CS 4300: Compiler Theory

Chapter 5 Syntax-Directed Translation

Dr. Xuejun Liang

Outlines (Sections)

Quick Review of Last Lecture

- Syntax-Directed Translation Schemes
  - Postfix Translation Schemes
  - Parser-Stack Implementation of Postfix SDT's
  - SDT's With Actions Inside Productions
  - Eliminating Left Recursion From SDT's

$$\boxed{\begin{array}{l} E \rightarrow E + T \ \ \{\text{print('+');}\} \\ E \rightarrow T \end{array}} \quad \Rightarrow \quad \boxed{\begin{array}{l} E \rightarrow T \ R \\ R \rightarrow + \ T \ \ \{\text{print('+');}\} \ R \\ R \rightarrow \varepsilon \end{array}}$$

$$\boxed{\begin{array}{l} A \rightarrow A_1 \, Y \ \{ \ A.\text{a} = g(A_1.\text{a}, \ Y.\text{y}) \ \} \\ A \rightarrow X \ \{ \ A.\text{a} = f(X.\text{x}) \ \} \end{array}} \quad \Rightarrow \quad \boxed{\begin{array}{l} A \rightarrow X \ \{ \ R.\text{i} = f(X.\text{x}) \ \} \ R \ \{ \ A.\text{a} = R.\text{s} \ \} \\ R \rightarrow Y \ \{ \ R_1.\text{i} = g(R.\text{i}, \ Y.\text{y}) \ \} \ R_1 \ \{ \ R.\text{s} = R_1.\text{s} \ \} \\ R \rightarrow \varepsilon \ \{ \ R.\text{s} = R.\text{i} \ \} \end{array}}$$

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.

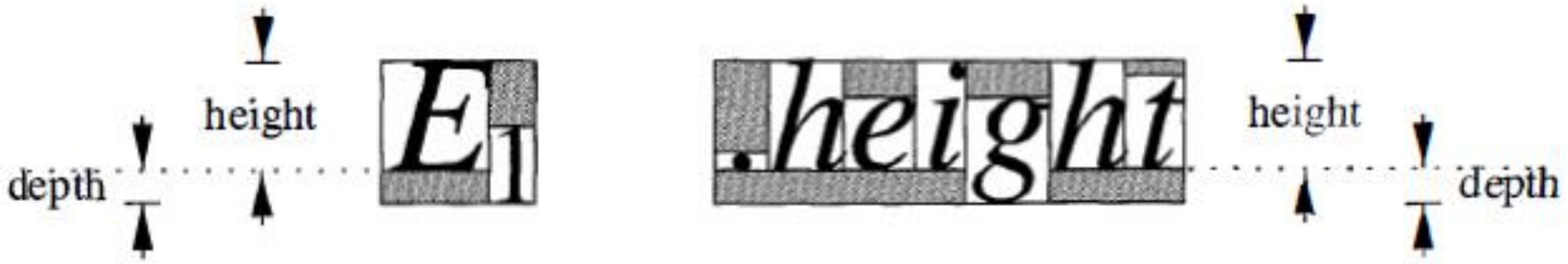
Example: Typesetting

Consider the following grammar

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

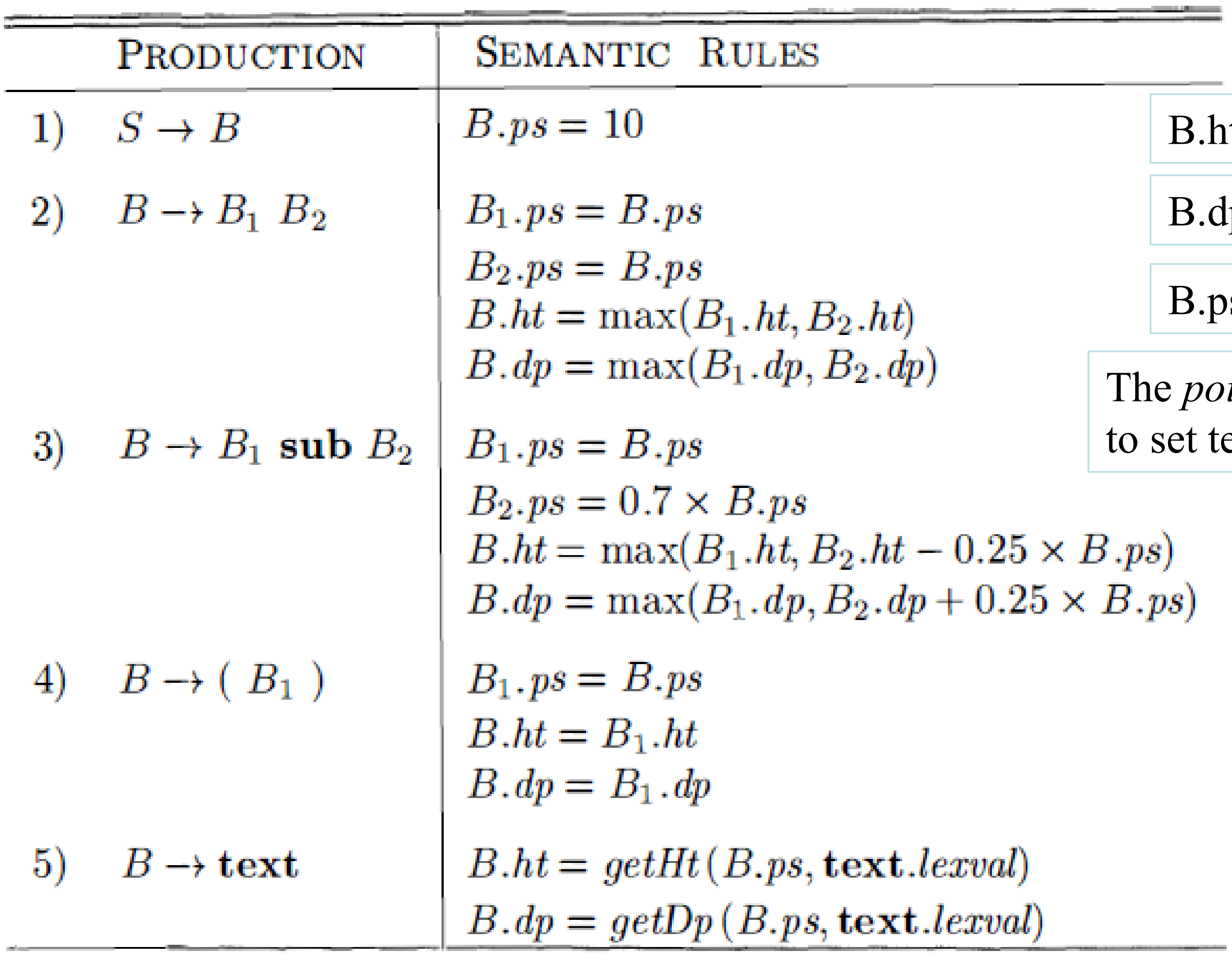
The input string **a sub i sub j b sub k** will produce $a_{i_j} b_k$

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



Constructing larger boxes from smaller ones

SDD for typesetting boxes



	PRODUCTION	SEMANTIC RULES
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)	$B \rightarrow B_1$ **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)	$B \rightarrow$ **text**	$B.ht = getHt(B.ps, \textbf{text}.lexval)$ $B.dp = getDp(B.ps, \textbf{text}.lexval)$

B.ht synthesized

B.dp synthesized

B.ps inherited

The *point size* is used to set text within a box

SDT for typesetting boxes

B.ht synthesized

B.dp synthesized

B.ps inherited

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

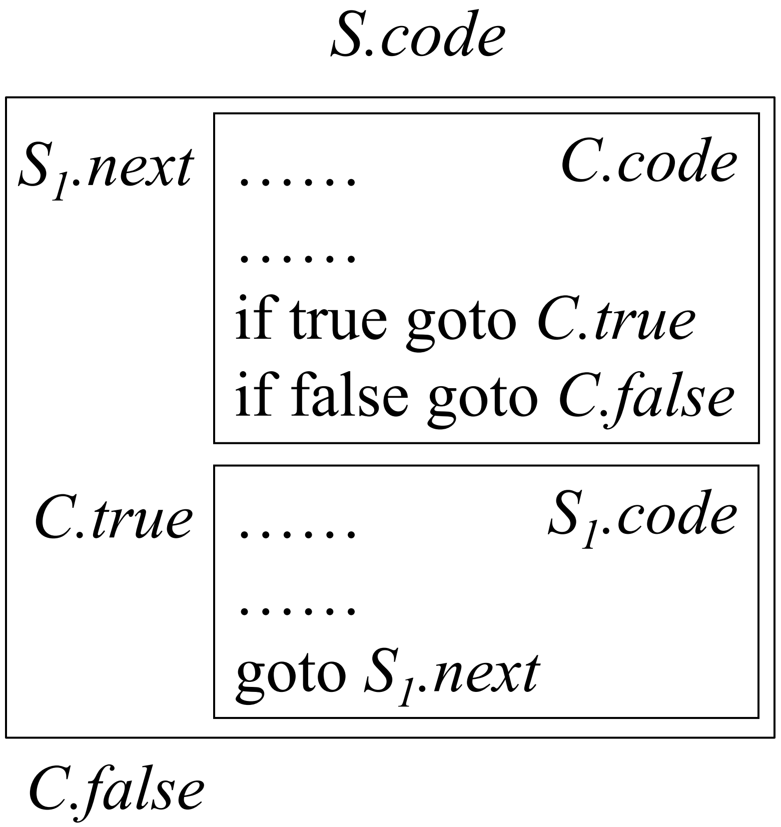
$S \rightarrow while\ (\ C\ )\ S_1$

Inherited attributes

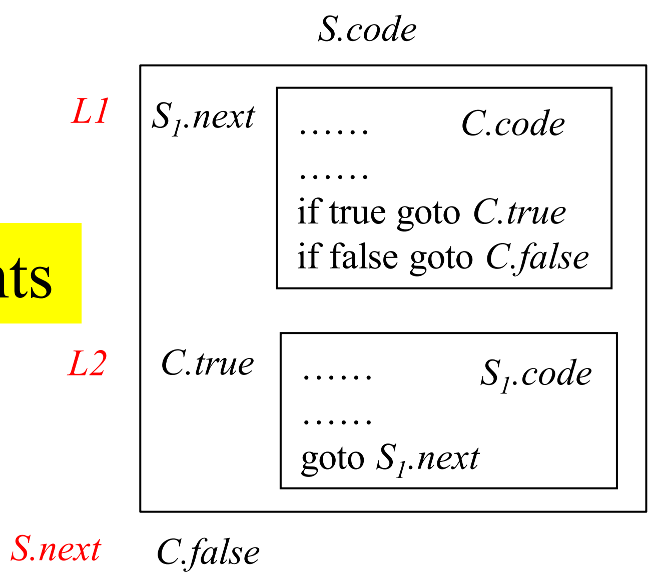
S.next
C.true
C.false

synthesized attributes

S.code
C.code



Example (Cont.)



SDD for while-statements

$S \rightarrow$ **while** (C) S_1 $\quad$ $L1 = new();$
$L2 = new();$
$S_1.next = L1;$
$C.false = S.next;$
$C.true = L2;$
$S.code = \textbf{label} \,\|\, L1 \,\|\, C.code \,\|\, \textbf{label} \,\|\, L2 \,\|\, S_1.code$

SDT for while-statements

$S \rightarrow$ **while** ($\quad$ { $L1 = new();\ L2 = new();\ C.false = S.next;\ C.true = L2;$ }
C) $\quad$ { $S_1.next = L1;$ }
S_1 $\quad$ { $S.code = \textbf{label} \,\|\, L1 \,\|\, C.code \,\|\, \textbf{label} \,\|\, L2 \,\|\, S_1.code;$ }

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

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

```
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 */
}
```

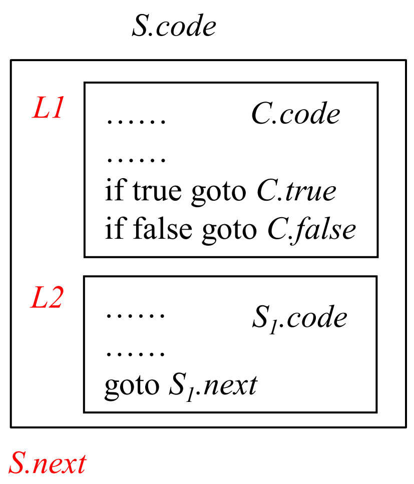


Figure 5.29: Implementing with a recursive-descent parser

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 */
}
```

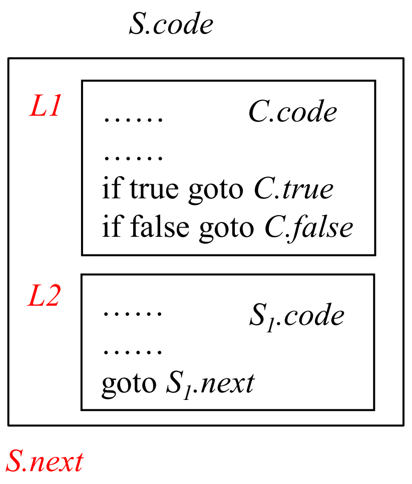


Figure 5.31: On-the-fly recursive-descent code generation

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

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

$$
\begin{array}{lll}
S \rightarrow \textbf{while}\ (& & \{\ L1 = new();\ L2 = new();\ C.false = S.next;\ C.true = L2;\ \} \\
 & C\ ) & \{\ S_1.next = L1;\ \} \\
 & S_1 & \{\ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code;\ \}
\end{array}
$$

Figure 5.28: SDT for while-statement

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

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

C. L-Attributed SDD's and LL Parsing

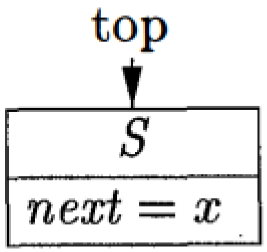
- For an SDT with embedded actions converted from an L-attributed SDD with an LL-grammar, the translation can be performed during LL parsing by extending the parser stack to hold actions and certain data items needed for attribute evaluation.
- In addition to records representing terminals and nonterminals,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

Example 5.23: Implement the SDT of Fig.5.32

```
S  →  while (   { L1 = new(); L2 = new(); C.false = S.next;
                  C.true = L2; print("label", L1); }
      C )       { S1.next = L1; print("label", L2); }
      S1
```

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

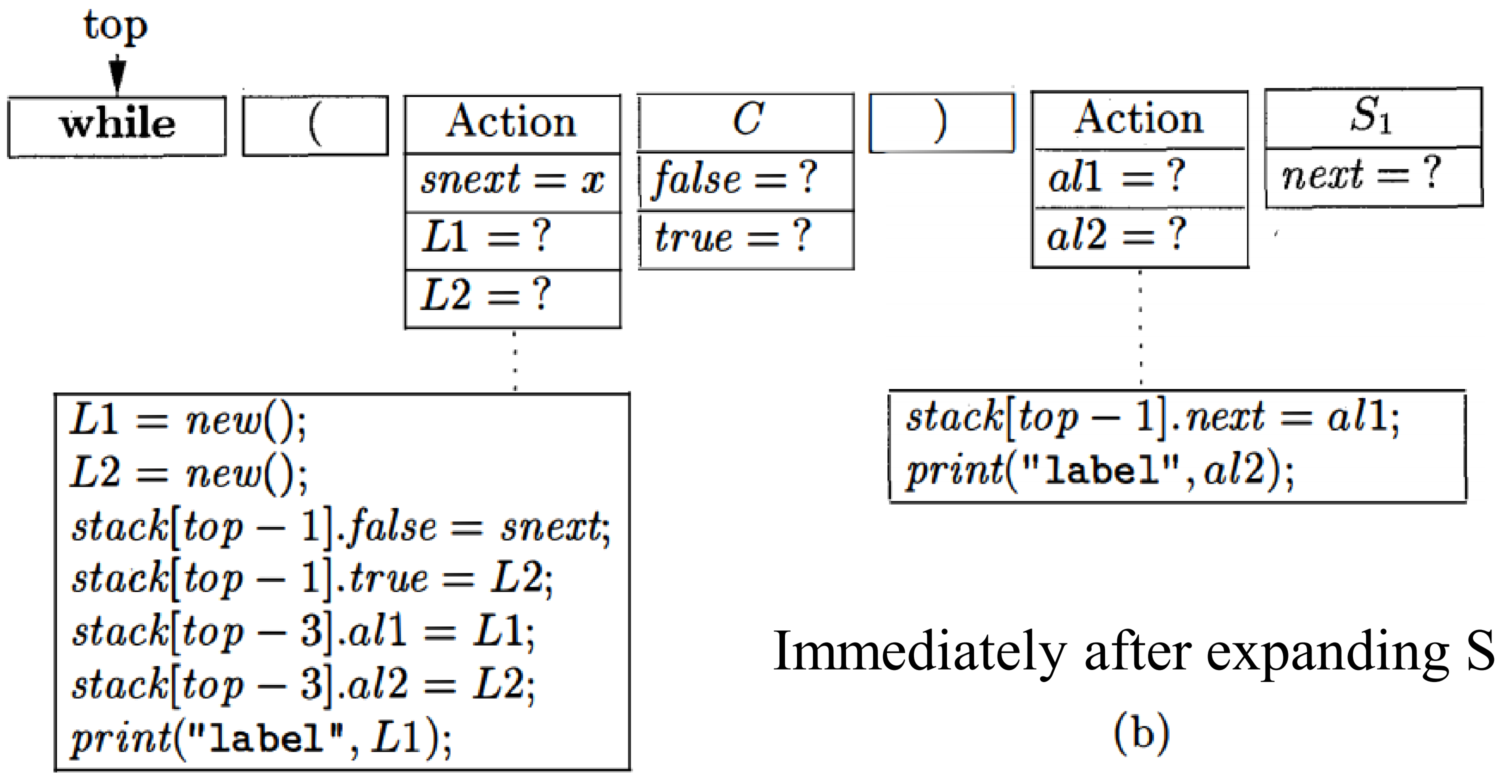
This example will illustrate the implementation of inherited attributes during LL parsing by copying attribute values.



(a) ⟵ Just before expanding S

Example 5.23 (Cont.)

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



$L1 = new();$
$L2 = new();$
$stack[top-1].false = snext;$
$stack[top-1].true = L2;$
$stack[top-3].al1 = L1;$
$stack[top-3].al2 = L2;$
$print(\texttt{"label"}, L1);$

$stack[top-1].next = al1;$
$print(\texttt{"label"}, al2);$

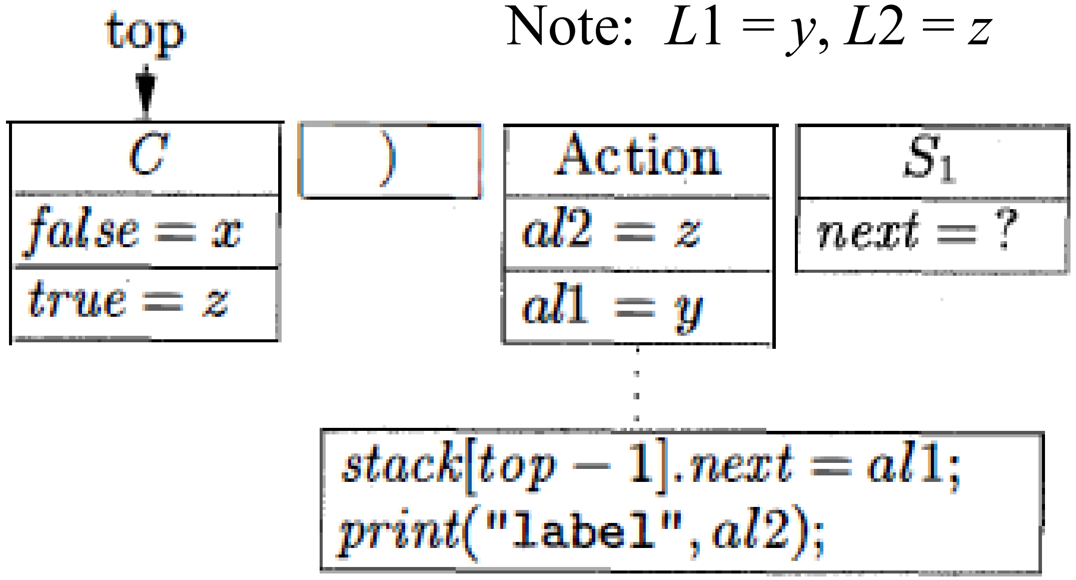
Immediately after expanding S

(b)

Example 5.23 (Cont.)

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

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



After the action above C is performed

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 SDT with 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 $A \rightarrow \alpha\{$**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 5.26: Let us turn the SDT of Fig. 5.28 into an SDT that can operate with an LR parse of the revised grammar

$S \rightarrow$ **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; }

Underlying grammar

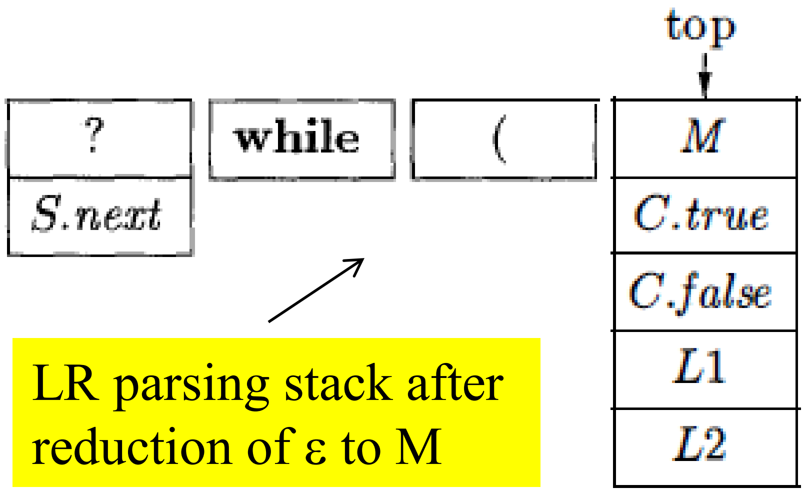
$$\begin{array}{lll} S & \rightarrow & \textbf{while}\ (\ M\ C\ )\ N\ S_1 \\ M & \rightarrow & \epsilon \\ N & \rightarrow & \epsilon \end{array}$$

$S \rightarrow$ **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;

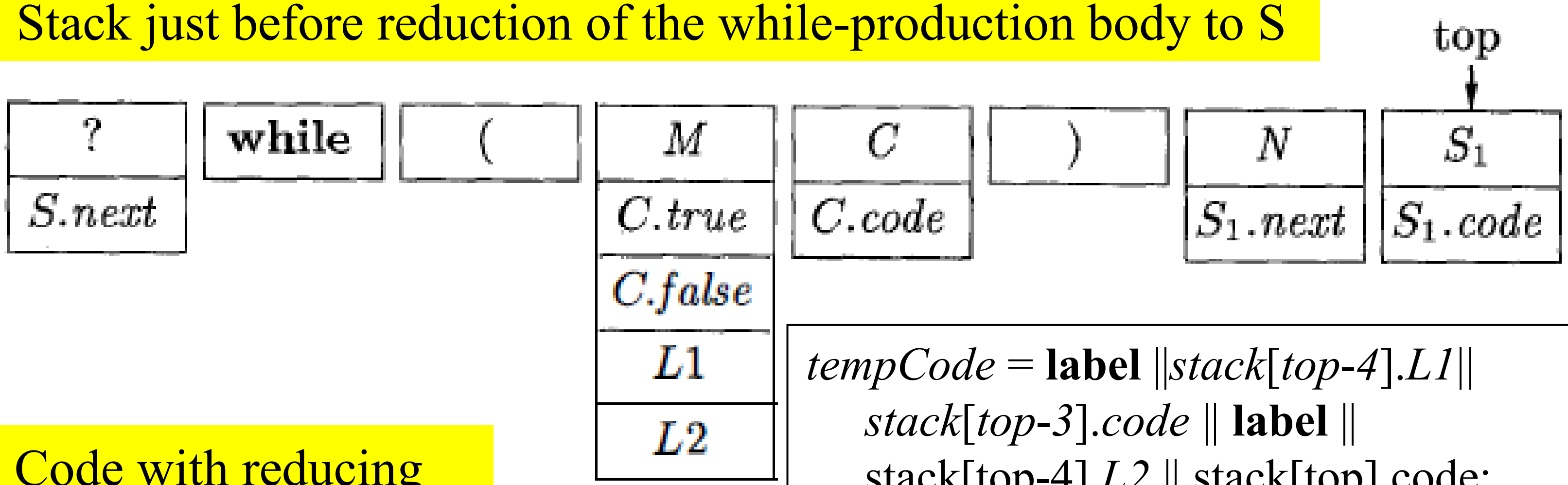
$S \rightarrow$ **while** (M C) N S_1 { *S*.code = **label** || *L1* || *C*.code || **label** || *L2* || S_1.code; }
$M \rightarrow \varepsilon$ { *L1* = *new*(); *L2* = *new*(); *C*.false = *S*.next; *C*.true = *L2*;}
$N \rightarrow \varepsilon$ { S_1.next = *L1*; }

Example 5.26 (Cont.)

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*;

$S \rightarrow$ **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 = $L1$; }