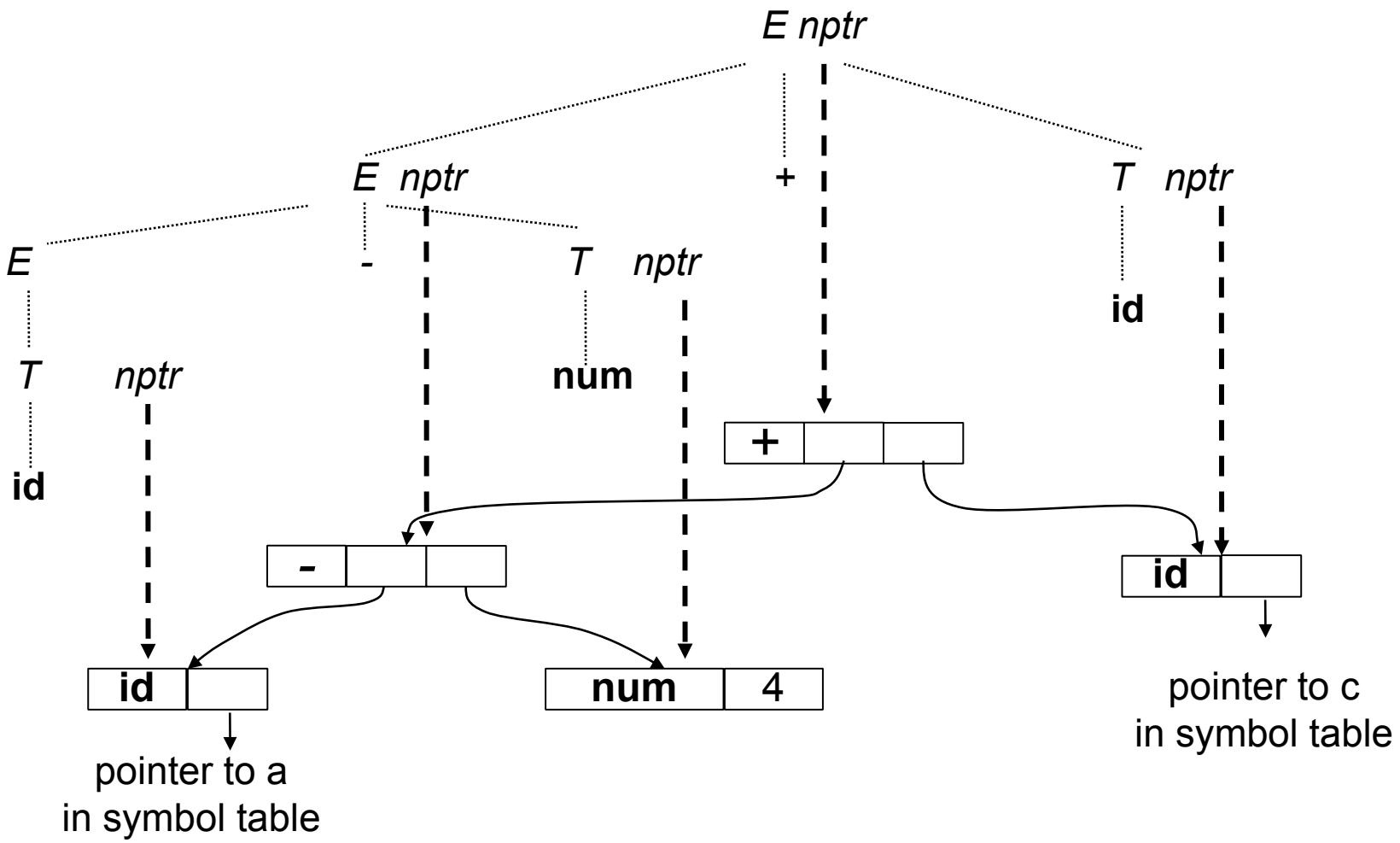


Construction of Syntax Trees using Syntax-Directed Definitions

mknode (*op, left, right*)
mkleaf (**id**,*entry*)
mkleaf(**num**, *val*)

Production	Semantic Rules
$E \rightarrow E_1 + T$	*E.nptr* := *mknode* ('+', E_1*.nptr, T.nptr*)
$E \rightarrow E_1 - T$	*E.nptr* := *mknode* ('-', E_1*.nptr, T.nptr*)
$E \rightarrow T$	*E.nptr := T.nptr*
$T \rightarrow (E)$	*T.nptr* := *E.nptr*
$T \rightarrow$ **id**	*T.nptr* := *mkleaf (***id**, **id***.entry)*
$T \rightarrow$ **num**	*T.nptr* := *mkleaf (***num**, **num***.val)*

Construction of Syntax Trees using Syntax-Directed Definitions



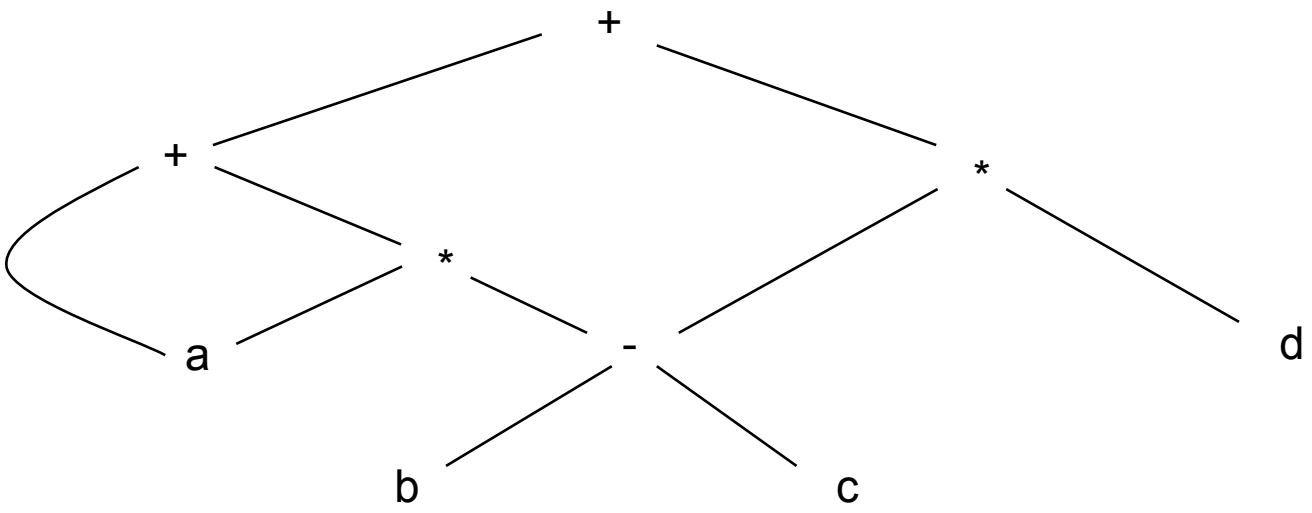
a - 4 + c

p_1:=*mkleaf* (**id**, *entrya*);
p_2:=*mkleaf* (**num**, 4);
p_3:=*mknode* ('-', *p1, p2*);
p_4:=*mkleaf* (**id**, *entryc*);
p_5:=*mknode* ('+', *p3, p4*);

Directed Acyclic Graphs for Expressions

- DAG – Directed Acyclic Graph
- It identifies the common sub-expressions in the expression
- The function constructing a node first checks, whether an identical node already exists

a + a * (b - c) + (b - c) * d



Bottom-up Evaluation of The S-Attributed Definitions

- Basic idea
  - Evaluated by a bottom-up parser as the input is being parsed
  - The parser keeps the values of the synthesized attributes associated with the grammar symbols on its stack
  - When a reduction is made, the values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the reducing production

Bottom-up Evaluation of The S-Attributed Definitions

- A translator for an S-attributed definition can often be implemented with an LR-parser
- The stack is used to hold information about sub-trees that have been parsed
- We can use extra fields in the parser stack to hold the values of synthesized attributes

Attributes on the Parser Stack

$A \rightarrow XYZ$

top →

Symbol	Attribute
...	...
X	$X.x$
Y	$Y.y$
Z	$Z.z$
...	...

Production	Code Fragment
$L \rightarrow E$ n	*print*(*val*[*top-1*])
$E \rightarrow E_1 + T$	*val*[*ntop*] := *val*[*top-2*] + *val*[*top*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val*[*ntop*] := *val*[*top-2*] * *val*[*top*]
$T \rightarrow F$	
$F \rightarrow (E)$	val[ntop] := val[top-1]
$F \rightarrow$ **digit**	

Attributes on the Parser Stack

Input	Stack	Attributes	Production
3 * 5 + 4 **n**			
* 5 + 4 **n**	3	3	
* 5 + 4 **n**	*F*	3	*F* → **digit**
* 5 + 4 **n**	*T*	3	*T* → *F*
5 + 4 **n**	*T* *	3 -	
+ 4 **n**	*T* * 5	3 - 5	
+ 4 **n**	*T* * *F*	3 - 5	*F* → **digit**
+ 4 **n**	*T*	15	*T*→ *T* * *F*
+ 4 **n**	*E*	15	*E*→ *T*
4 **n**	*E* +	15 -	
n	*E* + 4	15 - 4	
n	*E* + *F*	15 - 4	*F* → **digit**
n	*E* + *T*	15 - 4	*T*→ *F*
n	*E*	19	*E*→ *E* + *T*
	E **n**	19 -	
	L	19	*L*→ E **n**

L-Attributed Definition

- A syntax-directed definition is L-attributed if each inherited attribute of X_j, $1 \leq j \leq n$, on the right side of $A \rightarrow X_1 X_2,...,X_n$ depends only on:
  - the attributes of the symbols $X_1, X_2,...,X_{j-1}$ *to the left of* X_j in the production and
  - the inherited attributes of A

L-Attributed Definition and Evaluation Order

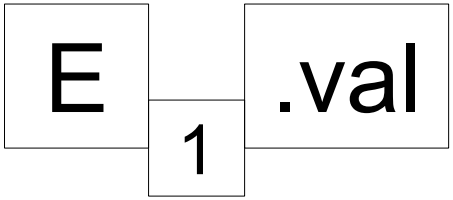
- In L-attributed definition all the attributes can be computed according to the depth-first visit algorithm
  - Hence, in top-down parsing they can be evaluated while parsing
  - Every S-attributed syntax-directed definition is also L-attributed

```
procedure dfvisit(n: node);
begin
    for each child m of n,
    from left to right do begin
        evaluate inherited attributes of m;
        dfvisit(m)
    end;
    evaluate synthesized attributes of n
end
```

L-Attributed Definition Example

Production	Semantic Rules
$S \rightarrow B$	$B.ps := 10$ $S.ht := B.ht$
$B \rightarrow B_1 B_2$	$B_1.ps := B.ps$ $B_2.ps := B.ps$ $B.ht := \max(B_1.ht, B_2.ht)$
$B \rightarrow B_1$ *sub* B_2	$B_1.ps := B.ps$ $B_2.ps :=$ *shrink*$(B.ps)$ $B.ht :=$ *disp*$(B_1.ht, B_2.ht)$
$B \rightarrow$ **text**	$B.ht :=$ **text**$.h * B.ps$

E sub 1 .val



Translation Schemes

- 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
  - Translation schemes is a useful notation for specifying translation during parsing
  - The type of the attributes here can be synthesized attribute or inherited attribute

Rules on A Translation Scheme Design

- Some restrictions should be observed to ensure that an attribute value is available when an action refers to it
- The restrictions can be motivated by L-attributed definitions
- When only synthesized attributes are needed, construct the translation scheme by creating an action consisting of an assignment for each semantic rule, and placing this action at the end of the right side of the associated production

Rules on A Translation Scheme Design

- E.g. Production Semantic Rule

  $T \rightarrow T_1 * F$ $T.val \rightarrow T_1.val * F.val$

- Yield the following production and semantic action:

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

Rules on A Translation Scheme Design

- When both inherited and synthesized attributes are needed
  - An inherited attribute for a symbol on the right side of a production must be computed in an action before that symbol
  - An action must not refer to a synthesized attribute of a symbol to the right of the action
  - A synthesized attribute for the non-terminal on the left can only be computed after all attributes it references have been computed.The action computing such attributes can usually be placed at the end of the right side of the production

Translation Scheme

S	$\rightarrow$		{ $B.ps := 10$ }
		B	{ $S.ht := B.ht$ }
B	$\rightarrow$		{ $B_1.ps := B.ps$ }
		B_1	{ $B_2.ps := B.ps$ }
		B_2	{ $B.ht := max(B_1.ht, B_2.ht)$ }
B	$\rightarrow$		{ $B_1.ps := B.ps$ }
		B_1	
		sub	{ $B_2.ps := shrink(B.ps)$ }
		B_2	{ $B.ht := disp(B_1.ht, B_2.ht)$ }
B	$\rightarrow$	**text**	{ $B.ht :=$ **text**$.h * B.ps$ }

Top-Down Translation

Basic Idea

- Use L-attributed definition and translation schemes during predictive parsing
- Extend the algorithm for eliminating left recursion to a translation schemes with synthesized attributes

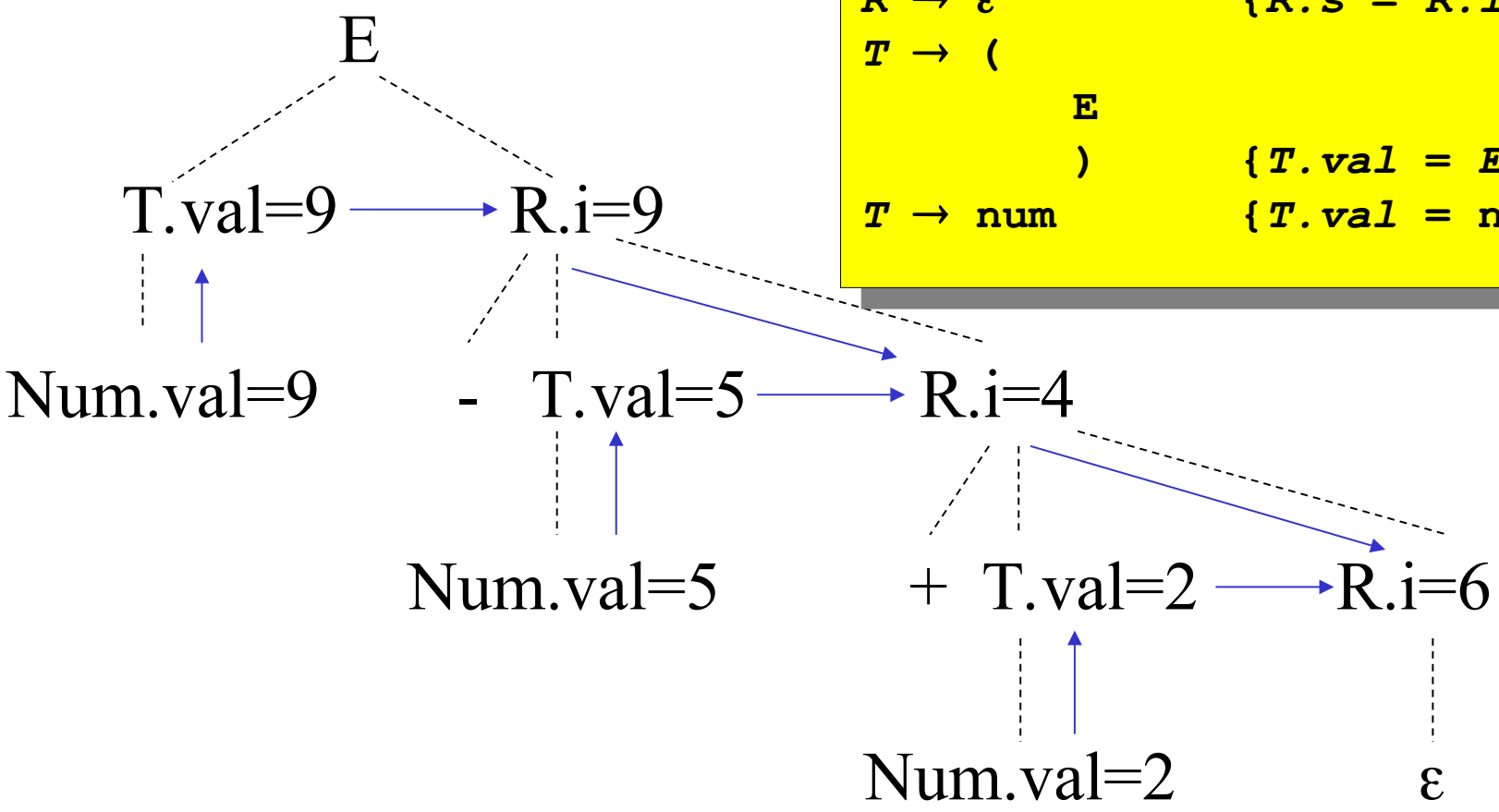
Eliminating Left Recursion from a Translation Scheme

```
E → E1 + T   {E.val = E1.val + T.val}
E → E1 - T   {E.val = E1.val - T.val}
E → T        {E.val = T.val}
T → (E)      {T.val = E.val}
T → num      {T.val = num.val}
```

```
E → T          {R.i = T.val}
      R        {E.val = R.s}
R → +
      T        {R1.i = R.i + T.val}
      R1       {R.s = R1.s}
R → -
      T        {R1.i = R.i - T.val}
      R1       {R.s = R1.s}
R → ε          {R.s = R.i}
T → (
      E
      )        {T.val = E.val}
T → num      {T.val = num.val}
```

Eliminating Left Recursion from a Translation Scheme

```
E → T          {R.i = T.val}
      R        {E.val = R.s}
R → +
      T        {R1.i = R.i + T.val}
      R1       {R.s = R1.s}
R → -
      T        {R1.i = R.i - T.val}
      R1       {R.s = R1.s}
R → ε          {R.s = R.i}
T → (
      E
      )        {T.val = E.val}
T → num        {T.val = num.val}
```



Eliminating Left Recursion from a Translation Scheme

- Algorithm for eliminating left recursion from a translation scheme with synthesized attribute
- Suppose we have the following translation scheme

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

  $A \rightarrow X \quad \{ A.a = f(X.x) \}$
- We assume here each grammar symbol has a synthesized attribute, and f and g are arbitrary functions.

Eliminating Left Recursion from a Translation Scheme

- After eliminating left recursion:

  $A \rightarrow XR$

  $R \rightarrow YR \mid \varepsilon$

- Taking the semantic actions into account, the transformed scheme becomes:

  $A \rightarrow X \quad \{ R.i = f(X.x) \}$

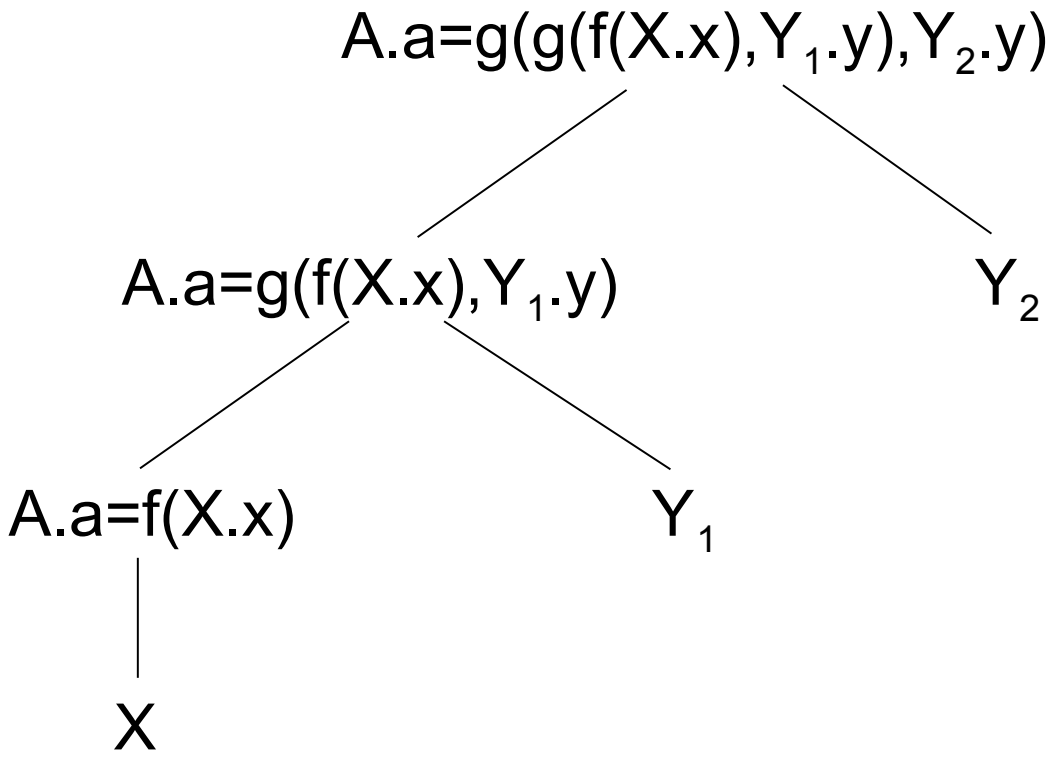
  $\quad\quad R \quad \{ A.a = R.s \}$

  $R \rightarrow Y \quad \{ R_1.i = g(R.i, Y.y) \}$

  $\quad\quad R_1 \quad \{ R.s = R_1.s \}$

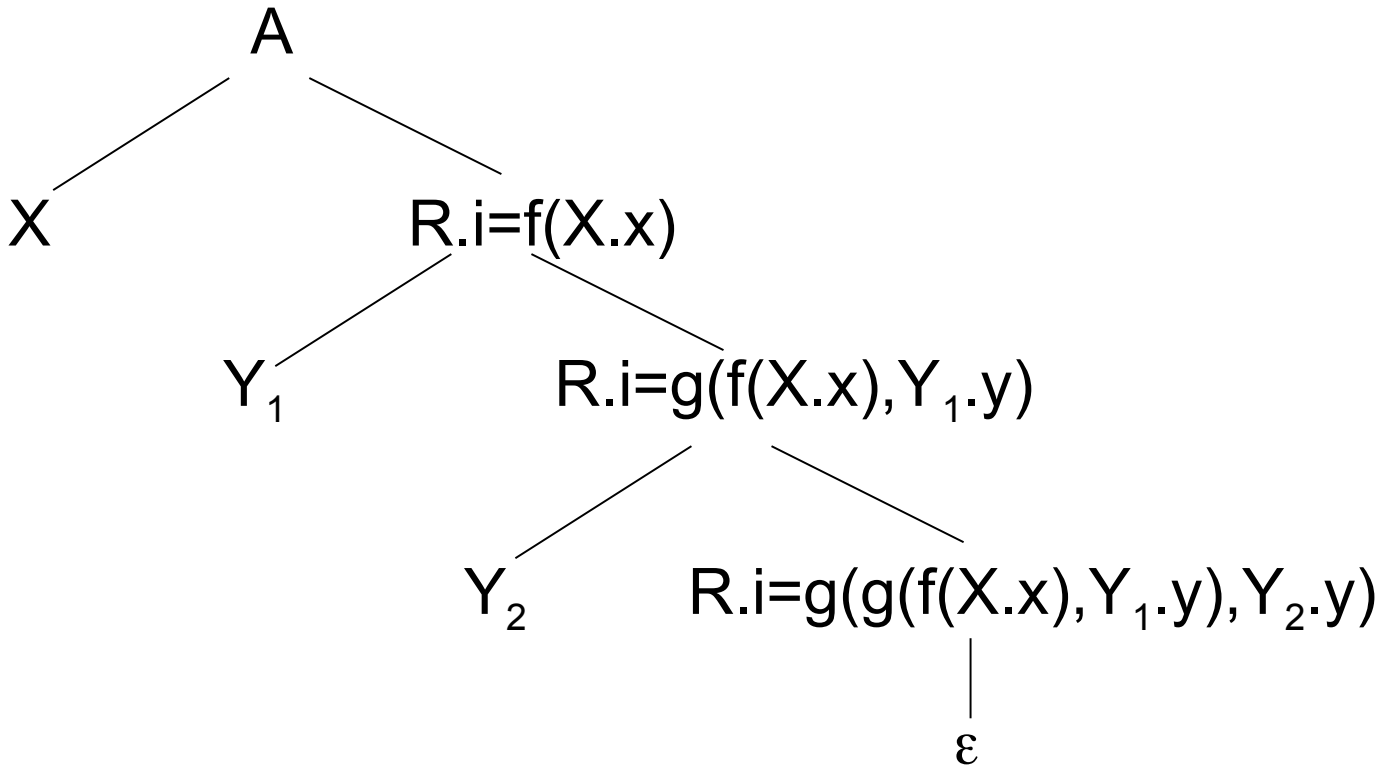
  $R \rightarrow \varepsilon \quad \{ R.s = R.i \}$

Eliminating Left Recursion from a Translation Scheme



One way of computing an attribute value

Eliminating Left Recursion from a Translation Scheme



Another way of computing an attribute value

Bottom-Up Translation

Bottom-up Evaluation of Inherited Attributes

- Implement L-attributed definitions in the framework of bottom-up parsing
- Use a transformation that makes all embedded actions in a translation scheme occur at the right ends of their productions
  - Insert new marker non-terminals generating ε into the base grammar
  - Replace each embedded action by a distinct marker non-terminal M and attach the action to the end of the production $M \rightarrow \varepsilon$
- LL(1) grammar after this transformation will still be LL(1), but LR(1) grammar may no longer be LR(1)

Removing Embedded Actions from Translation Schemes

$E \rightarrow \quad T\ R$

$R \rightarrow \quad + T\ \{\ print('+')\}\ R \mid -\ T\ \{print('-')\}\ R \mid \in$

$T \rightarrow \quad \text{num}\ \{print(\text{num}.val)\}$

$E \rightarrow \quad T\ R$

$R \rightarrow \quad + T\ M\ R \mid -\ T\ N\ R \mid \in$

$T \rightarrow \quad \text{num}\ \{print(\text{num}.val)\}$

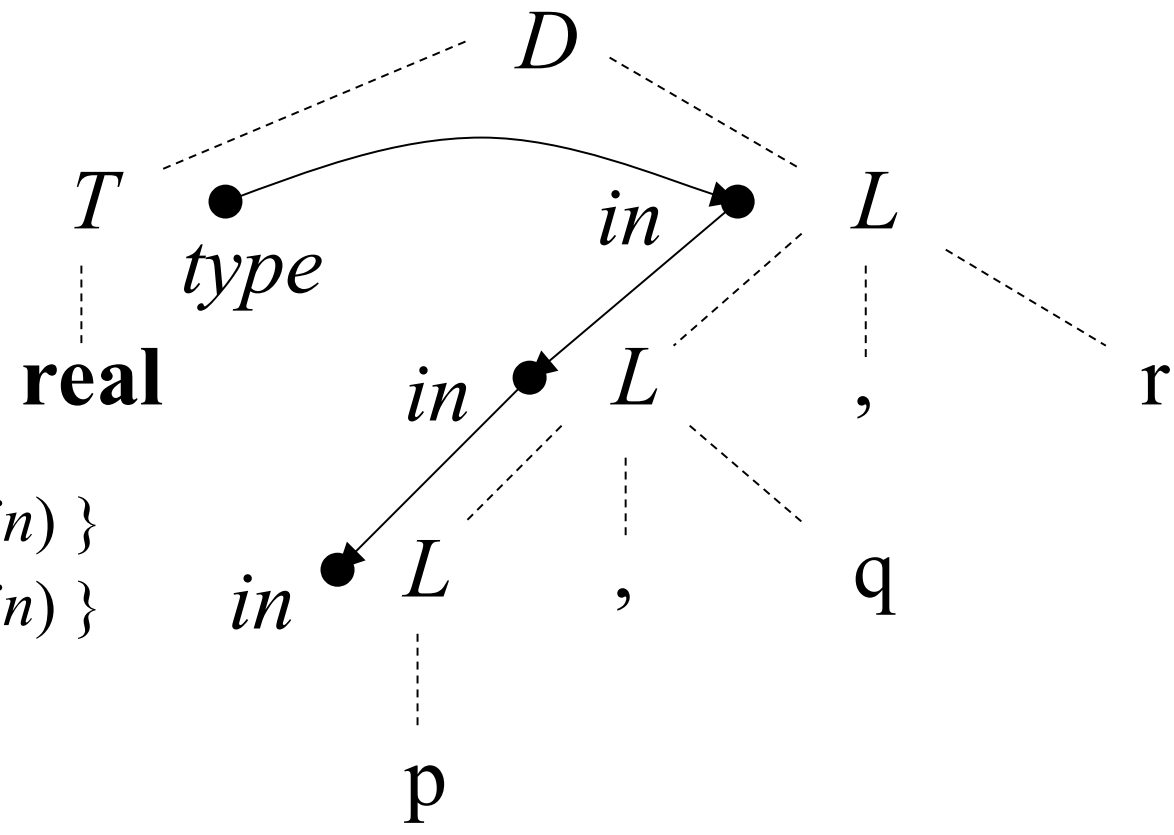
$M \rightarrow \quad \in\ \{\ print('+')\}$

$N \rightarrow \quad \in\ \{\ print('-')\}$

Inherited Attributes and Bottom-Up Translation

For this grammar, always *L.in* = *T.type*

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



Inherited Attributes and Bottom-Up Translation

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

State	Input	Production
-	**real** p , q , r	
real	p , q , r	
T	p , q , r	*T* → **real**
T p	, q , r	
T L	, q , r	*L* → **id**
T L ,	q , r	
T L , q	, r	
T L	, r	*L* → *L* , **id**
T L ,	r	
T L , r		
T L		*L* → *L* , **id**
D		*D* → *TL*

T is always immediately under *L* at the parser stack

Inherited Attributes and Bottom-Up Translation

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

$\quad\quad\quad L$

$T \rightarrow \quad \textbf{int} \quad \{ T.type := integer \}$

$T \rightarrow \quad \textbf{real} \quad \{ T.type := real \}$

$L \rightarrow \quad \{ L_1.in := L.in \}$

$\quad\quad\quad L_1 \textbf{, id} \quad \{ addtype(id.entry, L.in) \}$

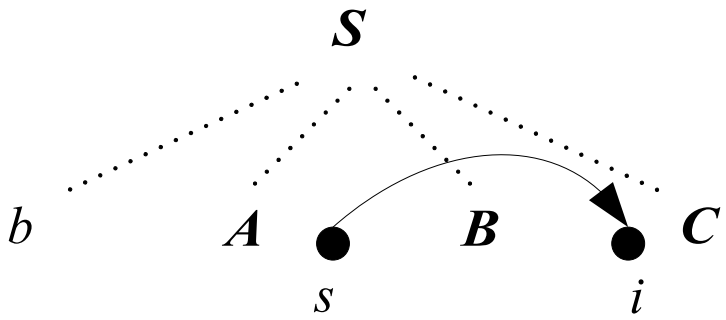
$L \rightarrow \quad \textbf{id} \quad \{ addtype(id.entry, L.in) \}$

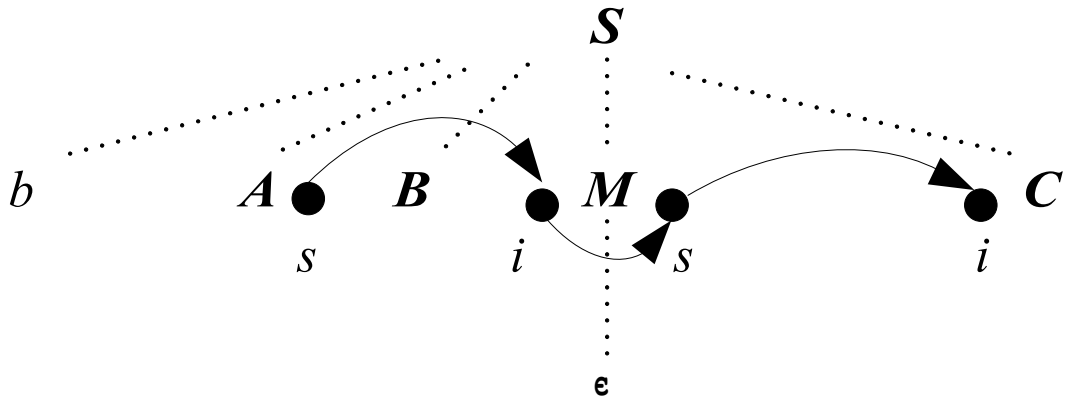
Production	Code Fragment
$D \rightarrow T\ L\ ;$	
$T \rightarrow$ **int**	*val*[*ntop*] := *integer*
$T \rightarrow$ **real**	*val*[*ntop*] := *real*
$L \rightarrow L$, **id**	*addtype(val*[*top*], *val*[*top-3*]*)*
$L \rightarrow$ **id**	*addtype(val*[*top*], *val*[*top-1*]*)*

Inherited Attributes - Application of Marker Nonterminal

$S \rightarrow$	aAC	$C.i := A.s$
$S \rightarrow$	$bABC$	$C.i := A.s$
$C \rightarrow$	c	$C.s := g(C.i)$

$S \rightarrow$	aAC	$C.i := A.s$
$S \rightarrow$	$bABMC$	$M.i := A.s;\ C.i := M.s$
$C \rightarrow$	c	$C.s := g(C.i)$
$M \rightarrow$	$\in$	$M.s := M.i$





Application of Marker Nonterminal-General Case

$S \rightarrow$	aAC	$C.i := f(A.s)$

$S \rightarrow$	$aANC$	$N.i := A.s;\ C.i := N.s$
$N \rightarrow$	$\in$	$N.s := f(N.i)$

Bottom-Up Processing of L-Attributed Definition

$S \rightarrow$	$L\ B$	$B.ps := L.s$
		$S.ht := B.ht$
$L \rightarrow$	$\in$	$L.s := 10$
$B \rightarrow$	$B_1\ M\ B_2$	$B_1.ps := B.ps$
		$M.i := B.ps$
		$B_2.ps := M.s$
		$B.ht := max(B_1.ht, B_2.ht)$
$B \rightarrow$	B_1 **sub** $N\ B_2$	$B_1.ps := B.ps$
		$N.i := B.ps$
		$B_2.ps := N.s$
		$B.ht := disp(B_1.ht, B_2.ht)$
$B \rightarrow$	**text**	$B.ht :=$ **text**$.h * B.ps$
$M \rightarrow$	$\in$	$M.s := M.i$
$N \rightarrow$	$\in$	$N.s := shrink(N.i)$

Replacing Inherited by Synthesized Attributes

```
D → L:T  {L.type=T.type}
T → integer|char {T.type=integer|char}
L → L1,id {L1.type=L.type; addtype(id.entry,L.type)
L → id {addtype(id.entry,L.type)}
```

```
D → id L {addtype(id.entry,L.type)}
L → ,id L1 {L.type=L1.type; addtype(id.entry,L1.type)}
L → :T {L.type=T.type}
T → integer|char {T.type=integer|char}
```

- Now, the type can be carried along as a synthesized attribute *L.type*. As each identifier is generated by *L*, its type can be entered into the symbol table