Syntax Directed Translation

Reading: Section 5.1 – 5.6 of chapter 5

To translate a programming language construct, compiler needs to keep track of many quantities to the grammar symbol.

Syntax Directed Translation

Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.

Values of these attributes are evaluated by the **semantic rules** associated with the production rules.

Evaluation of these 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, data types, intermediate representation etc.

Syntax Directed Translation

When we associate semantic rules with productions, we use two notations:

- **Syntax-Directed Definitions**
- **Translation Schemes**

Syntax-Directed Definitions are high level specifications for translations. They hide many implementation details and free the user from having to explicitly specify the order in which translation takes place.

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

Syntax-Directed Definitions

A syntax-directed definition is a generalization of a context-free grammar in which each grammar symbol is associated with a set of attributes. This set of attributes for a grammar symbol is partitioned into two subsets **synthesized** and **inherited** attributes of that grammar

Semantic rules set up dependencies between attributes which can be represented by a *dependency graph*.

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.

Syntax-Directed Definitions

Given a production

$$A \rightarrow \alpha$$

then each semantic rule is of the form

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

where f is a function and c_i are attributes of A and α, and either

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

Syntax-Directed Definition

Example

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

Note: all attributes in this example are of the synthesized type

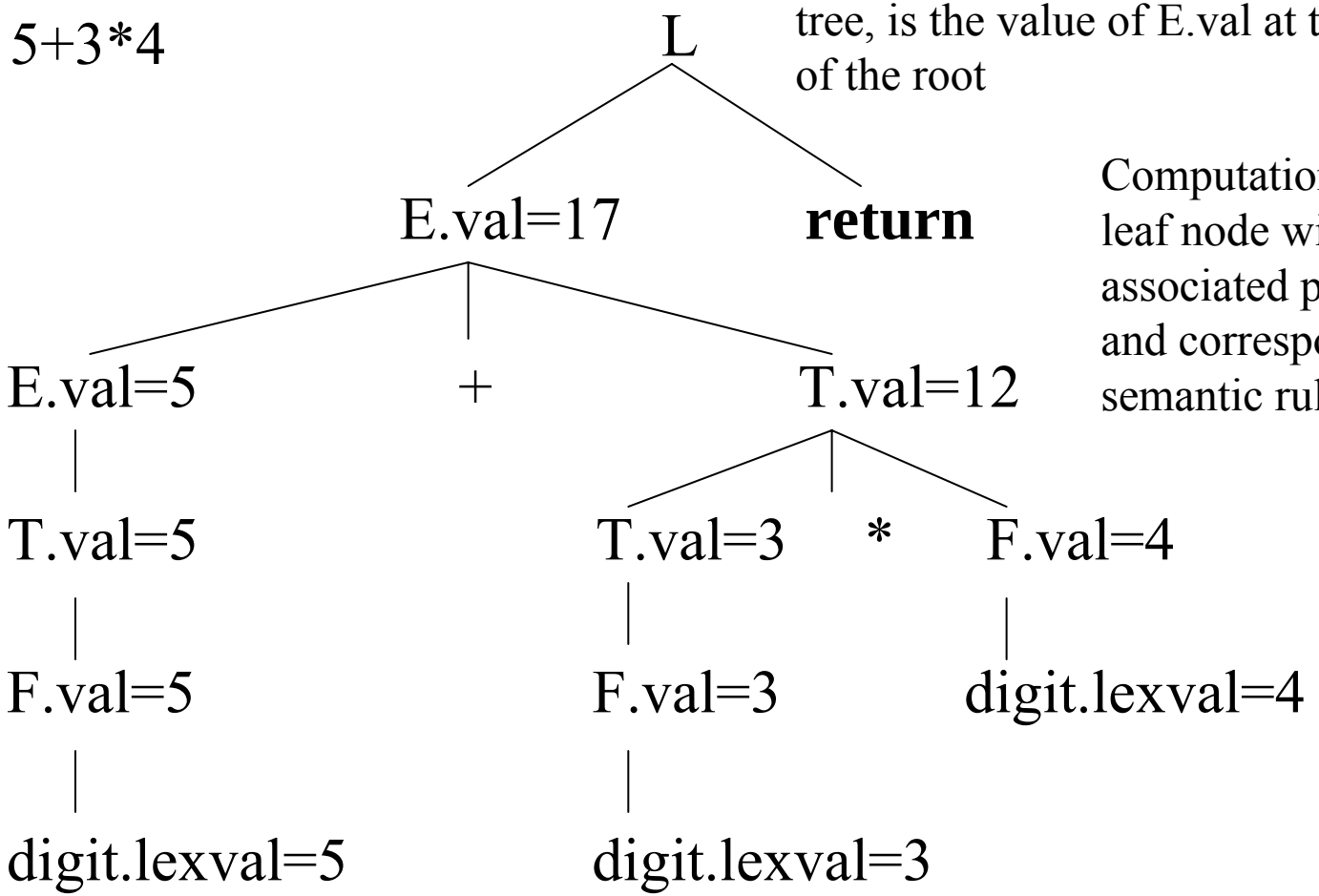
Symbols E, T, and F are associated with a synthesized attribute *val*.

The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

Annotated Parse Tree

Example

Input: 5+3*4



Output: The value, printed at the root of tree, is the value of E.val at the first child of the root

Computation start from leaf node with associated production and corresponding semantic rule

Inherited Attributes

An inherited attribute at any node is defined based on the attributes at the parent and/or siblings of the node.

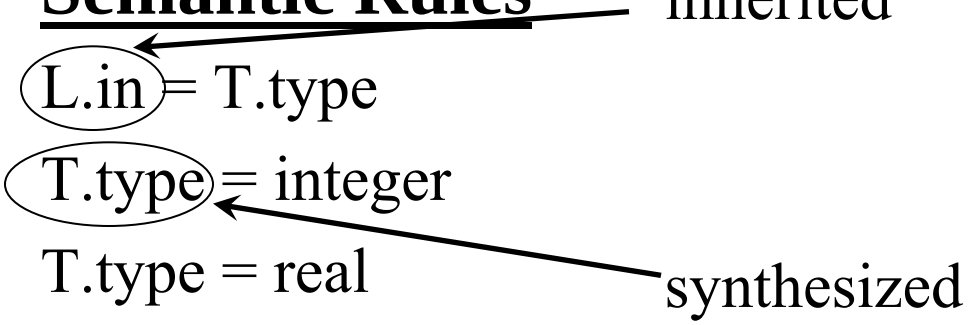
Useful for describing context sensitive behavior of grammar symbols.

For example, an inherited attribute can be used to keep track of whether an identifier appears at the left or right side of an assignment operator. This can be used to decide whether the address or the value of the identifier needed.

Inherited Attributes

Example

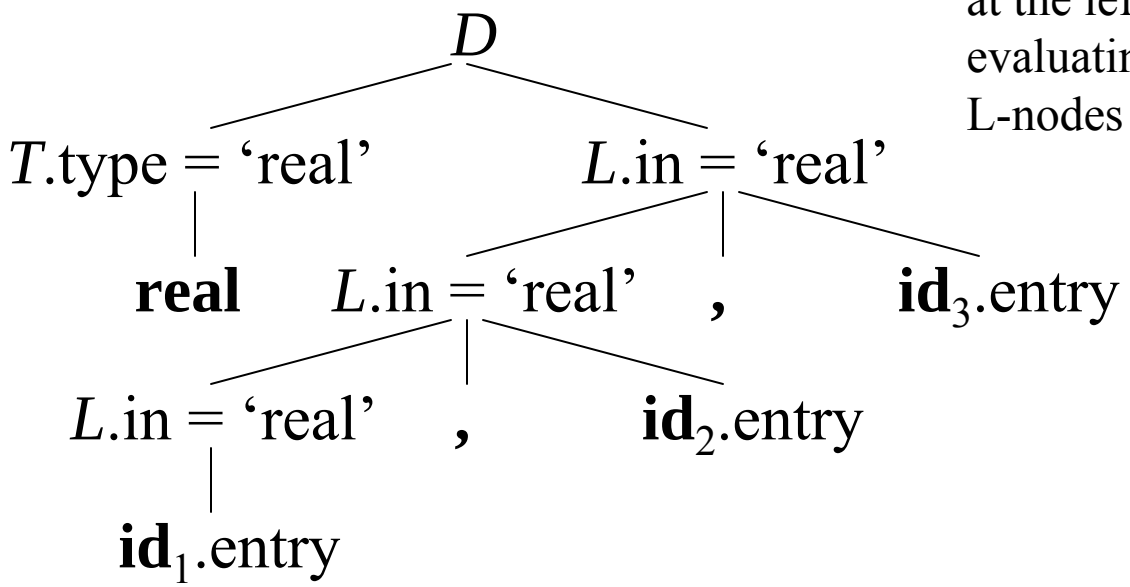
Production	**Semantic Rules**
D → T L	L.in = T.type
T → **int**	T.type = integer
T → **real**	T.type = real
L → L_1 **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
L → **id**	addtype(**id**.entry,L.in)



Inherited Attributes

Parse Tree

Input : real id1 , id2 , id3



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

Dependency Graph

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

Algorithm for dependency graph construction

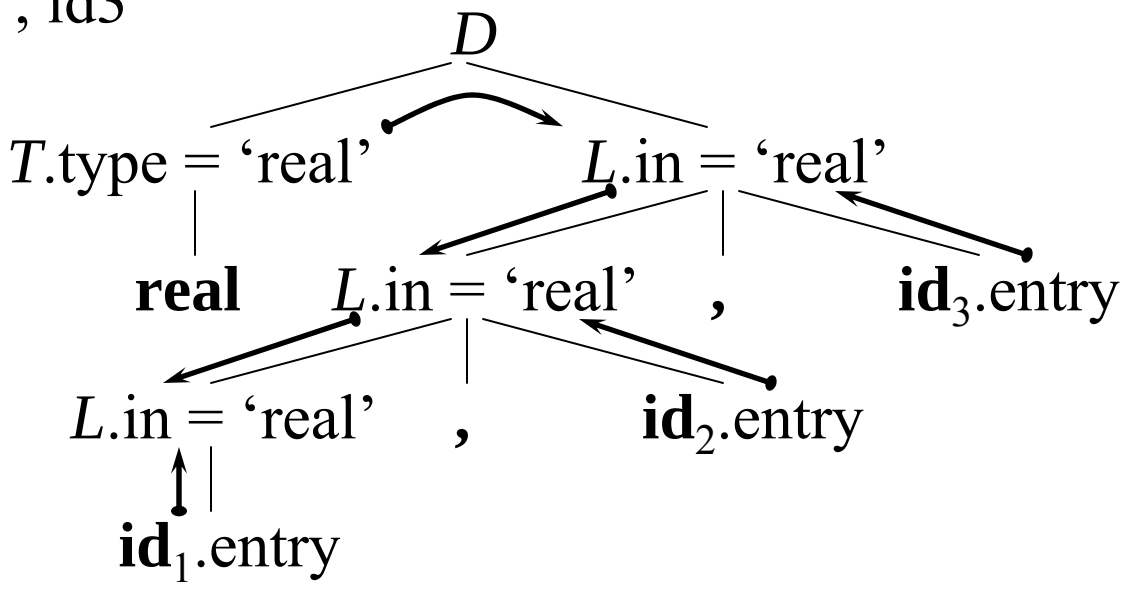
for each node **n** in the parse tree do
  for each attribute **a** of the grammar symbol at **n** do
    Construct a node in the dependency graph for **a**
for each node **n** in the parse tree do
  for each semantic rule b = f(c1, c2, …, ck) associated with the production used at **n** do
    for i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

Dependency Graph

Grammar in page 9

For Parse Tree of Page 10

Input : real id1 , id2 , id3



S-Attributed Definitions

A syntax-directed definition that uses synthesized attributes exclusively is called an *S-attributed definition* (or *S-attributed grammar*)

A parse tree of an S-attributed definition is annotated by evaluating the semantic rules for the attribute at each node bottom-up

Yacc/Bison only support S-attributed definitions

Bison Example

```
%token DIGIT
%%
L : E '\n'          { printf("%d\n", $1); }
  ;
E : E '+' T         { $$ = $1 + $3; }
  | T               { $$ = $1; }
  ;
T : T '*' F         { $$ = $1 * $3; }
  | F               { $$ = $1; }
  ;
F : '(' E ')'       { $$ = $2; }
  | DIGIT           { $$ = $1; }
  ;
%%
```

Synthesized attribute of parent node **F**

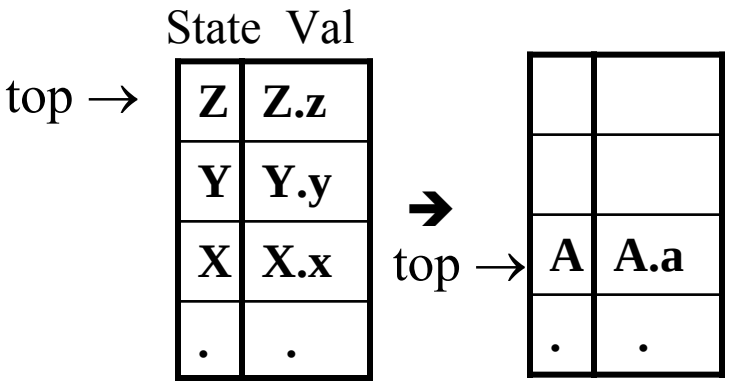
Bottom-up Evaluation of S-Attributed Definitions

An S-attributed grammar can be translated bottom-up as and when the grammar is being parsed using an LR parser.

The LR parser's stack can be extended to hold not only a grammar symbol, but also its semantic attribute as well.

Hence *action[s,a] = shift s'* becomes *action[s,a] = shift a*.$$, *s'*

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



When an entry of the parser stack holds a grammar symbol X (terminal or non-terminal), the corresponding entry will hold the synthesized attribute(s) of the symbol X.

the values of the attributes are evaluated during reductions.

Bottom-up Evaluation of S-Attributed Definitions in Yacc

Input: **3*5+4n**

Grammar in Page 14

Stack	val	Input	Action	Semantic Rule
$	_	**3*5+4n$**	shift	
$ 3	*3*	***5+4n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *F*	*3*	***5+4n$**	reduce $T \rightarrow F$	$$ = $1
$ *T*	*3*	***5+4n$**	shift	
$ *T* *****	*3* _	**5+4n$**	shift	
$ *T* *** 5**	*3* _ *5*	**+4n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *T* ***** *F*	*3* _ *5*	**+4n$**	reduce $T \rightarrow T$ ***** F	$$ = $1 * $3
$ *T*	*15*	**+4n$**	reduce $E \rightarrow T$	$$ = $1
$ *E*	*15*	**+4n$**	shift	
$ *E* +	*15* _	**4n$**	shift	
$ *E* + **4**	*15* _ *4*	**n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *E* + *F*	*15* _ *4*	**n$**	reduce $T \rightarrow F$	$$ = $1
$ *E* + *T*	*15* _ *4*	**n$**	reduce $E \rightarrow E + T$	$$ = $1 + $3
$ *E*	*19*	**n$**	shift	
$ *E* **n**	*19* _	**$**	reduce $L \rightarrow E$ **n**	*print* $1
$ *L*	*19*	**$**	accept	

L-Attributed Definitions

An L-attribute is an inherited attribute which can be evaluated in a left-to-right fashion. L-attributed definitions can be evaluated using a *depth-first* evaluation order. (L –> attributes flow from left to right)

More precisely, a syntax-directed definition is *L-attributed* if each inherited attribute of X_j on the right side of $A \rightarrow X_1 X_2 \ldots X_n$ depends only on

1. the attributes of the symbols $X_1, X_2, \ldots, X_{j-1}$ to the left of X_j in the production and
2. the inherited attributes of A

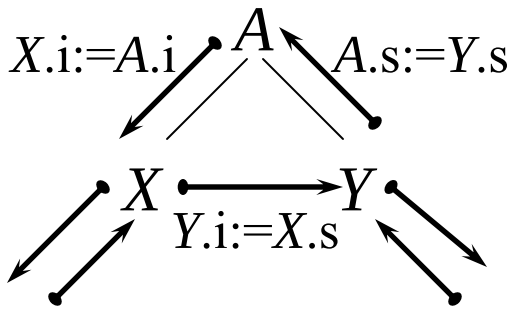
Every S-attributed definition is L-attributed, the restrictions only apply to the inherited attributes (not to synthesized attributes).

Evaluation of L-Attributed Definitions

L-attributed definitions allow for a natural order of evaluating attributes: depth-first and left to right

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

$A \rightarrow X\ Y$



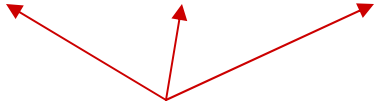
$X.i := A.i$
$Y.i := X.s$
$A.s := Y.s$

Translation Schemes

A **translation scheme** is a context-free grammar in which:

- attributes are associated with the grammar symbols and
- semantic actions enclosed between braces {} are inserted within the right sides of productions.

Ex: $A \rightarrow \{ \ldots \}\ X\ \{ \ldots \}\ Y\ \{ \ldots \}$



Semantic Actions

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.

A Translation Scheme

Example

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

$E \rightarrow T\ R$

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

$R \rightarrow \varepsilon$

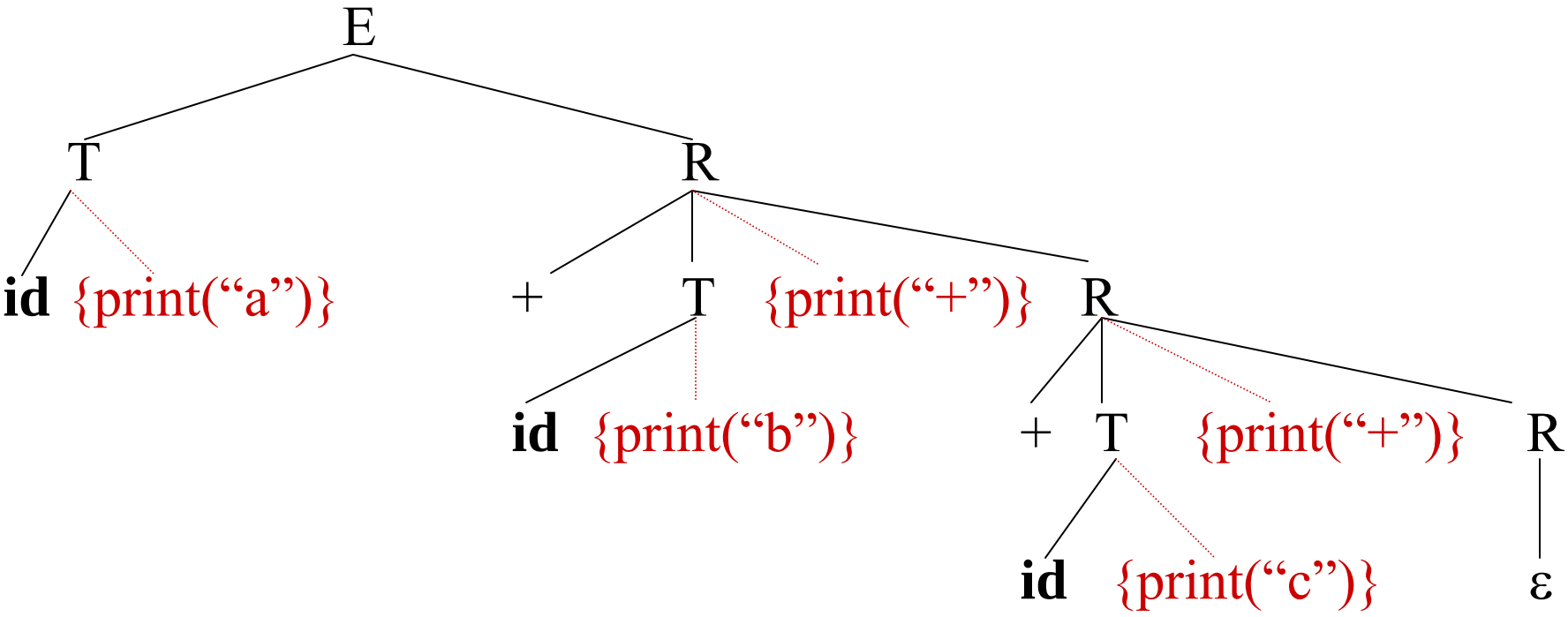
$T \rightarrow$ **id** { print(**id**.name) }

a+b+c ➔ ab+c+

infix expression postfix expression

A Translation Scheme

Example



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

Translation Schemes Requirements

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

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

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).

Top-Down Translation

L-attributed definitions can be evaluated in a top-down fashion (predictive parsing) with translation schemes. The algorithm for elimination of left recursion is extended to evaluate action and attribute.

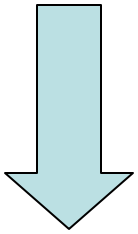
Attributes in L-attributed definitions implemented in translation schemes are passed as arguments to procedures (synthesized) or returned (inherited)

Eliminating Left Recursion from a Translation Scheme

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

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

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



eliminate left recursion

inherited attribute of the new non-terminal

synthesized attribute of the new non-terminal

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

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

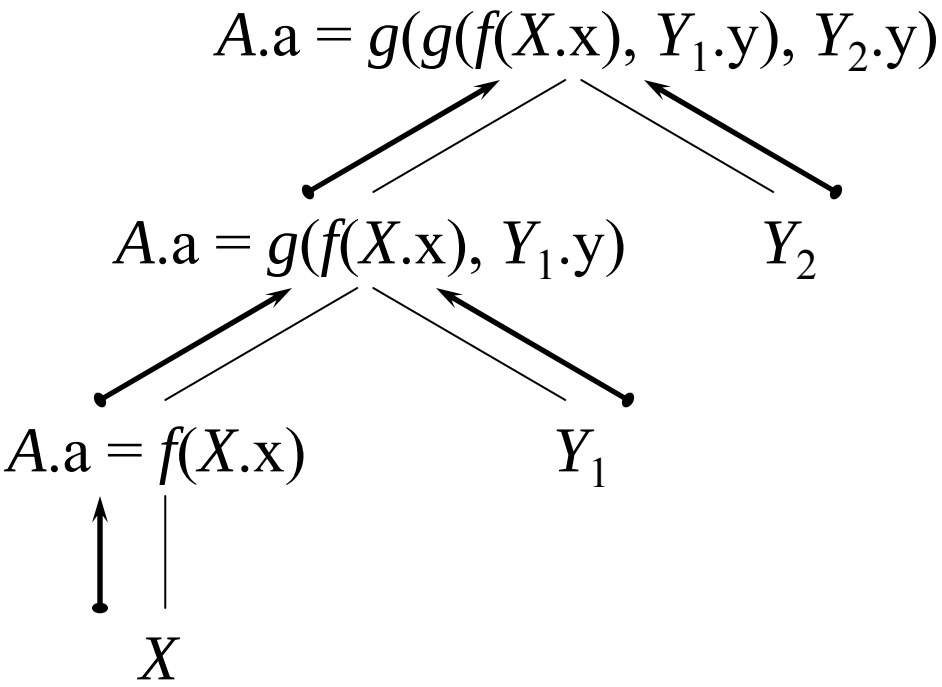
$R \rightarrow \varepsilon\ \{ R.syn = R.in \}$

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

Eliminating Left Recursion from a Translation Scheme

Evaluating attributes

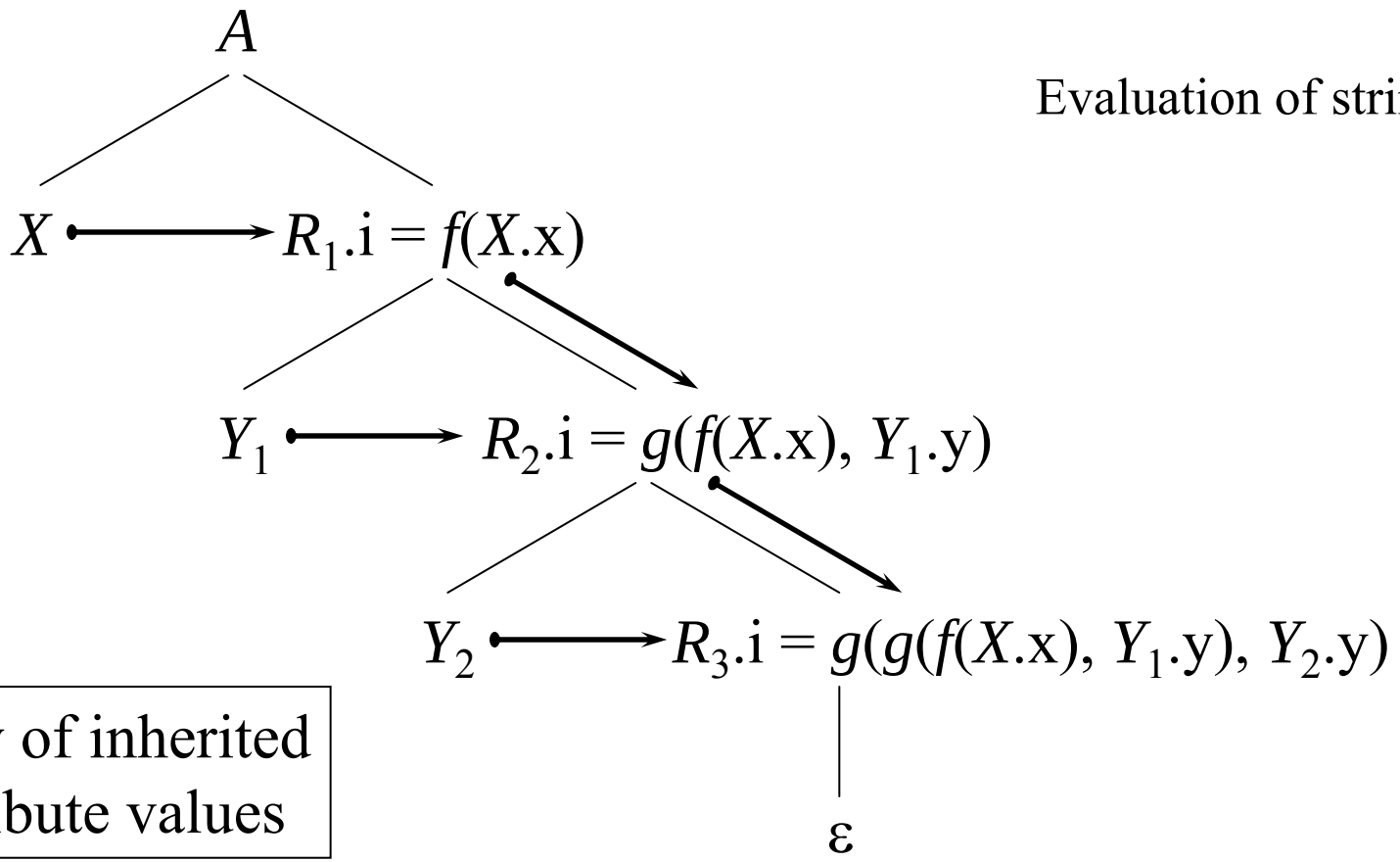
Evaluation of string XYY



The values are computed according to a left recursive grammar

Eliminating Left Recursion from a Translation Scheme

Evaluating attributes



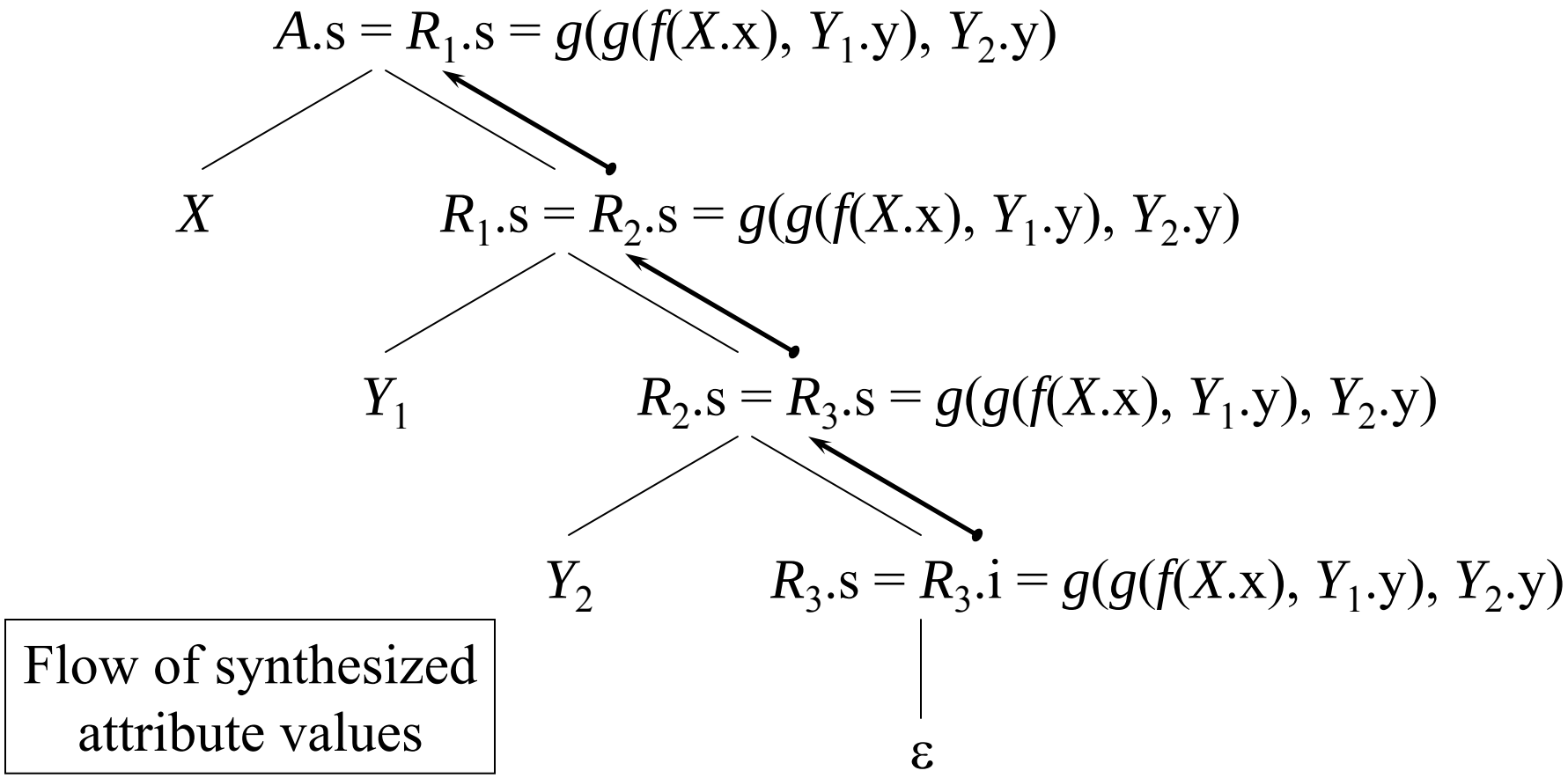
Evaluation of string XYY

Flow of inherited attribute values

Eliminating Left Recursion from a Translation Scheme

Evaluating attributes

Evaluation of string XYY



Flow of synthesized attribute values

Bottom-Up Evaluation of Inherited Attributes

While evaluating S-attributed definitions during bottom-up parsing is straight forward, evaluating inherited attributes is tricky. L-attributed definitions are a simple subclass of inherited attributes that can be effectively implemented in most bottom-up parsers.

The method used in Bottom-up Evaluation of S-Attributed Definitions is extended with conditions:

- All embedding semantic actions in translation scheme should be in the end of the production rules.
- All inherited attributes should be copied into the synthesized attributes (most of the time synthesized attributes of new non-terminals).
- Evaluate all semantic actions during reductions

Bottom-Up Evaluation of Inherited Attributes

Removing Embedding Semantic Actions

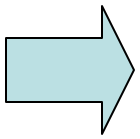
Insert marker nonterminals to remove embedded actions from translation schemes, that is $A \rightarrow X$ { actions } Y is rewritten with marker nonterminal N into

$A \rightarrow X\ N\ Y$

$N \rightarrow \varepsilon$ { actions }

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

E → T R
R → + T { print("+") } R1
R → ε
T → **id** { print(**id**.name) }



E → T R
R → + T M R1
R → ε
T → **id** { print(**id**.name) }
M → ε { print("+") }

Bottom-Up Evaluation of Inherited Attributes

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

Grammar.

E → T {L.type = T.val} L
L → L, id {settype(id.entry, L.type)}
L → id {settype(id.entry, L.type)}
T → int{T.val = integer}|
float{T.val = float}

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

Input	*Stack*
float id, id$	$
id, id$	$float
id, id$	$T {T.val = float}
, id$	$T id
, id$	$T L {L:type = float}
id$	$T L,
$	$T L, id
$	$T L {L.type = T.val}
$	$E

With L-attributed definitions, the required value is always below in the stack.

Exercise

1. For the following grammar:

   $E \rightarrow E + E \mid E * E \mid (E) \mid id$

   Annotate the grammar with syntax directed definitions using synthesized attributes. Remove left recursion from the grammar and rewrite the attributes correspondingly.

2. For the expression 2 * (3 + 5), construct an annotated parse tree using the modified grammar (that has no left recursion) from the previous question. Trace the path in which semantic attributes are evaluated.

3. Questions 5.1, 5.2, 5.4, 5.6, 5.8, 5.11, 5.12 & 5.14

Assignments

1. Write a bison program that can evaluate prefix expressions with arbitrary arity. This means, an expression is an operator followed by any number of expressions (not just two).

2. Write a bison program that converts a prefix expression into its corresponding infix expression.

3. In an infix expression of the form a = b + c, the type of a is int only if both b and c are of type int. Even if either b or c is float, then the type of a should be set to float. Write a grammar with syntax directed definitions to implement the above requirements.