4. Syntax-Directed Translation Schemes

- Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order.
- Some SDT's can be implemented during parsing, without building a parse tree.
  - But, not all SDT's can be implemented during parsing.
- Two important classes of SDD's will be considered:
  - LR-grammar and S-attributed SDD
  - LL-grammar and L-attributed SDD

Postfix Translation Schemes

SDT's with all actions at the right ends of the production bodies are called postfix SDT's

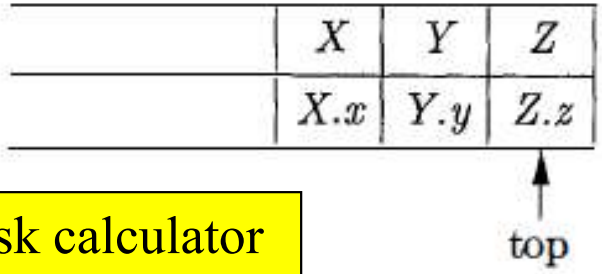
Example: Postfix SDT implementing the desk calculator

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	$\rightarrow$	$T_1 * F$	{ $T.val = T_1.val \times F.val$; }
T	$\rightarrow$	F	{ $T.val = F.val$; }
F	$\rightarrow$	(E)	{ $F.val = E.val$; }
F	$\rightarrow$	**digit**	{ $F.val =$ **digit**.$lexval$; }

Since the underlying grammar is LR, and the SDD is S-attributed, these actions can be correctly performed along with the reduction steps of the parser

Parser-Stack Implementation of Postfix SDT's

Parser stack



Implementing the desk calculator on a bottom-up parsing stack

PRODUCTION	ACTIONS
$L \rightarrow E$ **n**	{ print($stack[top-1].val$); $top = top - 1$; }
$E \rightarrow E_1 + T$	{ $stack[top-2].val = stack[top-2].val + stack[top].val$; $top = top - 2$; }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ $stack[top-2].val = stack[top-2].val \times stack[top].val$; $top = top - 2$; }
$T \rightarrow F$	
$F \rightarrow (E)$	{ $stack[top-2].val = stack[top-1].val$; $top = top - 2$; }
$F \rightarrow$ **digit**	

SDT's With Actions Inside Productions

- Consider a production: $B \rightarrow X \{\mathbf{a}\} Y$
  - The action **a** is done after we have recognized X (if X is a terminal) or all the terminals derived from X (if X is a nonterminal)
- Insert marker nonterminals to remove the embedded action and to change the SDT to a postfix SDT
  - Rewrite the product with marker nonterminal *M* into

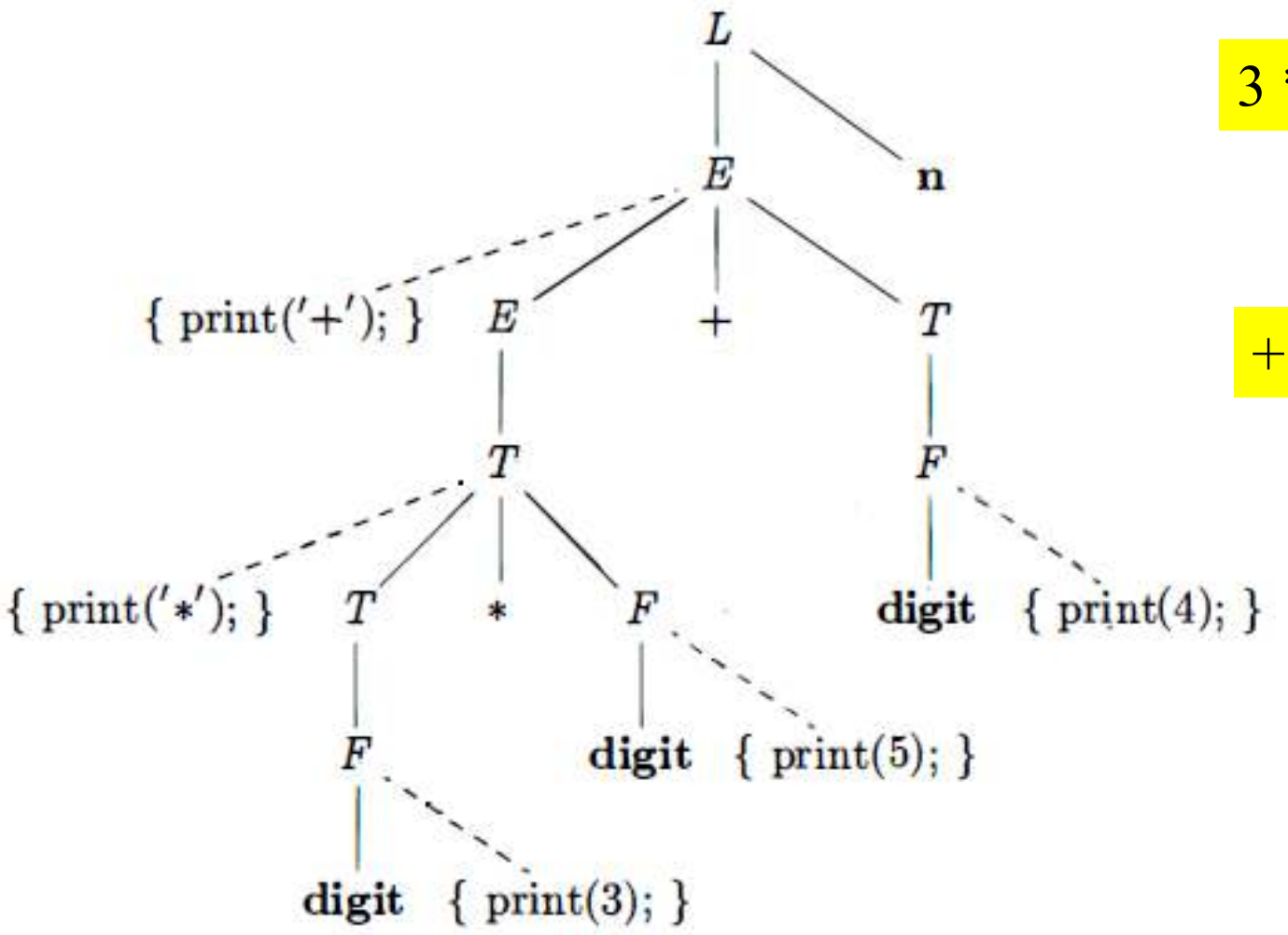
$$B \rightarrow X\, M\, Y$$
$$M \rightarrow \varepsilon \{\mathrm{a}\}$$

- Problems with inserting marker nonterminals
  - May introduce conflicts in the parse table
  - How to propagate inherited attributes?

Any SDT Can Be Implemented

1. Ignoring the actions, parse the input and produce a parse tree as a result.
2. Then, examine each interior node N, say one for production $A \rightarrow \alpha$. Add additional children to N for the actions in α, so the children of N from left to right have exactly the symbols and actions of α.
3. Perform a preorder traversal of the tree, and as soon as a node labeled by an action is visited, perform that action.

Parse Tree With Actions Embedded



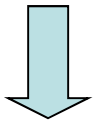
3 * 5 + 4

+ * 3 5 4

Eliminating Left Recursion From SDT's

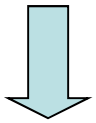
When the order in which the actions in an SDT is needed to consider only, the actions are treated as if they were terminal symbols during transforming the grammar,

$A \rightarrow A\ \alpha$
$A \rightarrow \beta$



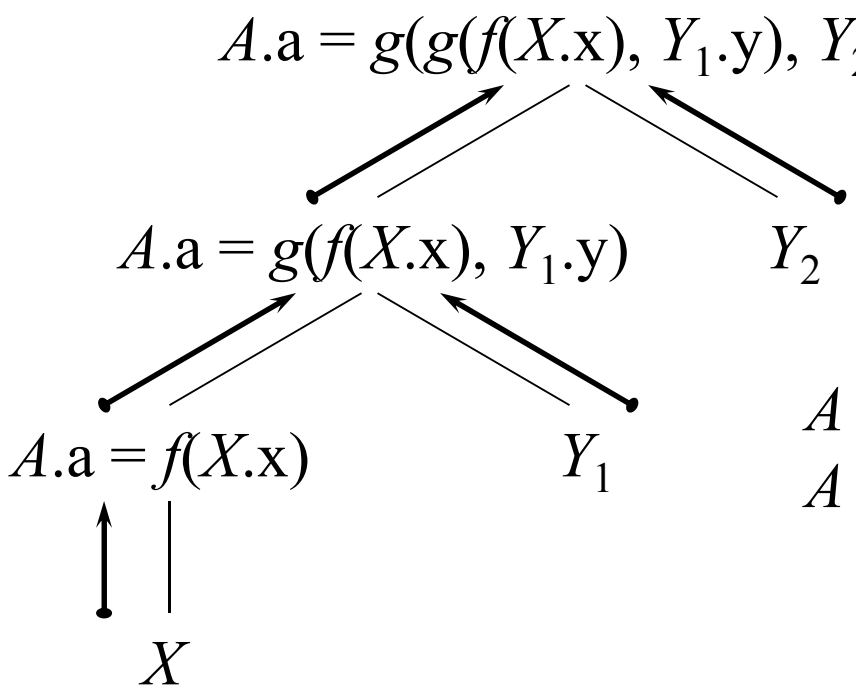
$A \rightarrow \beta\ R$
$R \rightarrow \alpha\ R$
$R \rightarrow \varepsilon$

$E \rightarrow E + T$ {print('+');}
$E \rightarrow T$



$E \rightarrow T\ R$
$R \rightarrow + T$ {print('+');} R
$R \rightarrow \varepsilon$

Eliminating Left Recursion From SDT's



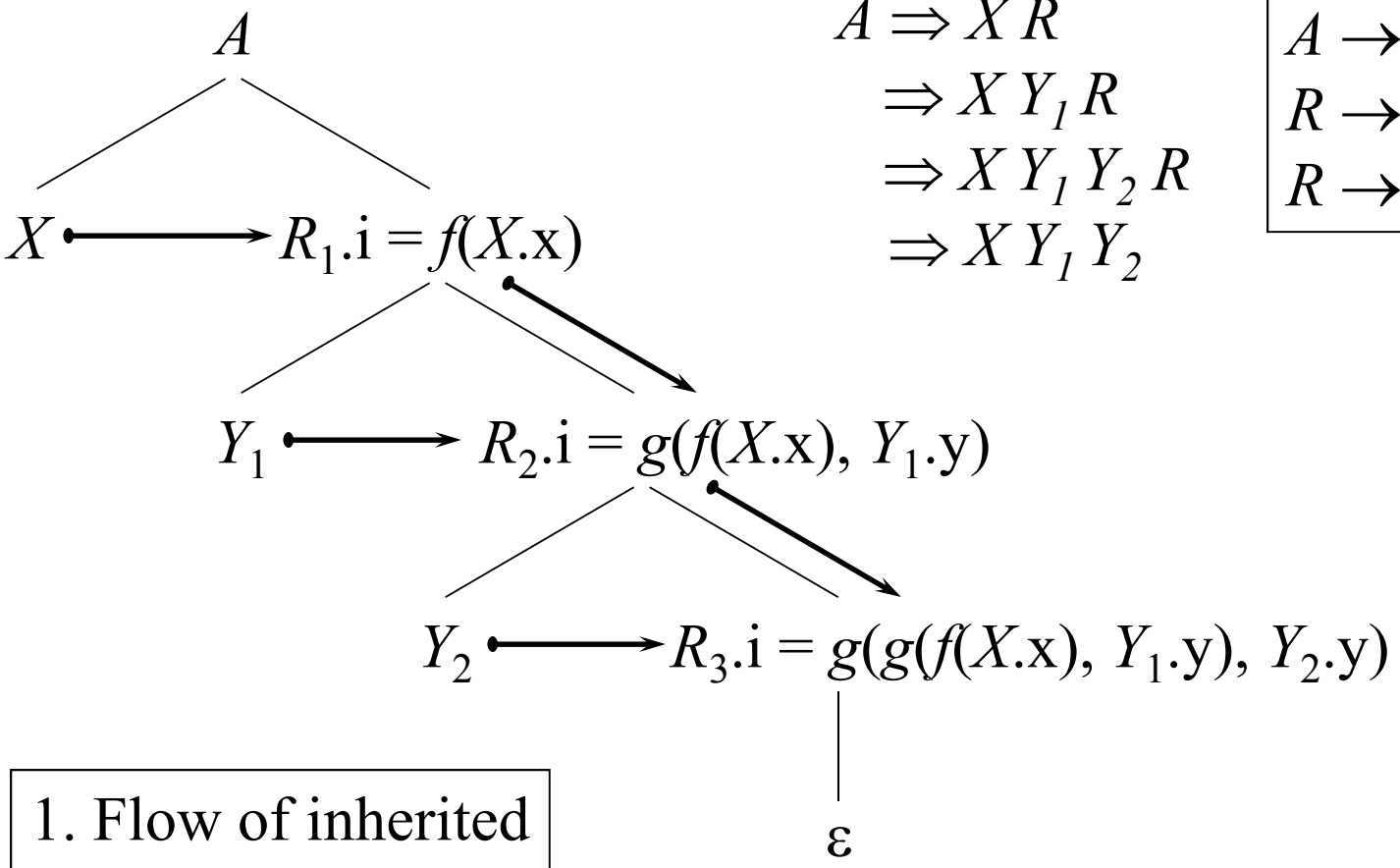
$A \Rightarrow A\ Y_2$
$\Rightarrow A\ Y_1\ Y_2$
$\Rightarrow X\ Y_1\ Y_2$

$A \to A\ Y$
$A \to X$

$A \to A_1\ Y\ \{\ A.\mathrm{a} = g(A_1.\mathrm{a},\ Y.\mathrm{y})\ \}$
$A \to X\ \{\ A.\mathrm{a} = f(X.\mathrm{x})\ \}$

$A \to X\ \{\ R.\mathrm{i} = f(X.\mathrm{x})\ \}\ R\ \{\ A.\mathrm{a} = R.\mathrm{s}\ \}$
$R \to Y\ \{\ R_1.\mathrm{i} = g(R.\mathrm{i},\ Y.\mathrm{y})\ \}\ R_1\ \{\ R.\mathrm{s} = R_1.\mathrm{s}\ \}$
$R \to \varepsilon\ \{\ R.\mathrm{s} = R.\mathrm{i}\ \}$

Eliminating Left Recursion (Cont.)

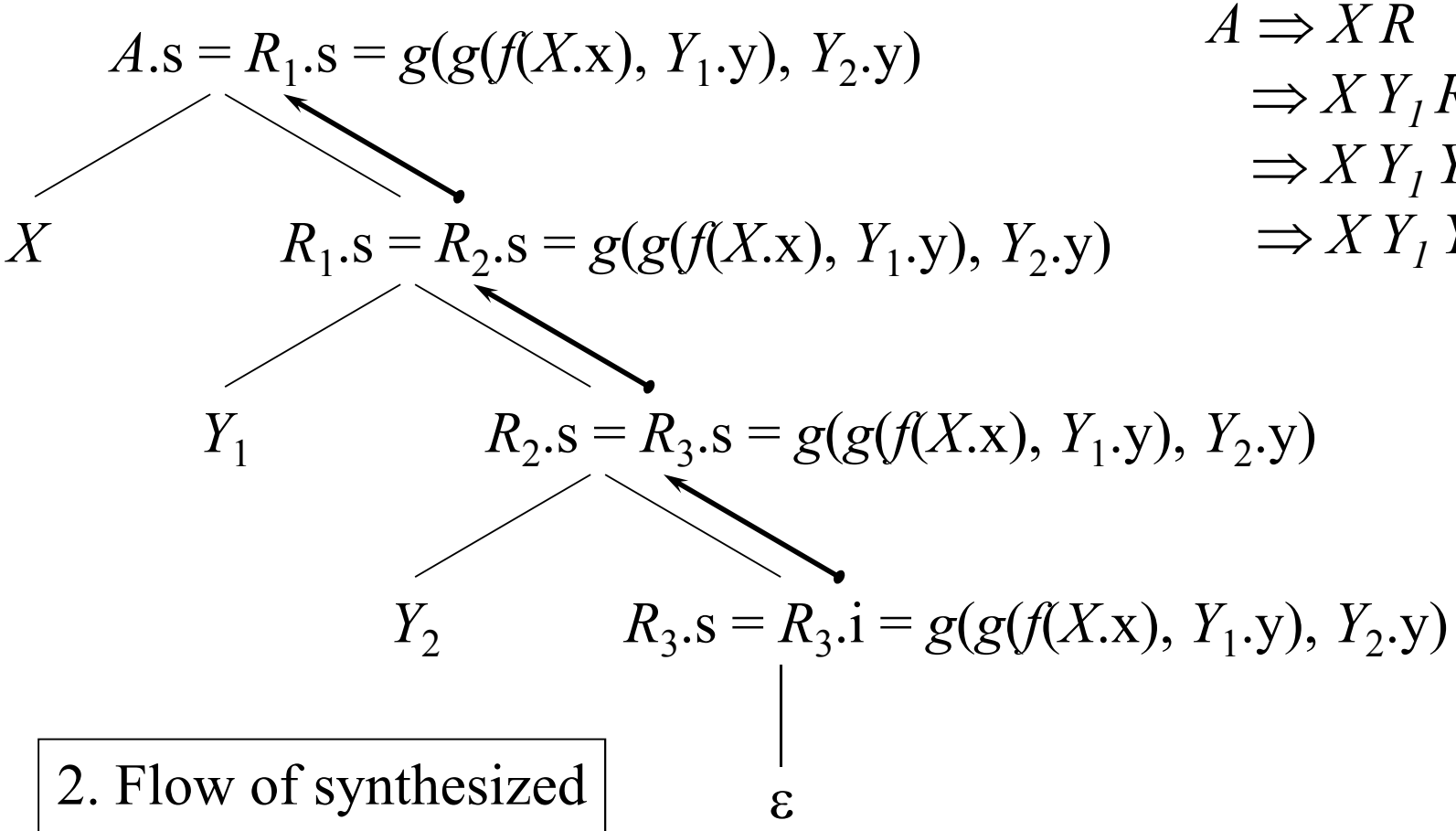


$A \Rightarrow X\,R$
$\Rightarrow X\,Y_1\,R$
$\Rightarrow X\,Y_1\,Y_2\,R$
$\Rightarrow X\,Y_1\,Y_2$

$A \rightarrow X\,R$
$R \rightarrow Y\,R$
$R \rightarrow \varepsilon$

1. Flow of inherited attribute values

Eliminating Left Recursion (Cont.)



$$A \Rightarrow X\,R$$
$$\Rightarrow X\,Y_1\,R$$
$$\Rightarrow X\,Y_1\,Y_2\,R$$
$$\Rightarrow X\,Y_1\,Y_2$$

2. Flow of synthesized attribute values

SDT's for L-Attributed Definitions

- Assume that the underlying grammar can be parsed top-down
- The rules for turning an L-attributed SDD into an SDT are as follows
  1. Embed the action that computes the inherited attributes for a nonterminal A immediately before that occurrence of A in the body of the production. If several inherited attributes for A depend on one another in an acyclic fashion, order the evaluation of attributes so that those needed first are computed first.
  2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production.

Example: Typesetting

Consider the following grammar

$$B \rightarrow B_1\ B_2 \mid B_1\ \textbf{sub}\ B_2 \mid (B_1) \mid \textbf{text}$$

The input string **a sub i sub j b sub k** will produce $a_{i_j} b_k$

This grammar is ambiguous, but we can still use it to parse bottom-up if we make subscripting and juxtaposition right associative, with sub taking precedence over juxtaposition.

Constructing larger boxes from smaller ones

SDD for typesetting boxes

	PRODUCTION	SEMANTIC RULES
1)	$S \to B$	$B.ps = 10$
2)	$B \to B_1\ B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = \max(B_1.ht, B_2.ht)$ $B.dp = \max(B_1.dp, B_2.dp)$
3)	$B \to B_1$ **sub** B_2	$B_1.ps = B.ps$ $B_2.ps = 0.7 \times B.ps$ $B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$
4)	$B \to (\ B_1\ )$	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
5)	$B \to$ **text**	$B.ht = getHt(B.ps, \textbf{text}.lexval)$ $B.dp = getDp(B.ps, \textbf{text}.lexval)$

SDT for typesetting boxes

	PRODUCTION	ACTIONS
1)	$S \rightarrow B$	$\{\ B.ps = 10;\ \}$
2)	$B \rightarrow B_1\ B_2$	$\{ B_1.ps = B.ps;\ \}$ $\{ B_2.ps = B.ps;\ \}$ $\{\ B.ht = \max(B_1.ht, B_2.ht);$ $\ B.dp = \max(B_1.dp, B_2.dp);\ \}$
3)	$B \rightarrow B_1\ \textbf{sub}\ B_2$	$\{ B_1.ps = B.ps;\ \}$ $\{ B_2.ps = 0.7 \times B.ps;\ \}$ $\{\ B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps);$ $\ B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps);\ \}$
4)	$B \rightarrow (\ B_1\ )$	$\{ B_1.ps = B.ps;\ \}$ $\{\ B.ht = B_1.ht;$ $\ B.dp = B_1.dp\ ;\ \}$
5)	$B \rightarrow \textbf{text}$	$\{\ B.ht = getHt(B.ps, \textbf{text}.lexval);$ $\ B.dp = getDp(B.ps, \textbf{text}.lexval);\ \}$

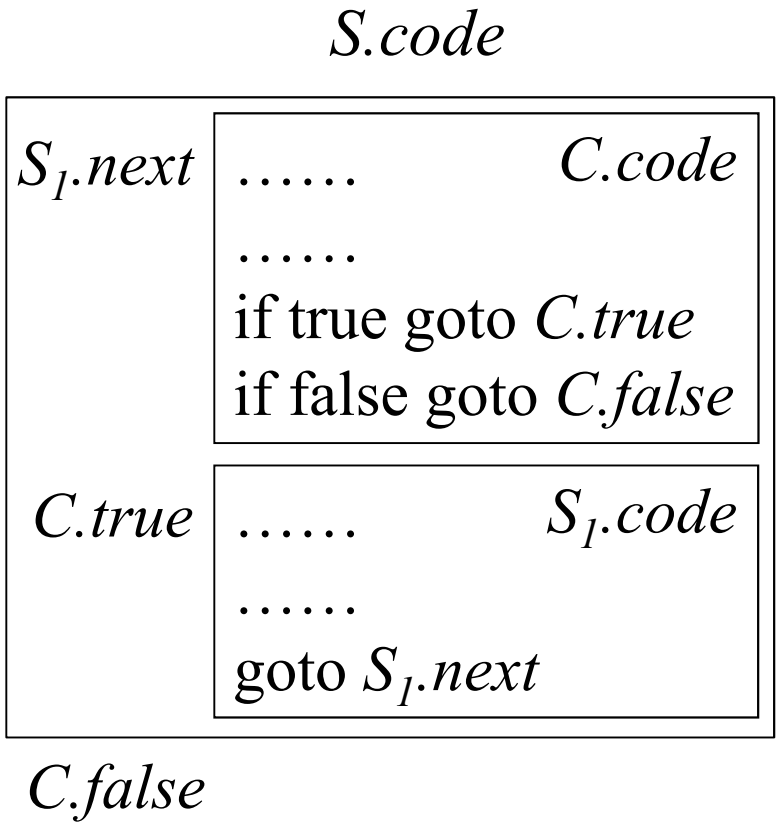
Example: Intermediate Code Generation

Consider the following grammar

$S \rightarrow while\ (\ C\ )\ S_1$

Inherited attributes
- *S.next*
- *C.true*
- *C.false*

synthesized attributes
- *S.code*
- *C.code*

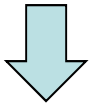


Intermediate Code Generation (Cont.)

SDD for while-statements

$S \rightarrow \textbf{while}\ (\ C\ )\ S_1$

$L1 = new();$
$L2 = new();$
$S_1.next = L1;$
$C.false = S.next;$
$C.true = L2;$
$S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code$



SDT for while-statements

$S \rightarrow \textbf{while}\ (\ C\ )\ S_1$

$\{\ L1 = new();\ L2 = new();\ C.false = S.next;\ C.true = L2;\ \}$
$\{\ S_1.next = L1;\ \}$
$\{\ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code;\ \}$

5. Implementing L-Attributed SDD's

Four methods for translation during parsing:

A. Use a recursive-descent parser with one function for each nonterminal.
   - The function for nonterminal A receives the inherited attributes of A as arguments and returns the synthesized attributes of A.

B. Generate code on the fly, using a recursive-descent parser.

C. Implement an SDT in conjunction with an LL-parser.
   - The attributes are kept on the parsing stack, and the rules fetch the needed attributes from known locations on the stack.

D. Implement an SDT in conjunction with an LR-parser.
   - If the underlying grammar is LL, we can always handle both the parsing and translation bottom-up.

A. Translation During Recursive-Descent Parsing

Example: Implementing while-statement

```
string S(label next) {
        string Scode, Ccode; /* local variables holding code fragments */
        label L1, L2; /* the local labels */
        if ( current input == token while ) {
                advance input;
                check '(' is next on the input, and advance;
                L1 = new();
                L2 = new();
                Ccode = C(next, L2);
                check ')' is next on the input, and advance;
                Scode = S(L1);
                return("label" || L1 || Ccode || "label" || L2 || Scode);
        }
        else /* other statement types */
}
```

B. On-The-Fly Intermediate Code Generation

Recursive-descent code generation for while-statement

```
void S(label next) {
    label L1, L2; /* the local labels */
    if ( current input == token while ) {
        advance input;
        check '(' is next on the input, and advance;
        L1 = new();
        L2 = new();
        print("label", L1);
        C(next, L2);
        check ')' is next on the input, and advance;
        print("label", L2);
        S(L1);
    }
    else /* other statement types */
}
```

On-The-Fly Code Generation (cont.)

SDT for on-the-fly code generation for while statements

Incidentally, we can make the same change to the underlying SDT: turn the construction of a main attribute into actions that emit the elements of that attribute

$$
\begin{array}{lll}
S \rightarrow \textbf{while} (& \{\ L1 = new();\ L2 = new();\ C.false = S.next; \\
 & \quad C.true = L2;\ print(\texttt{"label"}, L1);\ \} \\
C\ ) & \{\ S_1.next = L1;\ print(\texttt{"label"}, L2);\ \} \\
S_1 &
\end{array}
$$

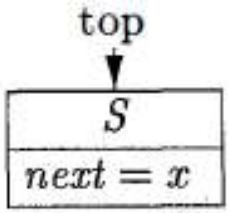
Figure 5.32: SDT for on-the-fly code generation for while statements

C. L-Attributed SDD's and LL Parsing

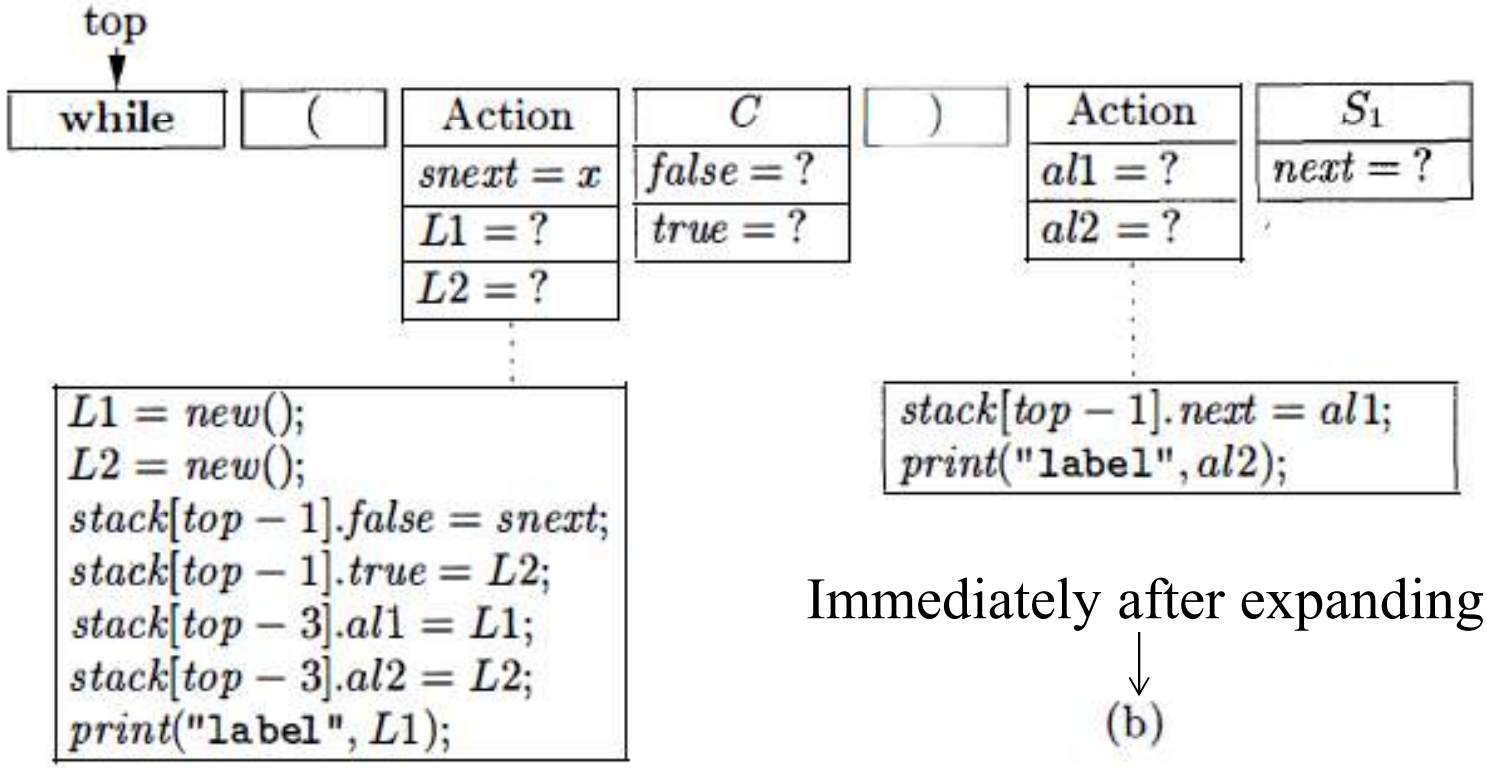
- For an SDT with embedded actions converted from an L-attributed SDD with an LL-grammar, then the translation can be performed during LL parsing by extending the parser stack to hold actions and certain data items needed for attribute evaluation.
- In addition to records representing terminals and nonterminals, the parser stack will hold
  - Inherited attributes for a nonterminal A are placed inside A
  - Action-records to represent actions to be executed are placed above A
  - Synthesize-records to hold the synthesized attributes for a nonterminals A are placed below A

Example: Implement the SDT of Fig.5.32

On-the-fly generation



(a) ← Just before expanding S



Immediately after expanding S

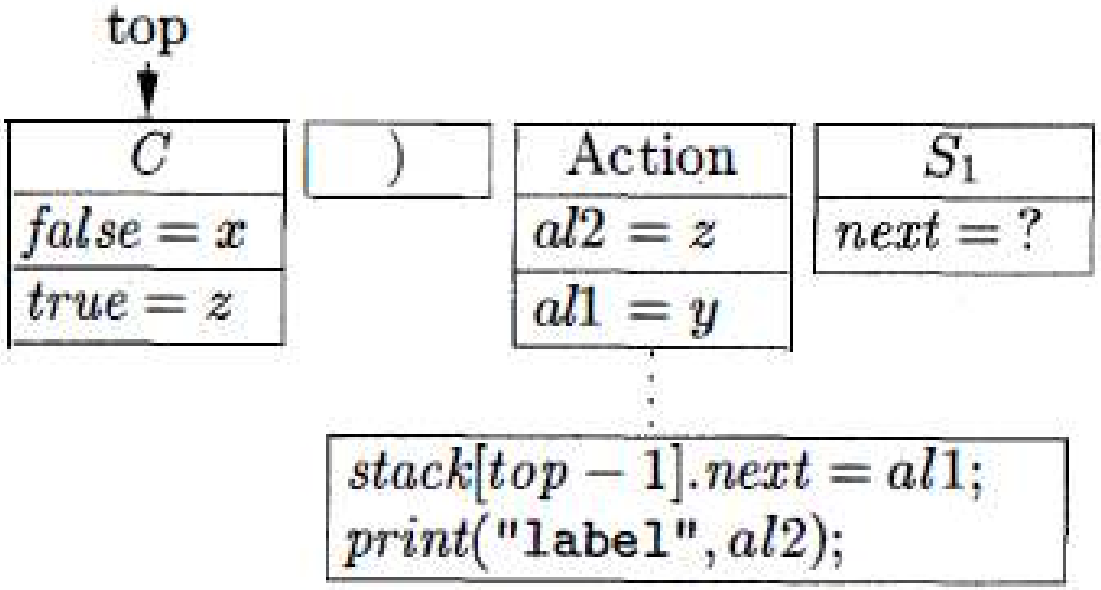
↓

(b)

Example: Implement the SDT of Fig.5.32 (Cont.)

On-the-fly generation

After the action above C is performed



Note: $L1 = y$, $L2 = z$

D. Bottom-Up Parsing of L-Attributed SDD's

L-attributed SDD on LL grammar can be adapted to compute the same SDD on the new grammar during an LR parse

1. Start with the SDTwith embedded actions before each nonterminal to compute its inherited attributes and an action at the end of the production to compute synthesized attributes.
2. Introduce into a distinct marker nonterminal M in place of each embedded action, and add one production $M \rightarrow \varepsilon$.
3. Modify the action **a** if M replaces it in some production $A \rightarrow \alpha\{\mathbf{a}\}\beta$, and associate with $M \rightarrow \varepsilon$ an action **a'** that
   a) Copies, as inherited attributes of M, any attributes of A or symbols of α that action **a** needs.
   b) Computes attributes in the same way as **a**, but makes those attributes be synthesized attributes of M.

Turn SDT to Operate with LR Parse

Example 5.25:

$A \rightarrow \{B.i = f(A.i);\ \}\ B\ C$ ⇒

$A \rightarrow M\ B\ C$

$M \rightarrow \varepsilon\ \{M.i = A.i;\ M.s = f(M.i);\ \}$

Example 5.26:

$S \rightarrow$ **while** ({ $L1 = new()$; $L2 = new()$; C.false = S.next; C.true = L2; }

C) { S_1.next = $L1$; }

S_1 { S.code = **label** || $L1$ || C.code || **label** || $L2$ || S_1.code; }

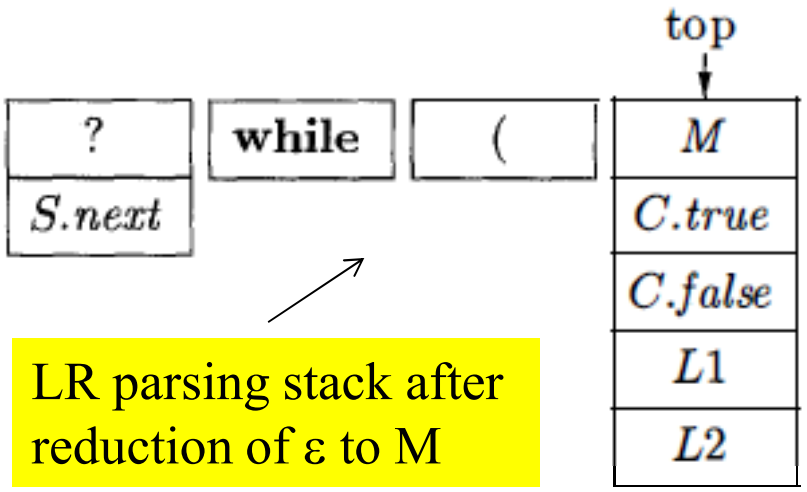
⇓

$S \rightarrow$ **while** ($M\ C$) $N\ S_1$ { S.code = **label** || $L1$ || C.code || **label** || $L2$ || S_1.code; }

$M \rightarrow \varepsilon$ { $L1 = new()$; $L2 = new()$; C.false = S.next; C.true = $L2$;}

$N \rightarrow \varepsilon$ { S_1.next = $L1$; }

Example 5.26 (Cont.)



LR parsing stack after reduction of ε to M

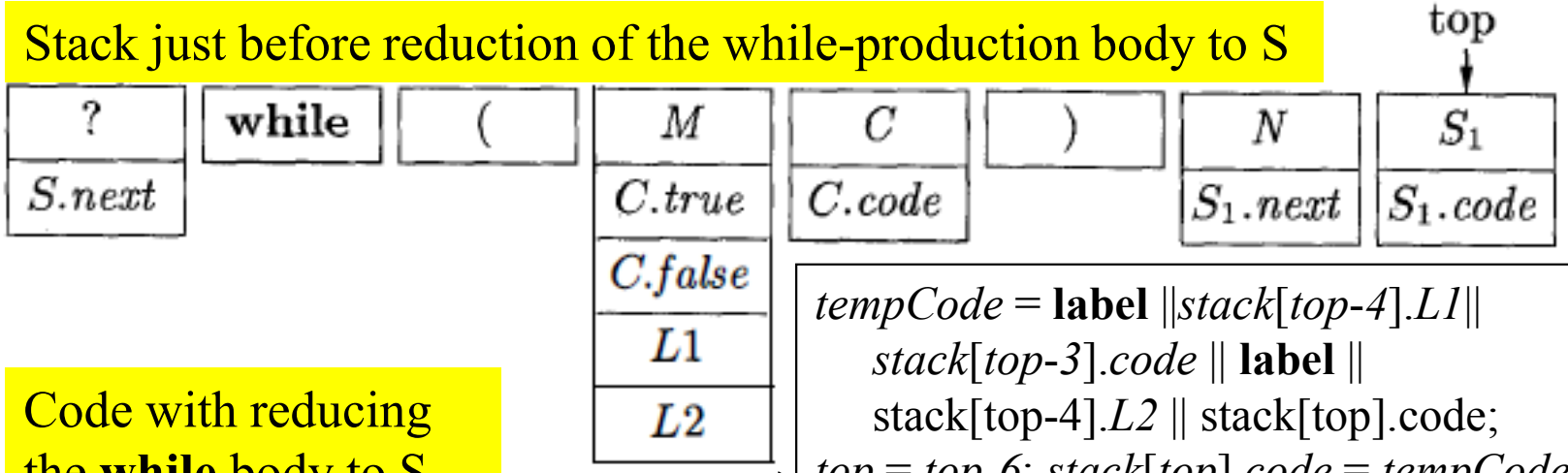
Code with M→ε

L1 = *new*();
L2 = *new*();
C.true = *L2*;
C.false = stack[top-3].next;

Code with N→ε

S_1.next = stack[top-3].L1;

Stack just before reduction of the while-production body to S



Code with reducing the **while** body to S.

tempCode = **label** ||*stack*[*top-4*].*L1*||
stack[*top-3*].*code* || **label** ||
stack[top-4].*L2* || stack[top].code;
top = *top-6*; *stack*[*top*].*code* = *tempCode*;