

CS 346: Syntax directed translation

Resource: Textbook

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, “*Compilers: Principles, Techniques, and Tools*”, Addison-Wesley, 1986.

Where are we?

- **Ultimate goal:** generate machine code
- Before we generate code, we must collect information about the program
- Front end
 - scanning (recognizing words) CHECK
 - parsing (recognizing syntax) CHECK
 - semantic analysis (recognizing meaning)
 - There are issues deeper than structure. Consider:

```
int func (int x, int y);  
int main () {  
    int list[5], i, j;  
    char *str;  
    j = 10 + 'b';  
    str = 8;  
    m = func ("aa", j, list[12]);  
    return 0;  
}
```

This code is syntactically correct, but will not work. What problems are there?

Beyond syntax analysis

- An identifier named x has been recognized
 - Is x a scalar, array or function?
 - How big is x ?
 - If x is a function, how many and what type of arguments does it take?
 - Is x declared before being used?
 - Where can x be stored?
 - Is the expression $x+y$ type-consistent?
- **SEMANTIC Analysis** is the phase where we collect information about the **types of expressions** and check for **type related errors**
- The more information we can collect at compile time, the less overhead we have at run time

Semantic analysis

- Collecting type information may involve "computations"
 - What is the type of $x+y$ given the types of x and y ?
- Tool: *attribute grammars*
 - CFG
 - Each grammar symbol has associated attributes
 - Grammar augmented by rules (*semantic actions*) that specify how the values of attributes are computed from other attributes
 - The process of using semantic actions to evaluate attributes is called *syntax-directed translation*
 - Examples:
 - Grammar of declarations
 - Grammar of signed binary numbers

Attribute grammars

Example 1: Grammar of declarations

Production	Semantic rule
$D \rightarrow T L$	$L.in = T.type$
$T \rightarrow \text{int}$	$T.type = \text{integer}$
$T \rightarrow \text{char}$	$T.type = \text{character}$
$L \rightarrow L_1, \text{id}$	$L_1.in = L.in$ $\text{addtype}(\text{id.index}, L.in)$
$L \rightarrow \text{id}$	$\text{addtype}(\text{id.index}, L.in)$

Syntax-Directed Translation

- **Grammar symbols**
 - associated with **attributes** to associate information with the programming language constructs that they represent
- **Evaluation of values of attributes**
 - by the **semantic rules** associated with the production rules
- **Evaluation of semantic rules**
 - may generate intermediate codes
 - may put information into the symbol table
 - may perform type checking
 - may issue error messages
 - may perform some other activities
 - *in fact, they may perform almost any activities !*
- An attribute may hold almost any thing
 - a string, a number, a memory location, a complex record

Syntax-Directed Definitions and Translation Schemes

- Two notations for semantic rules:
 - **Syntax-Directed Definitions**
 - **Translation Schemes**
- **Syntax-Directed Definitions:**
 - give high-level specifications for translations
 - hide many implementation details such as order of evaluation of semantic actions
 - production rule is associated with a set of semantic actions, but do not have any prior information about when they will be evaluated
- **Translation Schemes:**
 - indicate the order of evaluation of semantic actions associated with a production rule
 - In other words, translation schemes give a little bit information about implementation details

Syntax-Directed Definitions

- Syntax-directed definition is a generalization of a CFG in which:
 - Each grammar symbol is associated with a set of attributes
 - Set of attributes for a grammar symbol partitioned into two subsets
 - **synthesized** attributes
 - **inherited** attributes
 - Each production rule is associated with a set of semantic rules
- **Synthesized attribute for a non-terminal A at parse tree node N**
 - Defined by a semantic rule associated with the production at N (A: **head**)
 - Depends
 - Attribute values at the children of N
 - Attribute values at node N
 - Could also depend on the inherited attribute (s) at N

Syntax-Directed Definitions

- **Inherited attribute for a non-terminal A at parse tree node N**
 - Attributes of N's parent
 - Attributes of N itself
 - Attributes of N's siblings
- *Terminals can have synthesized but not inherited attribute*
- *Dependency graph*
 - represents dependencies between *attributes of semantic rules*
 - determines the evaluation order of semantic rules
- What we mean by evaluation of a semantic rule?
 - defines the value of an attribute
 - may also have some side effects such as *printing a value*

Syntax-Directed Definition

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

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

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 of 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 ($A \rightarrow \alpha$)

Attribute Grammar

- 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 can evaluate
 - value of an attribute or
 - 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)

Annotated Parse Tree

- **Annotated parse tree:** 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**) of the parse tree
- Order of computations depends on the dependency graph induced by the semantic rules

Syntax-Directed Definition -- Example

Production

$L \rightarrow E \text{ return}$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{digit}$

Semantic Rules

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

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

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

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

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

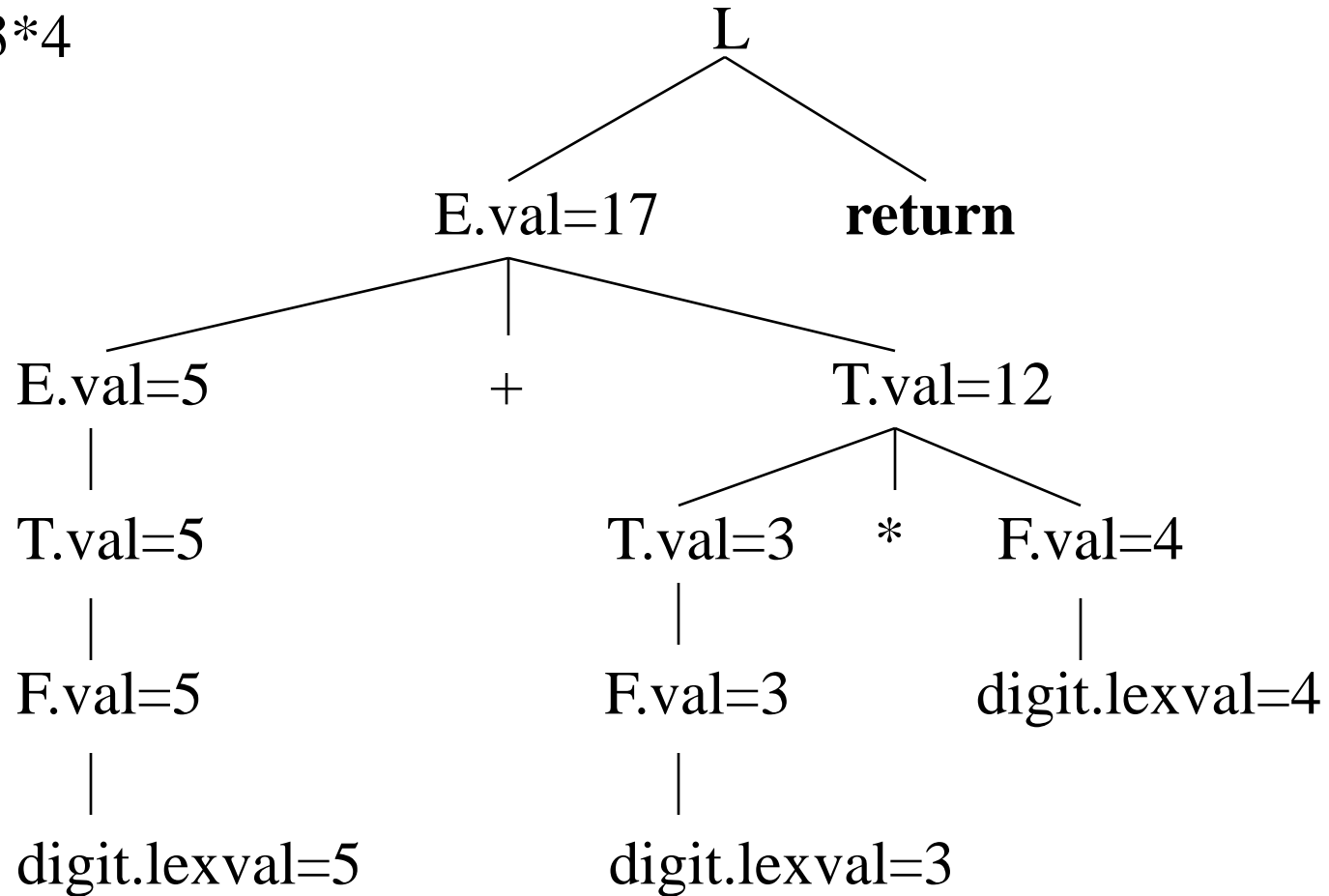
$F.\text{val} = E.\text{val}$

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

- 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 -- Example

Input: 5+3*4



Dependency Graph

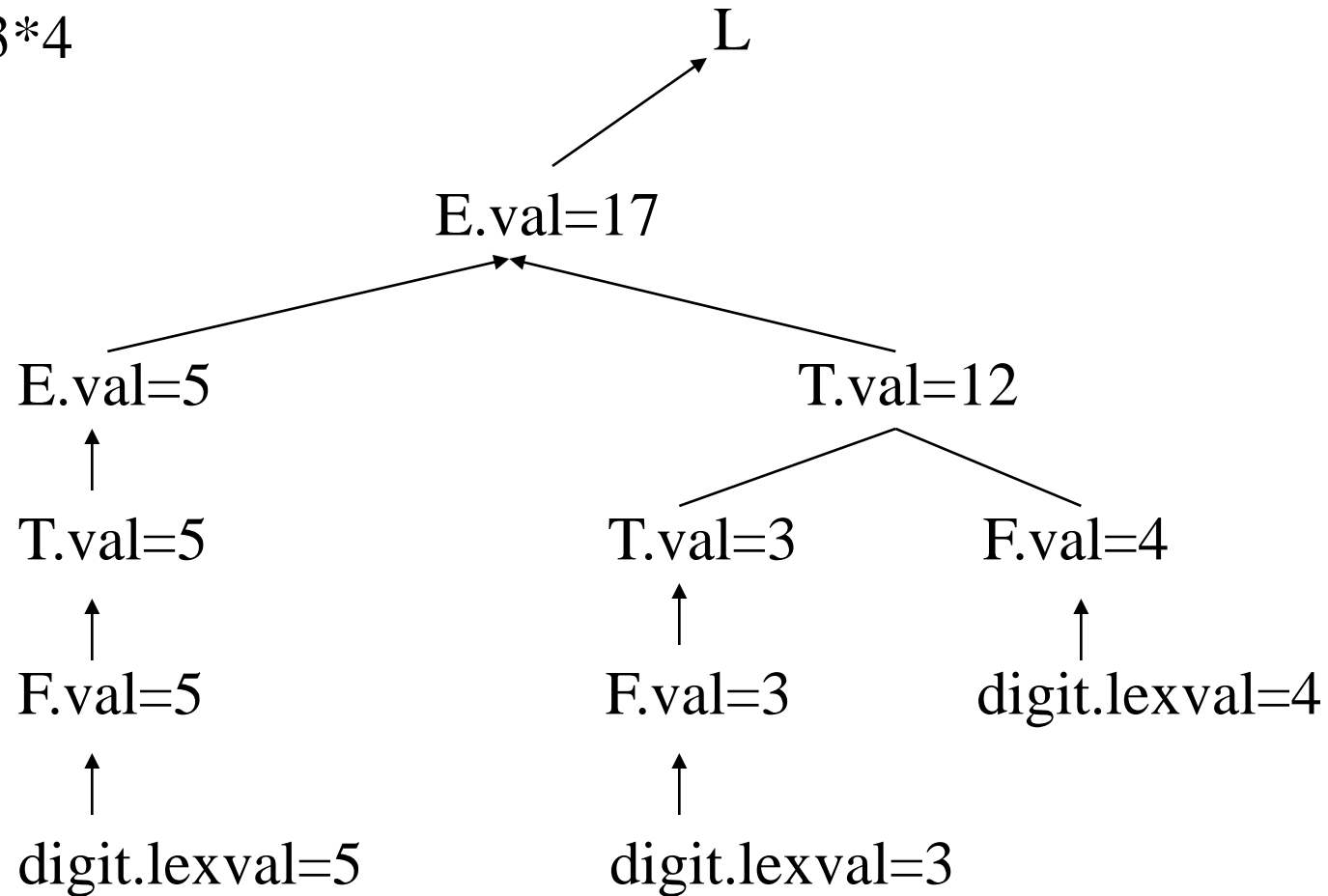
- Depicts the flow of information among the attribute instances in a parse tree
- Useful for determining an evaluation order for the attribute instances in a given parse tree
- Annotated parse tree vs. dependency graph
 - Annotated parse tree shows the values of attributes
 - Dependency graph help us determine how the values can be computed
- Edge from attribute instance X.a to Y.b
 - Value of X.a is necessary to compute Y.b

Dependency Graph

- For each grammar symbol X , the dependency graph has a node for each attribute associated with X
- Semantic rule associated with production p defines the value of *synthesized* attribute $A.b$ in terms of $X.c$
 - Add an edge from $X.c$ to $A.b$ ($X.c \rightarrow A.b$)
 - Create edge from the children of node labeled A to it ($\text{CHILDREN}(A) \rightarrow A$)
- Semantic rule associated with production p defines the value of *inherited* attribute $B.c$ in terms of $X.a$
 - Add an edge from $X.a$ to $B.c$ ($X.a \rightarrow B.c$)
 - Create edges from the parent of $X.a$ or siblings of $X.a$ to $X.a$ ($\text{PAREN}(X.a) / \text{SIBLINGS}(X.a) \rightarrow X.a$)

Dependency Graph

Input: $5+3*4$



Syntax-Directed Definition – Inherited Attributes

Production

$D \rightarrow T L$

$T \rightarrow \mathbf{int}$

$T \rightarrow \mathbf{real}$

$L \rightarrow L_1 \mathbf{id}$

$L \rightarrow \mathbf{id}$

Semantic Rules

$L.in = T.type$

$T.type = \text{integer}$

$T.type = \text{real}$

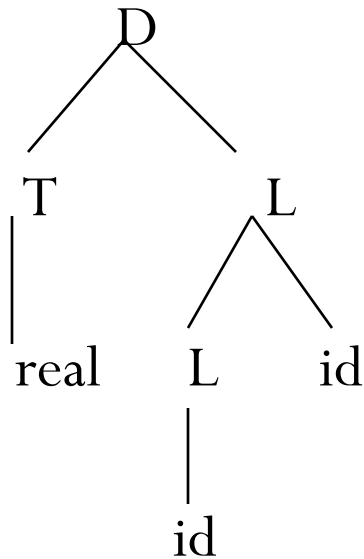
$L_1.in = L.in, \text{ addtype}(\mathbf{id}.entry, L.in)$

$\text{addtype}(\mathbf{id}.entry, L.in)$

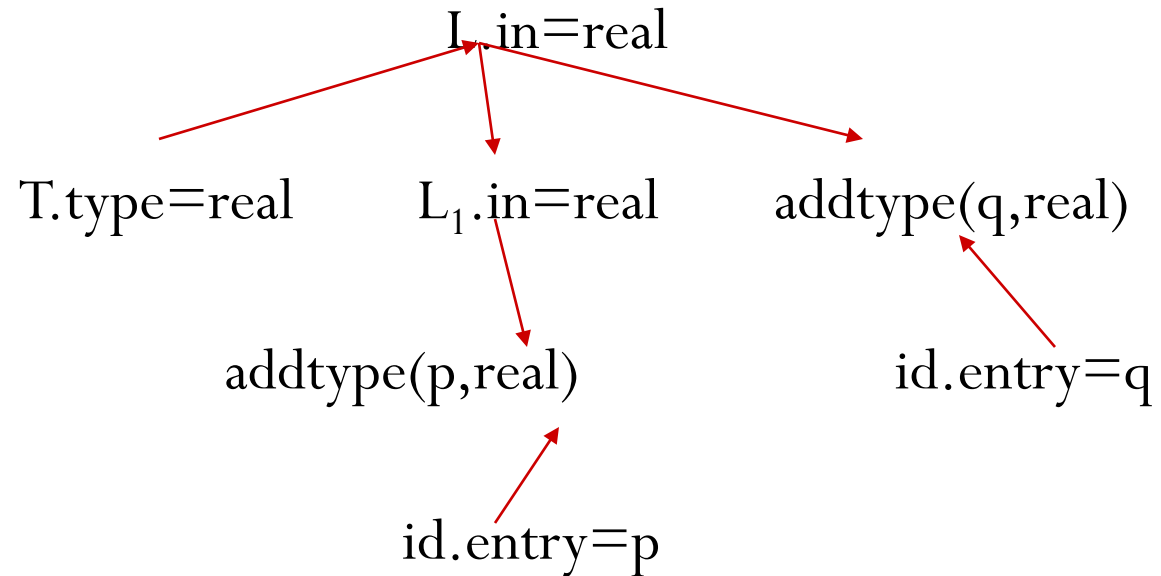
- Symbol T is associated with a synthesized attribute *type*.
- Symbol L is associated with an inherited attribute *in*.

A Dependency Graph – Inherited Attributes

Input: real p q



parse tree



dependency graph

Ordering the Evaluation of Attributes

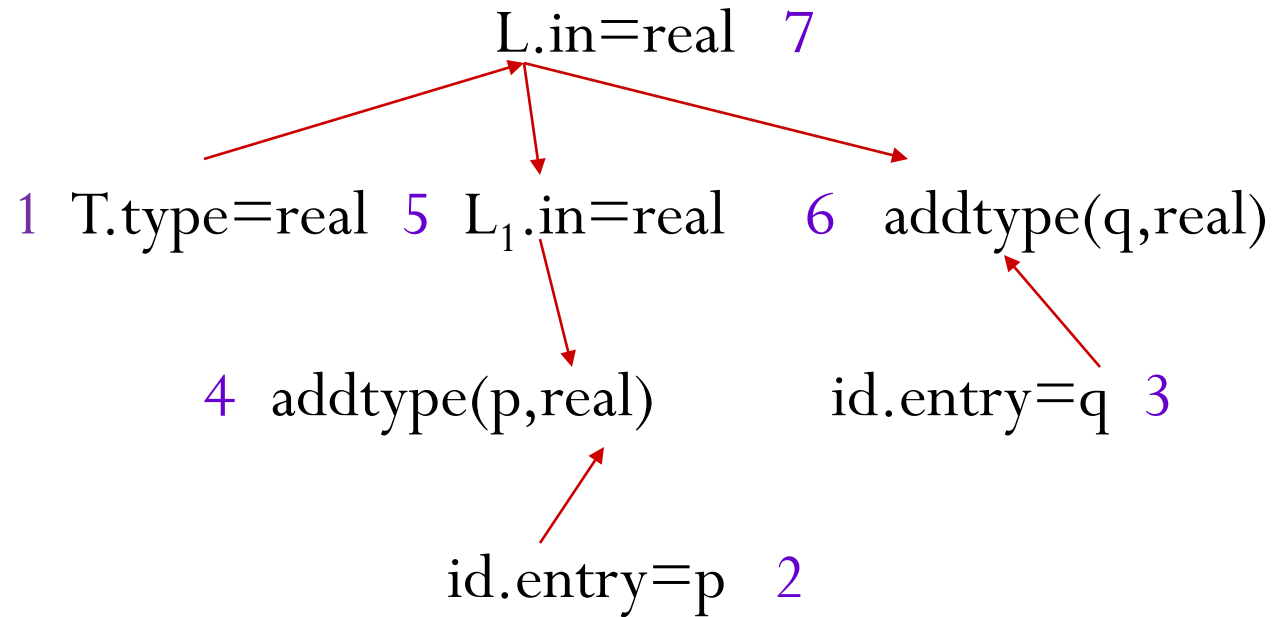
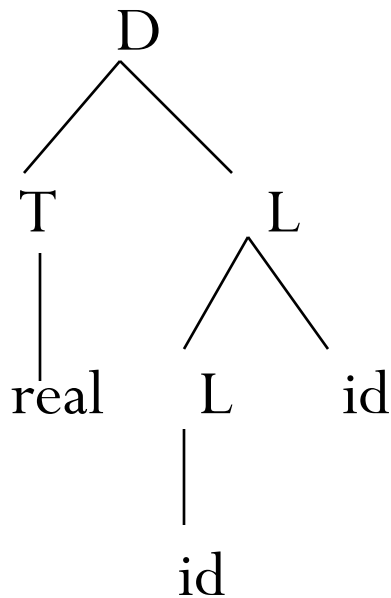
- Dependency graph characterizes the possible orders of evaluating the attributes at the various nodes of parse tree
- An edge of $M \rightarrow N$: attributes corresponding to M must be evaluated before the attributes of N
- **Allowable order of evaluation**: sequence of nodes N_1, N_2, \dots, N_K such that if there is an edge in the dependency graph from N_i to N_j then $i < j$
 - Such order embeds a directed graph in a linear order (*Topological sort*)
- Dependency graph without *cycle*
 - At least one topological sort
 - There exists a node with *no incoming edge*

Ordering the Evaluation of Attributes

- Absence of such node
 - proceed from predecessor to predecessor until we come across an already visited node (i.e., *cycle!*)
 - Treat this node as the first in topological order
 - Remove it from the dependency graph
 - Repeat the process on the remaining nodes
- Dependency graph with *cycle*
 - no topological sorts
 - Not possible to evaluate SDD

Topological Sorts – An Example

Input: `real p q`



Topological sorts: 1 2 3 4 5 6 7 (!)

1 2 3 7 5 6 4

Syntax-Directed Definition – Example2

Production Semantic Rules

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

- Symbols E, T, and F are associated with synthesized attributes *loc* and *code*
- 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

Syntax-directed definitions (SDD) contd..

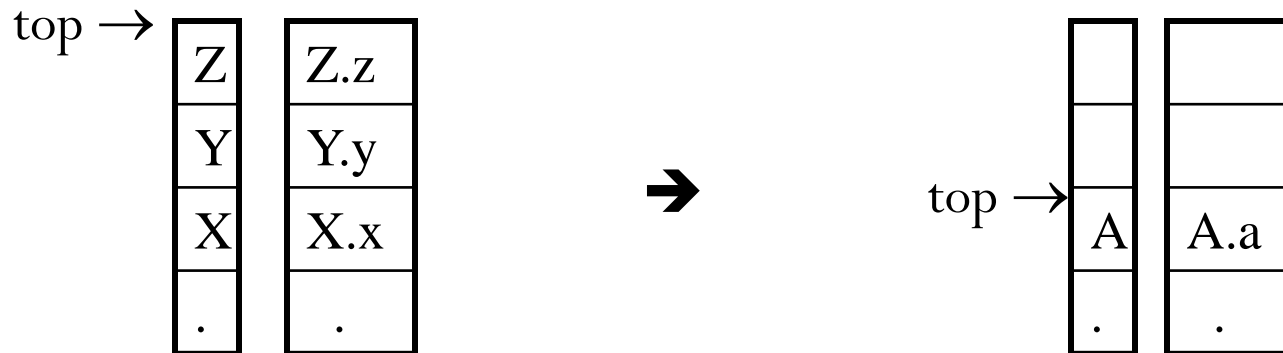
- SDD: useful for SDD translations
- Translator generation for an arbitrary SDD can be difficult
- Evaluate the semantic rules during parsing (in some cases)
 - i.e. in a single pass, *parse and evaluate semantic rules*
- Two sub-classes of the SDD:
 - **S-Attributed Definitions:** only synthesized attributes used in the definitions
 - **L-Attributed Definitions:** synthesized attributes + inherited attributes (in a restricted fashion)
- Implementation of S-attributed definitions and L-attributed definitions
 - **S-attributed:** Can be in conjunction with bottom-up parsing (*S: synthesized*)
 - **L-attributed:** Virtually all translations are done during parsing (*L: Left-to-right*)

Bottom-Up Evaluation of S-Attributed Definitions

- Put the values of the synthesized attributes of the grammar symbols into a parallel stack
 - When an entry of the parser stack holds a grammar symbol X (terminal or non-terminal), the corresponding entry in the parallel stack will hold the synthesized attribute(s) of the symbol X
- Evaluate the values of the attributes during reductions
- Perform *post order* traversal of parse tree \rightarrow *Bottom-up parsing* \rightarrow Order in which LR parser *reduces a production to head*

$A \rightarrow XYZ$ $A.a = f(X.x, Y.y, Z.z)$ where all attributes are synthesized.

stack *parallel-stack*



Bottom-Up Eval of S-Attributed Definitions (cont.)

Production

$L \rightarrow E \text{ return}$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{digit}$

Semantic Rules

$\text{print}(\text{val}[\text{top}-1])$

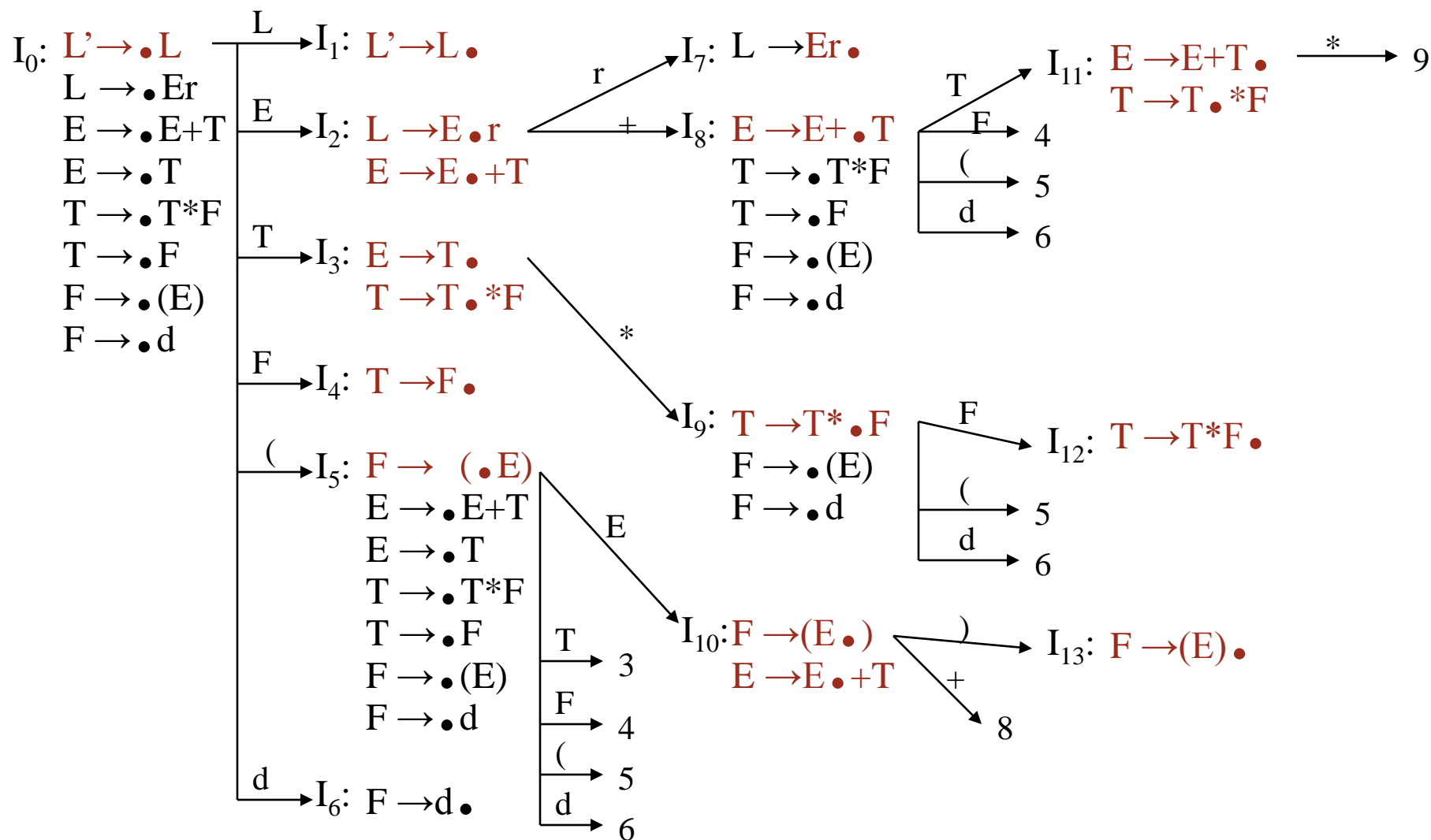
$\text{val}[\text{ntop}] = \text{val}[\text{top}-2] + \text{val}[\text{top}]$

$\text{val}[\text{ntop}] = \text{val}[\text{top}-2] * \text{val}[\text{top}]$

$\text{val}[\text{ntop}] = \text{val}[\text{top}-2]$

- 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*)

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
0d6	5	+3*4r	F→d	F.val=d.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+8d6	5-3	*4r	F→d	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*9d6	5-3-4	r	F→d	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	

L-Attributed Definitions

- S-attributed definitions can be efficiently implemented
- On looking for a larger (larger than S-attributed definitions) subset of SDD which can be efficiently evaluated
→ **L-Attributed Definitions**
- L-attributed definitions can always be evaluated by the *depth first visit of the parse tree*
- This can also be evaluated during parsing

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 (*synthesized / inherited*) 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
 3. *Synthesized/inherited* attributes associated with X_j (should not be any cycle)
- Every S-attributed definition is L-attributed, the restrictions only apply to the inherited attributes (not to synthesized attributes)

A Definition which is NOT L-Attributed

Productions

$A \rightarrow L M$

$A \rightarrow Q R$

Semantic Rules

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

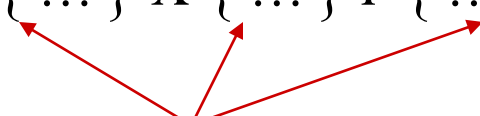
$R.in = r(A.in), Q.in = q(R.s), A.s = f(Q.s)$

- SDD is not L-attributed because the semantic rule $Q.in = q(R.s)$ violates the restrictions of L-attributed definitions
- $Q.in$ 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 able to evaluate the value of $Q.in$ before we enter to Q

Translation Schemes

- In a SDD, we do not say anything about the evaluation times of the semantic rules (i.e., when the semantic rules associated with a production should be evaluated?)
- A **translation scheme**: a CFG 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

- Restrictions to designing a SDT

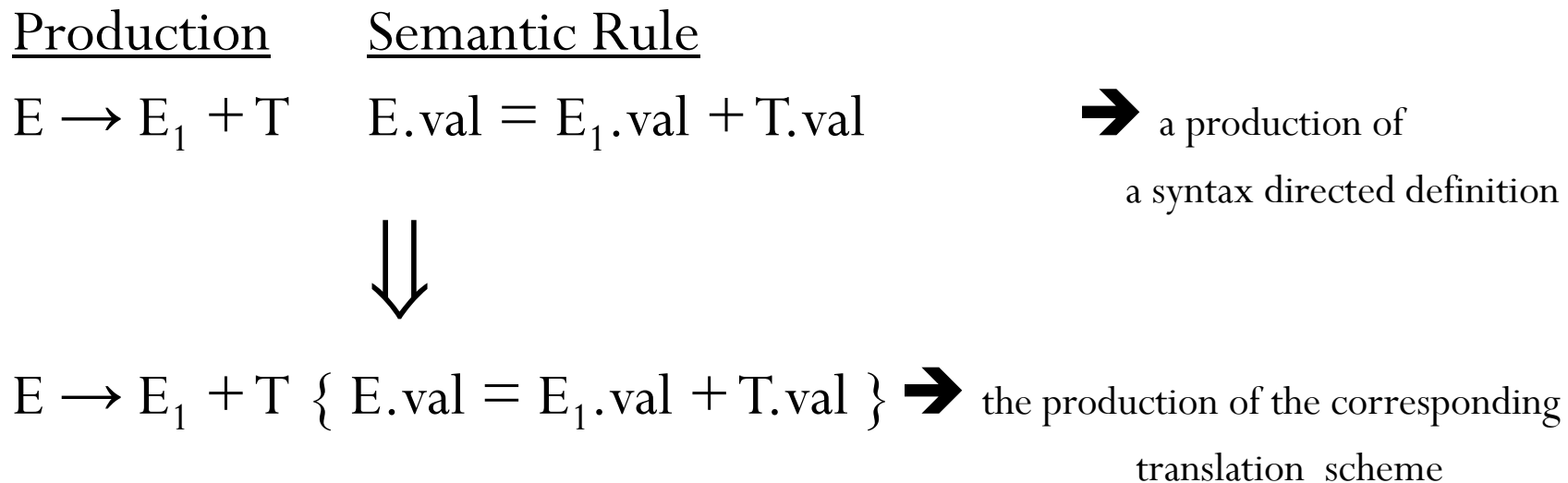
- to ensure that an attribute value is available when a semantic action refers to that attribute
- restrictions (motivated by L-attributed definitions) ensure that a semantic action does not refer to an attribute that has not yet been computed

Some notes:

- 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

Translation Schemes for S-attributed Definitions

- For S-attributed SDD, the construction of the corresponding SDT is simple
 - Each associated *semantic rule* in a *S-attributed SDD* will be inserted as a *semantic action* into the *end of the right side of the associated production*



SDT's with Actions Inside Productions

- Actions may be placed at any place within the body of the production
 - $B \rightarrow X\{a\}Y$
 - X : *terminal*: Action a performed after the recognition of X
 - X : *non-terminal*: Action a performed after the recognition of all terminals derived from X
- Bottom-up parsing: action a performed as soon as the occurrence of X appears on the top of the parsing stack
- Top-down parsing: action a performed
 - Just before we attempt to expand the occurrence of Y that appears on the top of the parsing stack (Y : *non-terminal*)
 - Check for Y on the input string (Y : *terminal*)

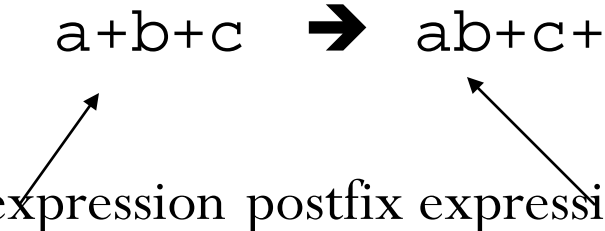
A Translation Scheme Example

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

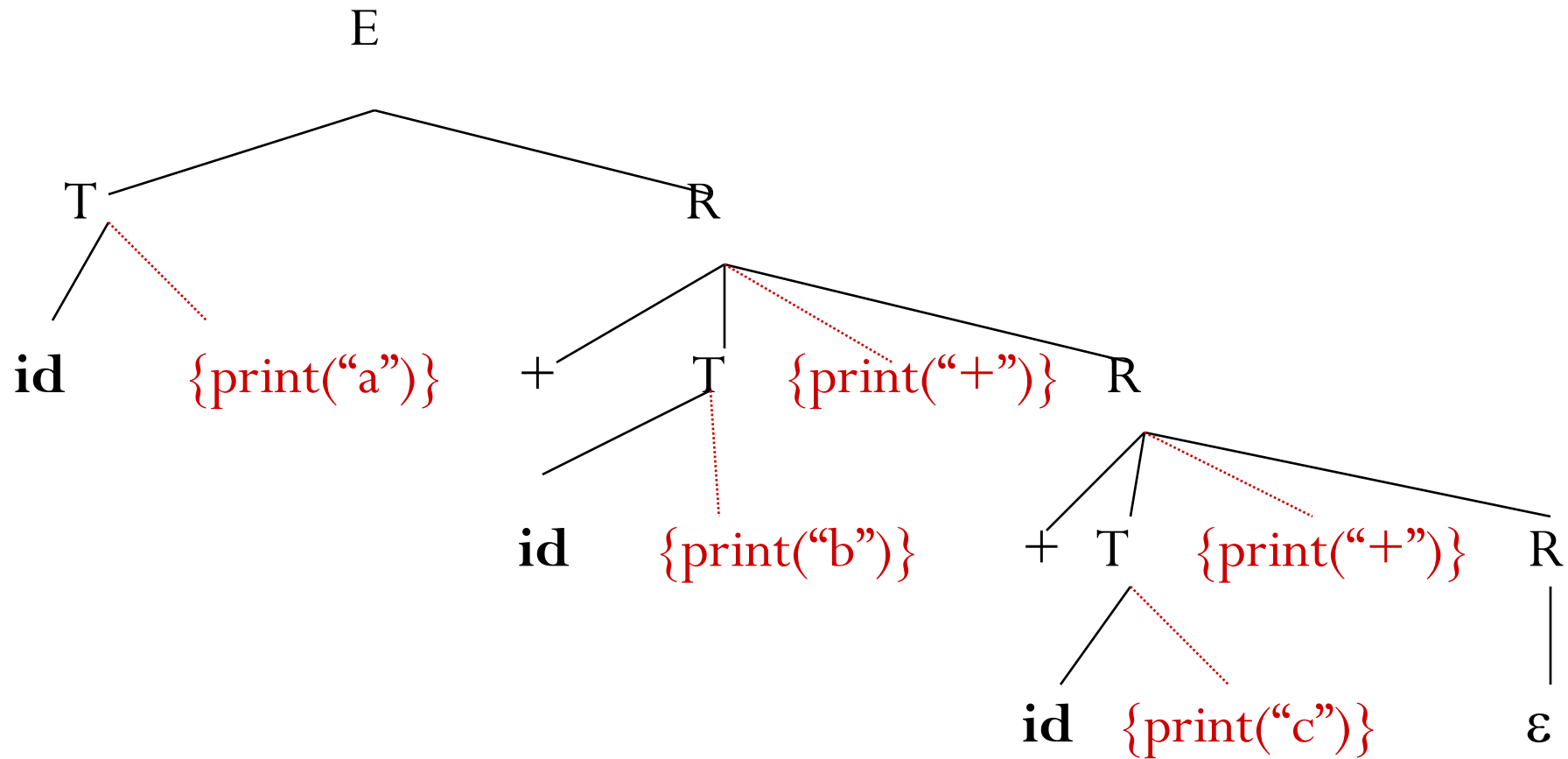
$$E \rightarrow T R$$
$$R \rightarrow + T \{ \text{print}(\text{"+"}) \} R_1$$
$$R \rightarrow \varepsilon$$
$$T \rightarrow \text{id} \{ \text{print}(\text{id.name}) \}$$

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

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

Inherited Attributes in Translation Schemes

- SDT with both synthesized and inherited attributes:
 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 any 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 (normally, this semantic action appears at the end of the right side of the production)

With a L-attributed SDD, it is always possible to construct a corresponding translation scheme which satisfies the above three conditions

Predictive Translation

- **Input:** translation scheme based on a grammar suitable for predictive parsing
- **Output:** Code for a syntax-directed translator
- **Method:**
 1. For each nonterminal A , construct a function with
 - Input parameters:* one for each inherited attribute of A ;
 - Return value:* synthesized attributes of A ;
 - Local variables:* one for each attribute of each grammar symbol that appears in a production for A
 2. Code for non-terminal A decides what production to use based on the current input symbol (*switch statement*). Code for each production forms one case of a switch statement.

Predictive Translation

3. In the code for a production, tokens, nonterminals, actions in the RHS are considered left to right.

(i) For token X: save X.s in the variable created for X;

generate a call to match X and advance input.

(ii) For nonterminal B: generate an assignment

$c = B(b_1, b_2, \dots, b_k);$ where:

b_1, b_2, \dots are variables corresponding to inherited attributes of B,

c is the variable for synthesized attribute of B,

B is the function created for B.

(iii) For an action, copy the code into the function, replacing each reference to an attribute by the variable created for that attribute.

A Translation Scheme with Inherited Attributes

$$D \rightarrow T \textbf{id} \{ \text{addtype}(\textbf{id.entry}, T.\text{type}), L.\text{in} = T.\text{type} \} L$$
$$T \rightarrow \textbf{int} \{ T.\text{type} = \text{integer} \}$$
$$T \rightarrow \textbf{real} \{ T.\text{type} = \text{real} \}$$
$$L \rightarrow \textbf{id} \{ \text{addtype}(\textbf{id.entry}, L.\text{in}), L_1.\text{in} = L.\text{in} \} L_1$$
$$L \rightarrow \varepsilon$$

- 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

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

- 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 (cont.)

inherited attribute

synthesized attribute

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

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

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

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

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

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

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

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

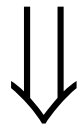
$F \rightarrow \text{digit} \{ F.val = \text{digit.lexval} \}$

Eliminating Left Recursion (in general)

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

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

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



eliminate left recursion

inherited attribute of the new non-terminal

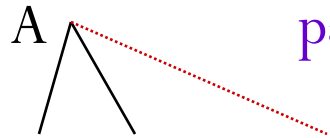
synthesized attribute of the new non-terminal

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

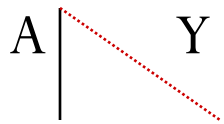
$R \rightarrow Y \{ R_1.in = g(R.in, Y.y) \} R_1 \{ R.syn = R_1.syn \}$

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

Evaluating attributes



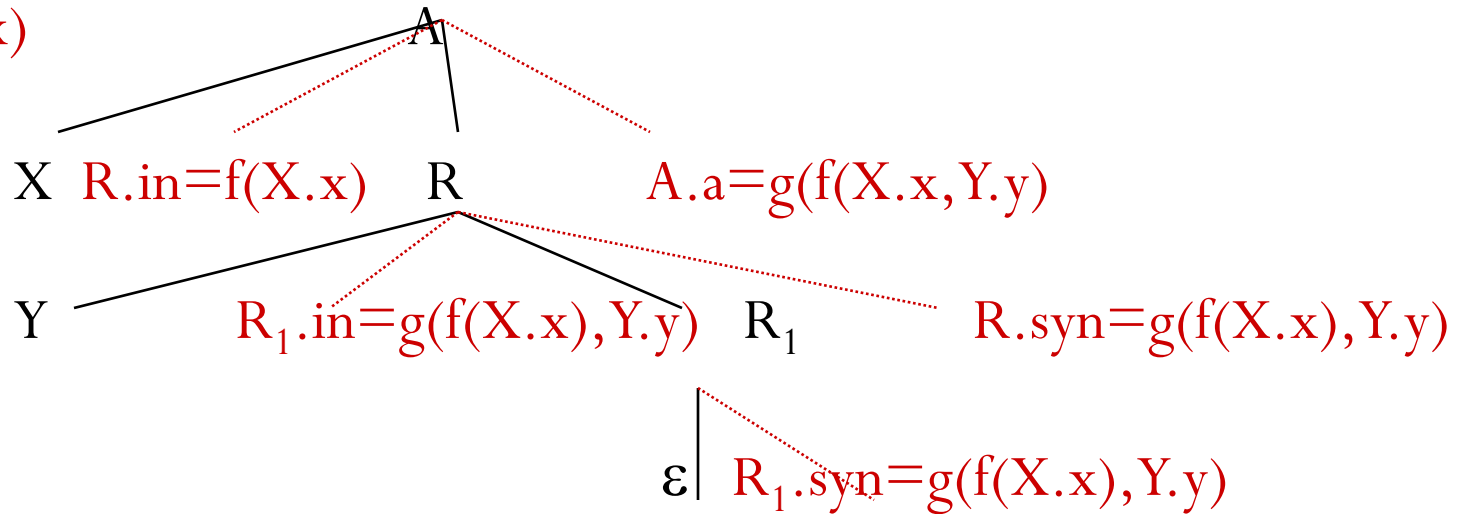
parse tree of left recursive grammar



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

parse tree of non-left-recursive grammar

$X \quad X.x = f(X.x)$



Bottom-Up Evaluation of Inherited Attributes

- Top-down translation scheme: *Any L-attributed definition can be implemented (based on a LL(1) grammar)*
- Bottom-up translation scheme: *Any L-attributed definition can be implemented based on a LL(1) grammar (LL(1) grammar equivalent to LR(1) grammar)*
- Some of L-attributed definitions based on LR(1) grammars (not all of them) can be implemented using the bottom-up translation scheme

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 to put inherited attributes?
- *Solution*:
 - Convert grammar to an equivalent grammar (to ensure)
 - Move all embedding semantic actions in SDT to the end of the production rules
 - Introduce new non-terminals
 - Copy all inherited attributes into the synthesized attributes (most of the time synthesized attributes of new non-terminals)
 - Thus, all semantic actions evaluated during reductions, and we find a place to store an inherited attribute

Removing Embedding Semantic Actions

Transformation of 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 the semantic action S_i into the end of a new production rule $M_i \rightarrow \varepsilon$ for that non-terminal M_i
3. Semantic action S_i will be evaluated when the new production rule is reduced
4. Evaluation order of the semantic rules are not changed by this transformation

Removing Embedding Semantic Actions

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



remove embedding semantic actions

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

$$M_1 \rightarrow \epsilon \{S_1\}$$

$$M_2 \rightarrow \epsilon \{S_2\}$$

.

.

$$M_n \rightarrow \epsilon \{S_n\}$$

Removing Embedding Semantic Actions

$E \rightarrow T R$

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

$R \rightarrow \varepsilon$

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

\Downarrow remove embedding semantic actions

$E \rightarrow T R$

$R \rightarrow + T M R_1$

$R \rightarrow \varepsilon$

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

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

Why Markers Work?

- For LL grammar, marker non-terminals can be added at any position in the body (resulting grammar is LR)
- LL grammar: string w can be derived from A that starts with $A \rightarrow y$, by seeing only the first symbol of w
- Parsing w bottom-up:
 - prefix of w must be reduced to y and then to S
 - It is known as soon as the beginning of w appears in the i/p
- If we insert markers anywhere in y then LR states will incorporate the fact that markers have to be there
 - Reduce null string to marker at the appropriate point in the i/p

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
 - replace the 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)

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

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

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

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

Translation with Inherited Attributes

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

Actual Translation Scheme

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

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

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

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

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

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

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

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

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

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

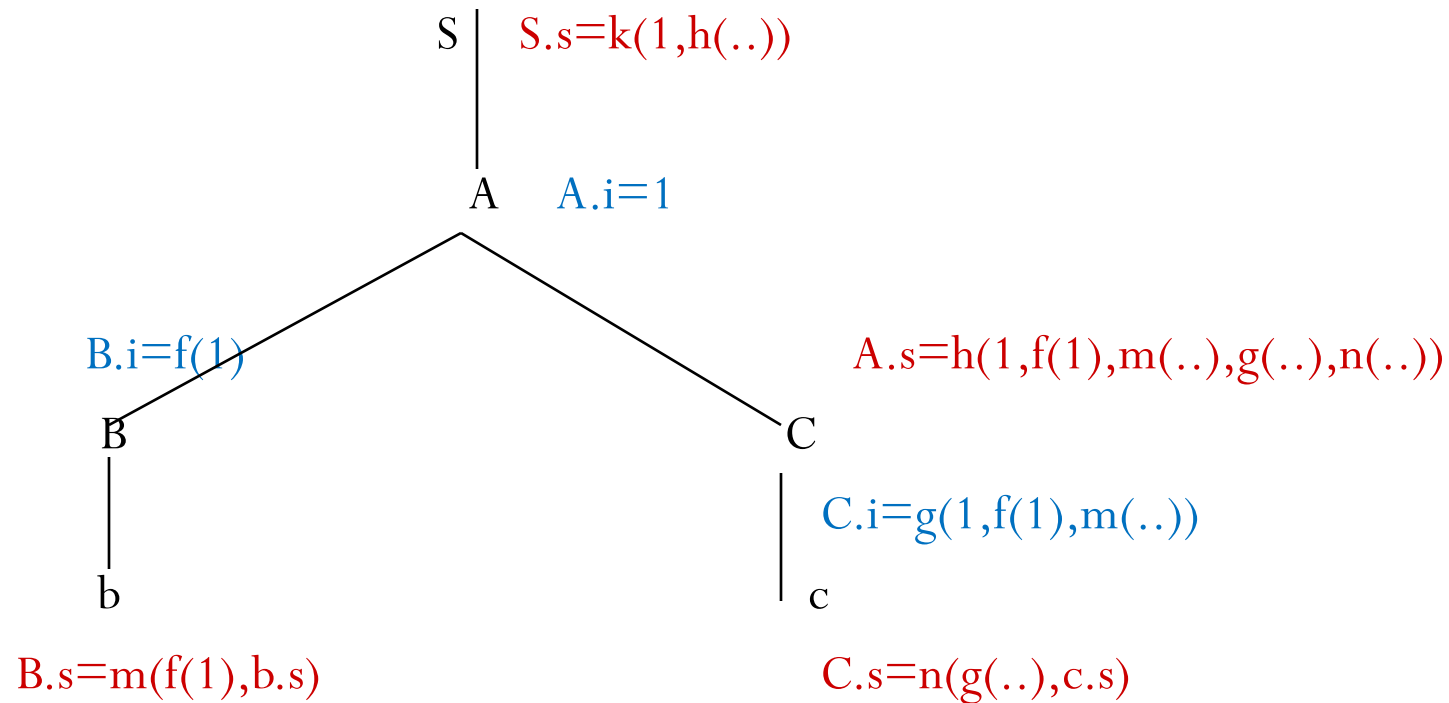
$$M_2 \rightarrow \epsilon \quad \{ s[ntop]=f(s[top]) \}$$

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

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

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

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(m(..),c.s)
$M_1 A$	\$	1 h(f(1),m(..),g(..),n(..))
S	\$	k(1,h(..))

Problems

- All L-attributed definitions based on LR grammars cannot be evaluated during bottom-up parsing

$S \rightarrow \{ L.i=0 \} L$

➔ this translations scheme cannot be implemented

$L \rightarrow \{ L_1.i=L.i+1 \} L_1 1$

during the bottom-up parsing

$L \rightarrow \epsilon \{ \text{print}(L.i) \}$

$S \rightarrow M_1 L$

$L \rightarrow M_2 L_1 1$

➔ Since $L \rightarrow \epsilon$ will be reduced first by the bottom-up parser, the translator cannot know the number of 1s.

$L \rightarrow \epsilon \{ \text{print}(s[\text{top}]) \}$

$M_1 \rightarrow \epsilon \{ s[\text{ntop}]=0 \}$

$M_2 \rightarrow \epsilon \{ s[\text{ntop}]=s[\text{top}]+1 \}$

Problems

- The modified grammar cannot be LR grammar anymore

$L \rightarrow \{ \} L b$

$L \rightarrow a$



$L \rightarrow M L b$

$L \rightarrow a$

$M \rightarrow \varepsilon$

NOT LR-grammar

$S' \rightarrow \cdot L, \$$

$L \rightarrow \cdot M L b, \$$

$L \rightarrow \cdot a, \$$

$M \rightarrow \cdot, a \rightarrow \text{shift/reduce conflict}$

$\text{ACTION}[I0, a] = \text{Shift } [L \rightarrow \cdot a, \$]$

$\text{ACTION}[I0, a] = \text{Reduce by } M \rightarrow \varepsilon [L \rightarrow \cdot, a]$