Abstract Syntax

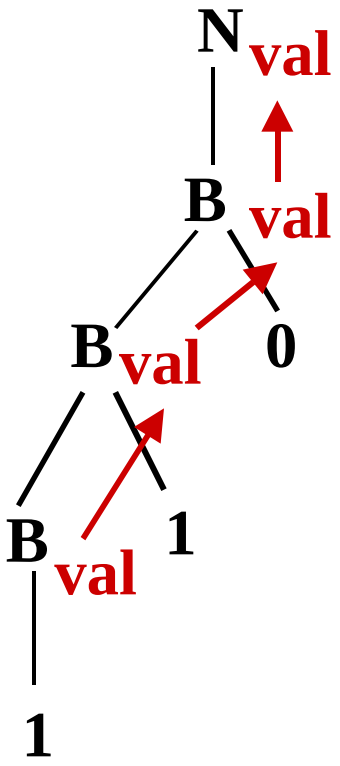
- ***Attribute grammars*** **- a methodology to check semantics during parsing**
  - **Inherited and synthesized attributes**
  - **Evaluation with dependency graph**
  - **S-attributed grammars**
  - **Interleaving evaluation with parsing**

Attribute Grammars

- **Each nonterminal has associated properties or *attributes***
  - **Attributes can be assigned values by *semantic actions* associated with each production**
  - **Success of the parse can be based, in part, on attribute values**
- **Examples -**
  - **Setting the type of identifiers in a declaration statement;**
  - **Calculating the integer value of a string of digits**

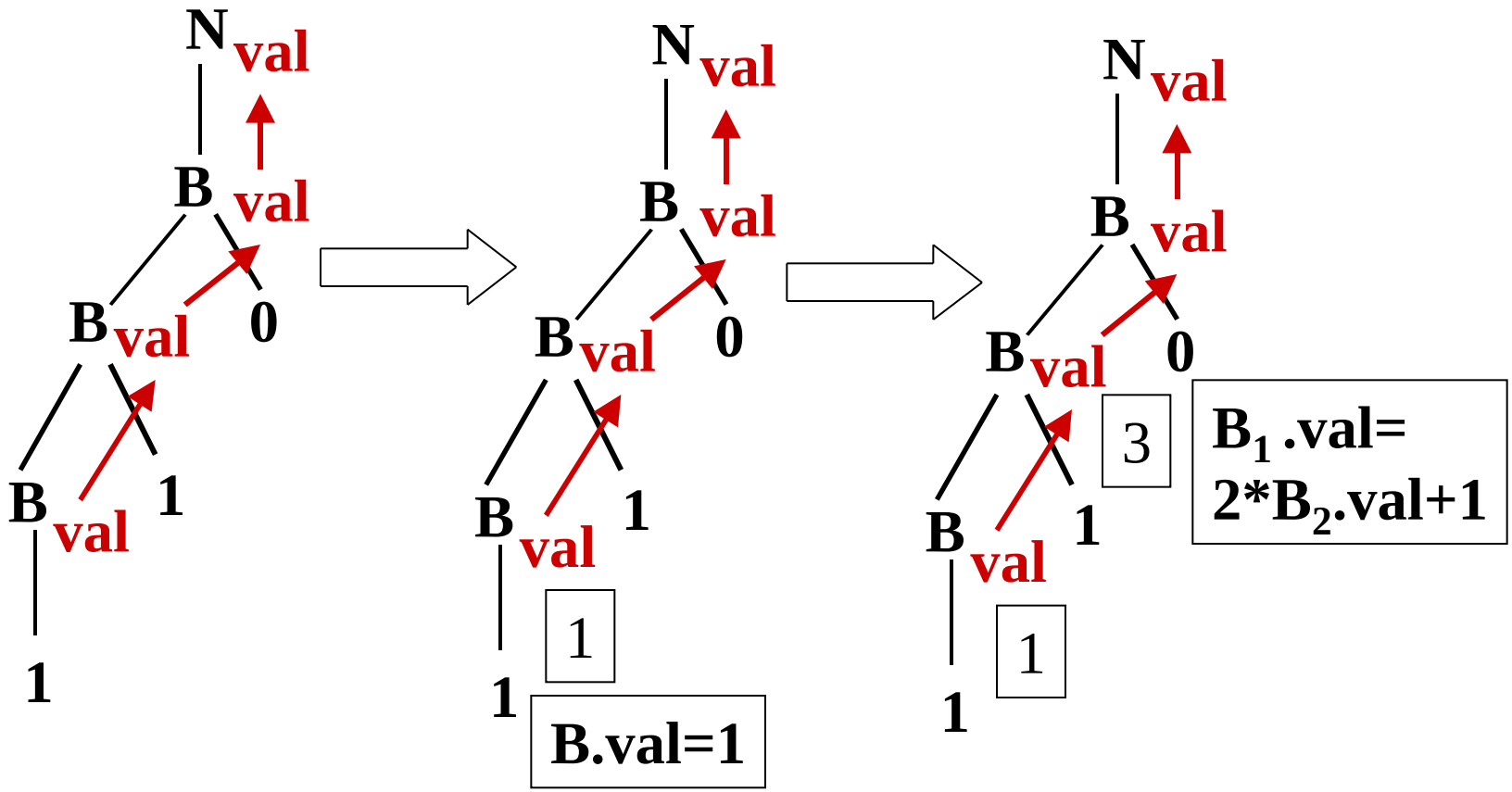
Example

nonterminals: N, B	*attributes:* B.val, N.val
productions	***semantic actions***
$B \rightarrow 1$	B.val = 1
$B \rightarrow 0$	B.val = 0
$B_1 \rightarrow B_2\ 1$	B_1.val = 2*B_2.val + 1
$B_1 \rightarrow B_2\ 0$	B_1.val = 2*B_2.val
$N \rightarrow B$	N.val = B.val

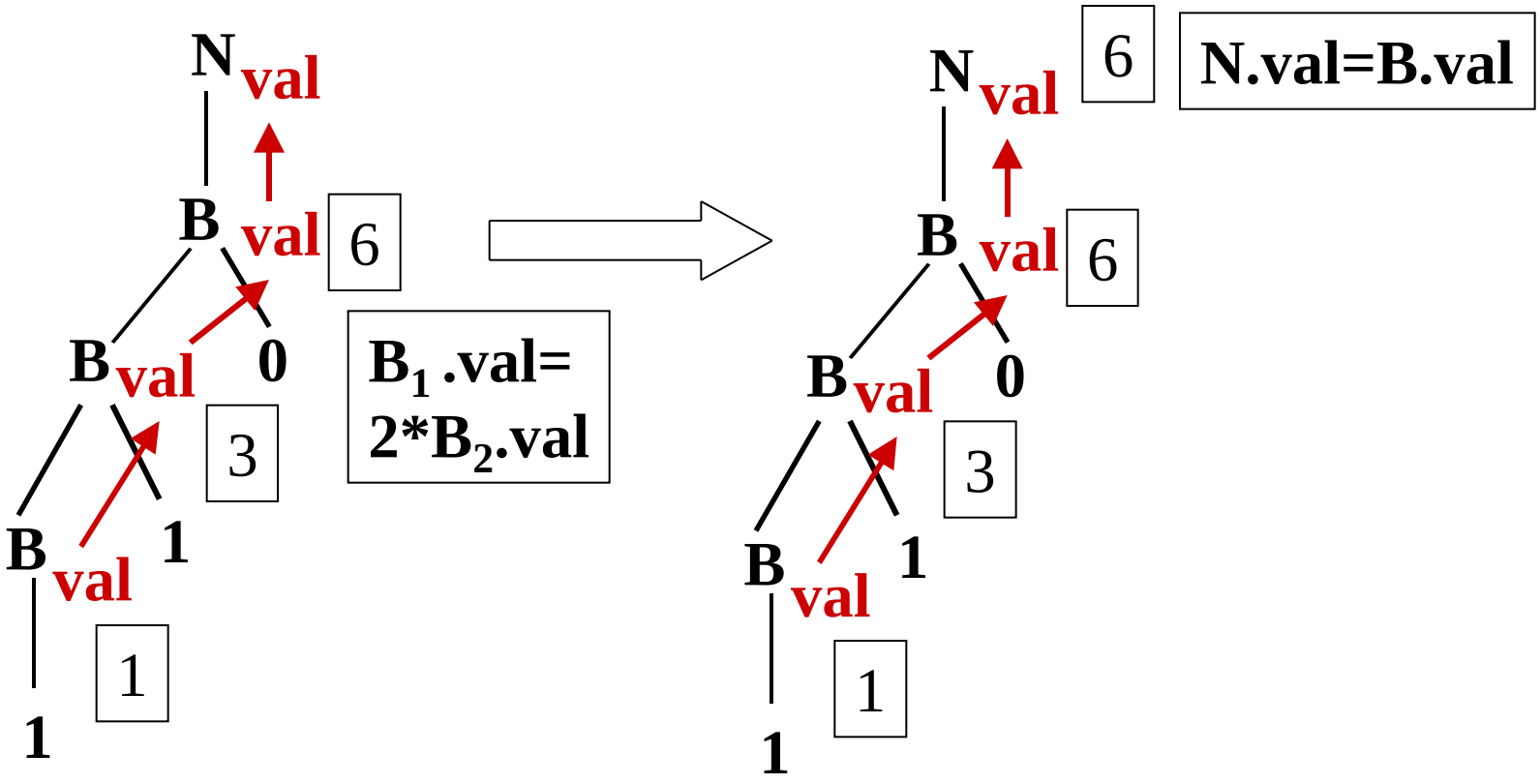


Attributes can only depend on attributes of their parent nonterminal or their children nonterminals in the parse tree

Example

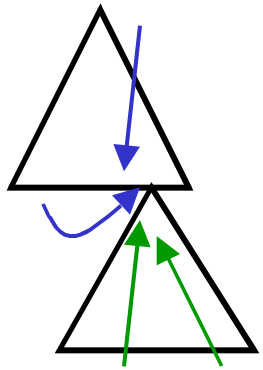

N
val
B
val
B
val
0
B
val
1
1
N
val
B
val
B
val
0
B
val
1
1
1
B.val=1
N
val
B
val
B
val
0
B
val
1
1
3
1
1
B1 .val=
2*B2.val+1

Example



Classifications

- ***Inherited* attributes:**
  - **Values based on attributes of parent (lefthand-side nonterminal) or siblings (nonterminals on righthand-side of same production)**
- ***Synthesized* attributes:**
  - **Values based on attributes of descendents (child nonterminals in same production)**



Classifications

- **Evaluation context: always within focus of a single production**
  - **Dependence edges go only one level up or down or across in parse tree**
- **Terminals can be associated with values returned by the scanner**
- **Distinguished nonterminal cannot have inherited attributes**

Example

Identifiers with no letters repeated

(e.g., moon - illegal, money - legal)

str(I) is same as I.str

D → I *str*(I) = {}; *val*(D) = *val*(I);
accept, if *val*(D) != *error*

I_1 → L I_2 *str*(L) = *str*(I_1); *str*(I_2) = *val*(L);
val(I_1) = *val*(I_2)

I → L *str*(L) = *str*(I); *val*(I) = *val*(L)

L → a | b | … | z *val*(L) = concatenation of *val* returned by scanner to *str*(L), if this character is not a repeated letter, else *error*.

(note: any comparison to *error* returns *error*.)

Inherited Attributes

D → I *str*(I) = {}; *val*(D) = *val*(I);
accept, if *val*(D) != *error*

I_1 → L I_2 *str*(L) = *str*(I_1); *str*(I_2) = *val*(L);
val(I_1) = *val*(I_2)

I → L *str*(L) = *str*(I); *val*(I) = *val*(L)

L → a | b | … | z *val*(L) = concatenation of *val* returned by scanner to *str*(L), if this character is not a repeated letter, else *error*.

(note: any comparison to *error* returns *error*.)

Synthesized Attributes

D → I *str*(I) = {}; *val*(D) = *val*(I);

accept, if *val*(D) != *error*

I_1 → L I_2 *str*(L) = *str*(I_1); *str*(I_2) = *val*(L);

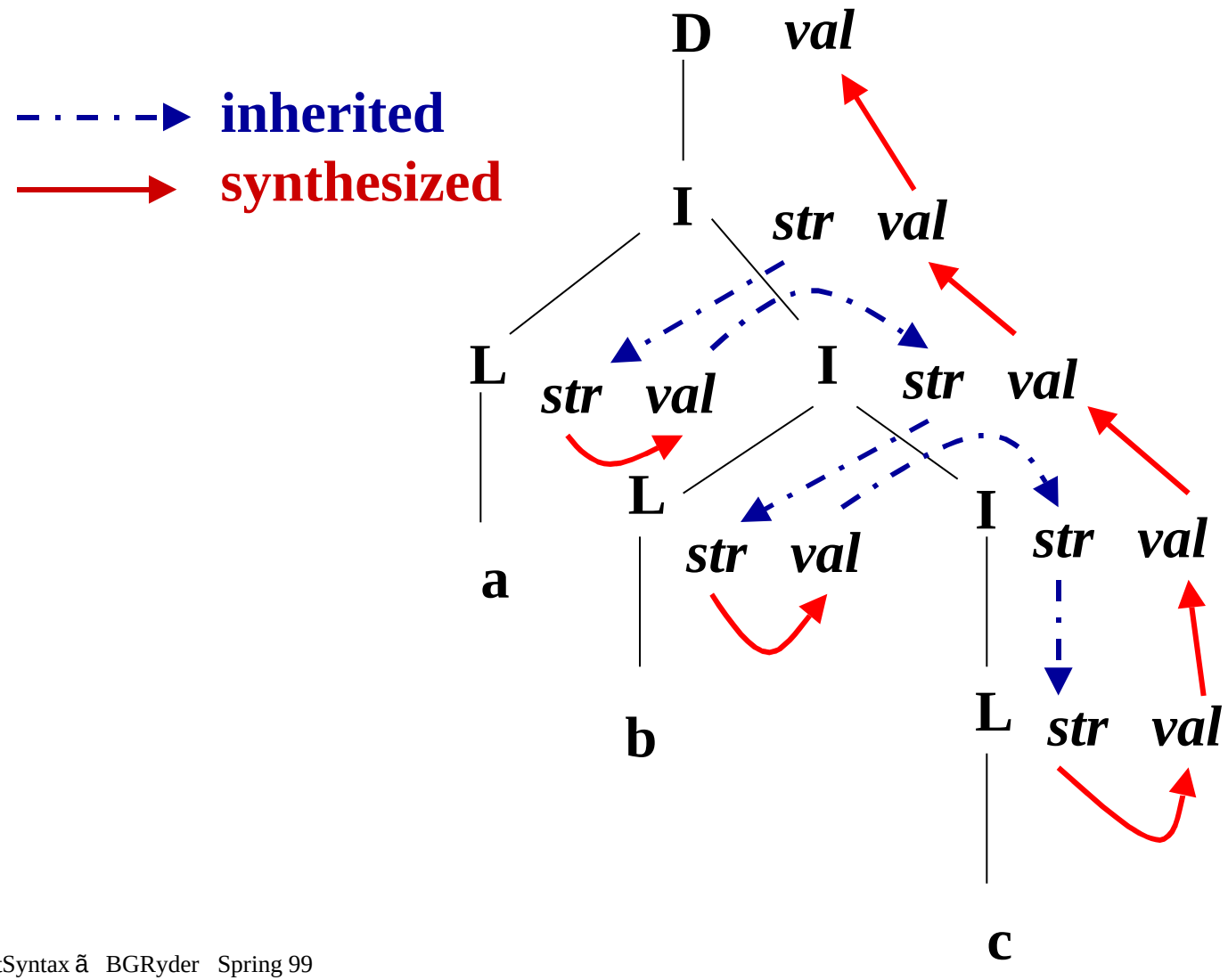
val(I_1) = *val*(I_2)

I → L *str*(L) = *str*(I); *val*(I) = *val*(L)

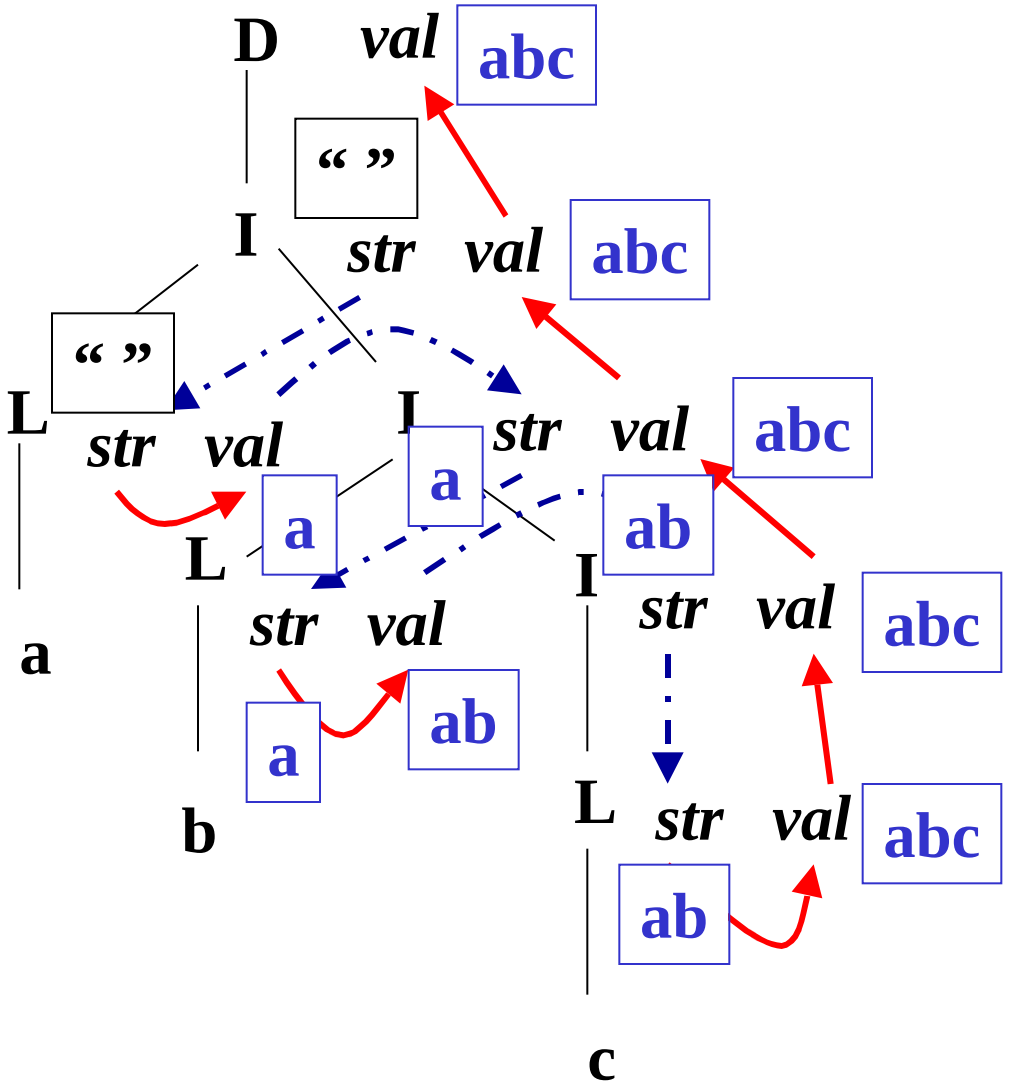
L → a | b | … | z *val*(L) = concatenation of *val* returned by scanner to *str*(L), if this character is not a repeated letter, else *error*.

(note: any comparison to *error* returns *error*.)

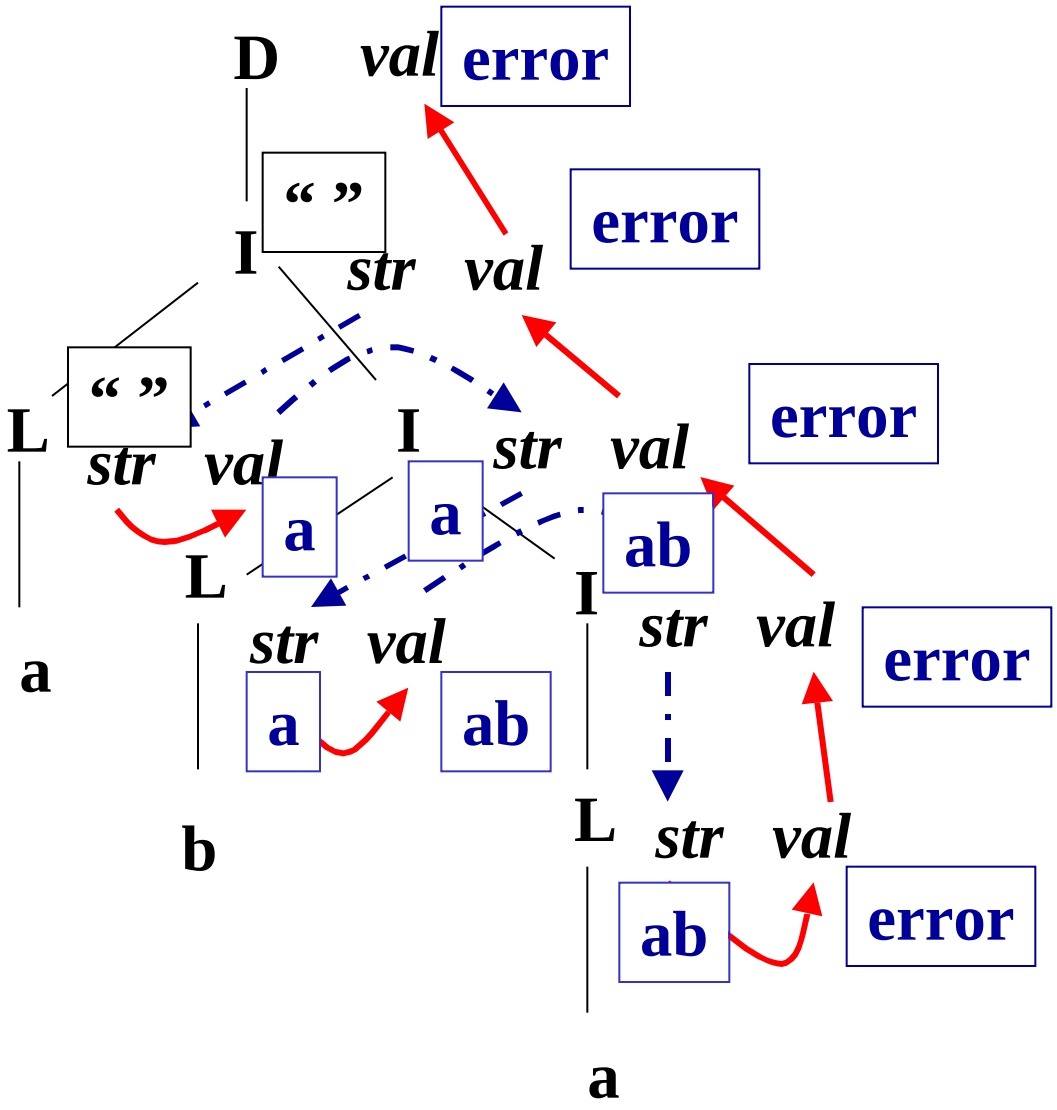
Parse Tree of abc



Decorated Parse Tree

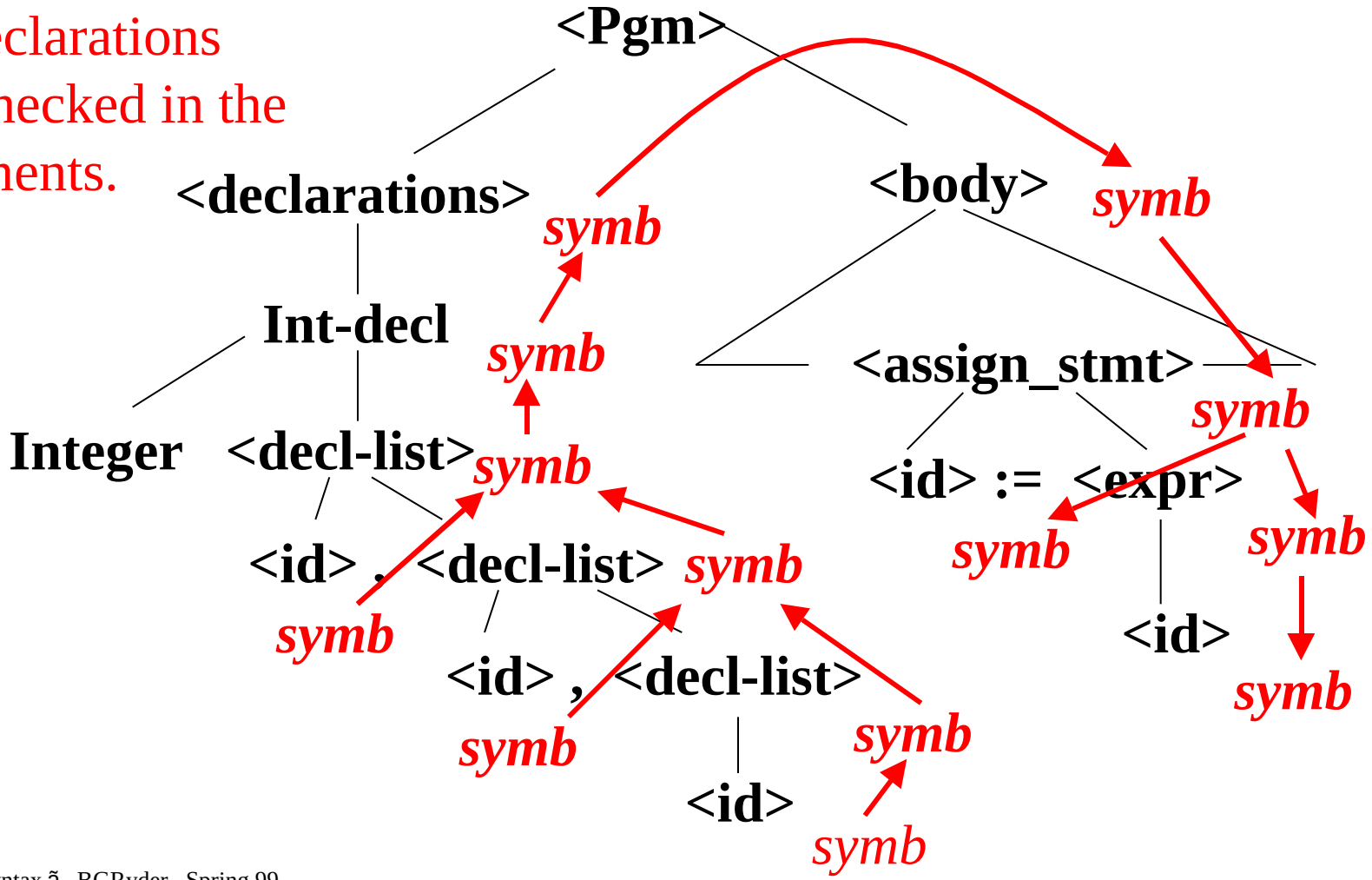


Example - aba


D
val
error
“ ”
I
str
val
error
L
“ ”
str
val
I
str
val
error
a
L
a
a
ab
str
val
I
str
val
error
a
ab
b
L
str
val
ab
error
a

Compiler Example

symb is the symbol table gathered from the declarations and checked in the statements.



S-attributed Grammars

- **S-attributed grammars: all attributes are synthesized**
- **Easy to interleave with shift-reduce parsing by using a parallel stack for attribute values**
  - ***Evaluate attributes when do a reduction***
- **Important: can code semantic functions *a priori,* because know all the handles from the grammar, so know where the associated attributes will be in the semantic value stack when a reduction is about to take place.**

Semantic Value Stack

stacks	input
$	1 + 2 $
$	
$ *int-const*	+ 2 $
$ 1	
$ E	+ 2 $
$ 1	
$ E +	2 $
$ 1 ε	
$ E + *int-const*	$
$ 1 ε 2	
$ E + E	$
$ 1 ε 2	
$ E	$
$ 3	

Attribute Grammars

- **Inherited attributes are not “natural” to use with bottom up parsing**
- **Sometimes can convert a grammar with inherited attributes to an alternative grammar without inherited attributes**
- **Other times can study the grammar and can precompute how to evaluate inherited attributes**

Example - Change Grammar

(ASU, p 315) Pascal declarations

D → L : T

T → *integer* | *character*

L → L , *id* | *id*

Here, as build parse tree for declaration, can't know type T when encountering *id*'s in subtree of L. seems like an inherited attribute from sibling T; PROBLEM

SOLUTION: form another grammr:

D → *id* L

L → , *id* L | : T

T → *integer* | *character*

Here, type can be a synthesized attribute of L and will have it when processing each *id*.

Example - Copy Rules

Another declaration grammar (ASU, p 310)

D → T : L	**L.type = T.type**
T → *int* | *real*	**T.type = *int* | *real***
L_1 → L_2 , *id*	**L_2 .type = L_1 .type; *entersym(id,L_2 .type)***
L → *id*	***entersym(id, L.type)***

L.type is an inherited attribute which is equal to T.type.

Because of the shape of the productions, we know that T.type is always directly beneath L.type in the semantic stack in the 1st production. T.type is 3 beneath top of value stack when 3rd rule is used for a reduction

So sometimes can use semantic stack position (precomputed) to get inherited attribute values passed even in bottom up parsing.

Example

stack	input	semantic stack
$	*int* : x , y $	
$ T	: x , y $	*$ integer*
$ T :	x , y $	*$ integer*
$ T : *id1*	, y $	*$ integer id1.sym*
		entersym(semst[top], semst[top-1])
$ T : L	, y $	*$ integer L.info*
$ T : L ,	y $	*$ integer L.info*
$ T : L , *id2*	$	*$ integer L.info id2.sym*
		entersym(semst[top], semst[top-2])
$ T : L	$	*$ integer L.info*
$ D		*$*

Marker Nonterminals

- **Marker nonterminals have righthandside ε and are added so as to make the semantic stack have the right depth to do the previous handling of inherited attributes**

E.g., **S → a A C** — ***C.in = A.syn***

semst[top] = semst[top-1]

S → b A B C — ***C.in = A.syn***

semst[top] = semst[top-2]

C → c — ***C.syn = g (C.in)***

B → b

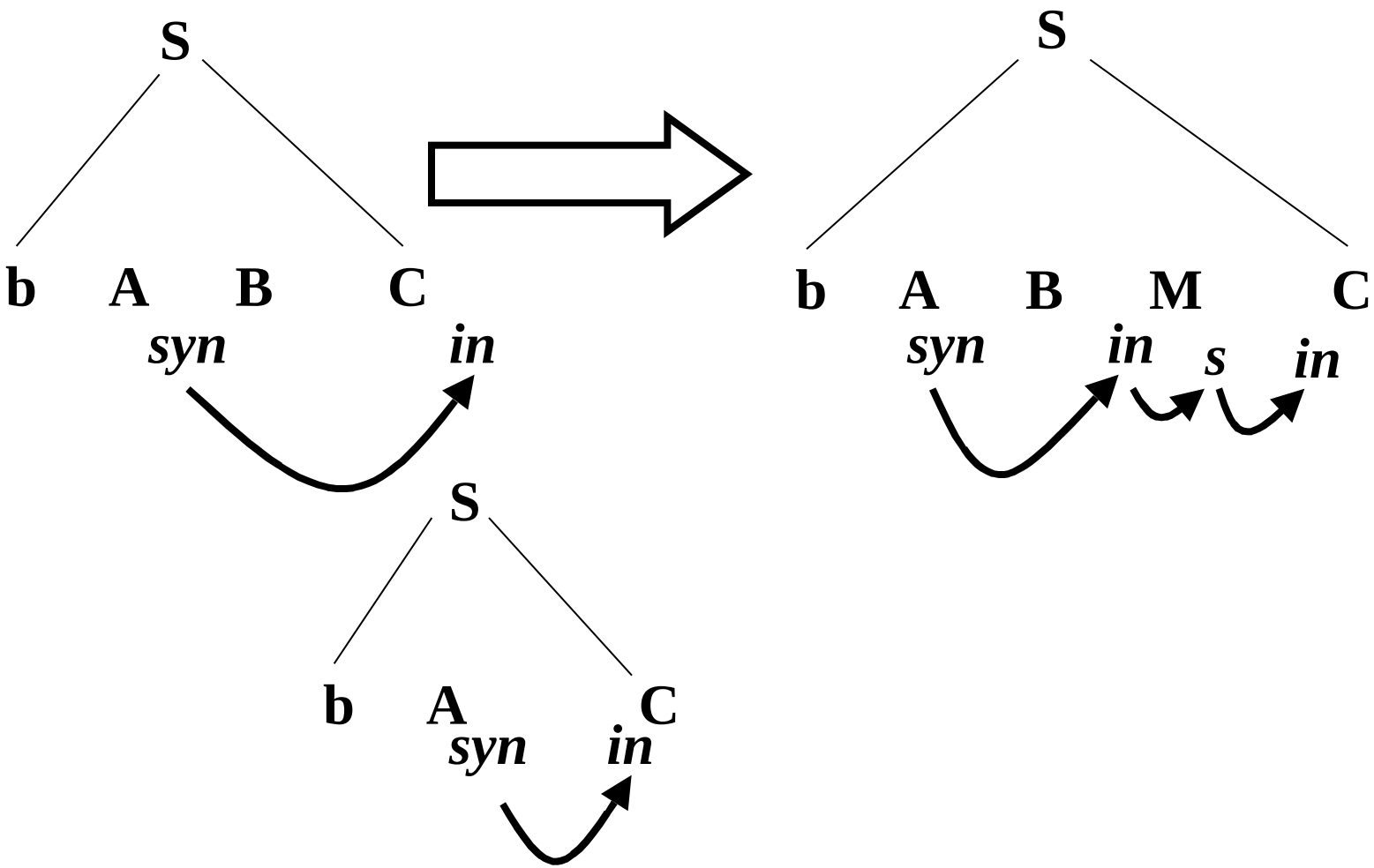
Example

S → a A C	***C.in = A.syn***
S → b A B M C	***M.in = A.syn; C.in = M.s***
C → c	***C.syn = g (C.in)***
B → b	
M → ε	***M.s = M.in***

Here M is used to copy attribute value of *A.syn* to the top of the stack before recognition of C proceeds. Then the value of *C.in* will be in semst[top-1] when 3rd production is used in a reduction.

We have reduced this grammar to the heuristic used in the previous case for inheriting an attribute value from an immediate left sibling.

Example



Abstract Syntax

- **LR parser reductions (and associated semantic actions) are performed in predictable order on the parse tree**
  - **Bottom up, left-to-right**
- **So, imperative semantic actions with global side effects can be performed.**
- **Appel shows 2 sorts of actions**
  - **Functional: no side effects, uses copies**
  - **Imperative: side effects with global variables**