



第5章 语法制导翻译 (2)

Syntax-Directed Translation
【第5.3, 5.4节】



回顾：语法制导定义 (SDD)

- 为每个符号X添加相应的属性 $X.x$, 对于产生式 $A \rightarrow XYZ$
 - 综合属性: $A.a = f(A.i, X.x, Y.y, Z.z)$
 - 继承属性: $Y.i = g(A.a, X.x, Y.y, Z.z)$
- SDD的分类
 - S-属性的SDD: 只包含综合属性的SDD
 - L-属性的SDD: 一个符号的继承属性只依赖于它左边的符号的属性。
 - 所有S-属性的SDD都是L-属性的SDD



主要内容

- 语法制导翻译概述
- 语法制导定义 (SDD)
 - SDD的求值顺序
- 语法制导翻译的应用
 - 抽象语法树的构造
 - 类型结构
- 语法制导的翻译方案 (SDT)
- L属性的SDD的实现方法



构造抽象语法树的SDD

□ 抽象语法树 (Abstract Syntax Tree)

- 每个结点代表一个语法结构；对应于一个运算符
- 结点的每个子结点代表其子结构；对应于运算分量
- 表示这些子结构按照特定方式组成更大的结构
- 可以忽略掉一些标点符号等非本质的东西

□ 语法树的表示方法

- 每个结点用一个对象表示
- 对象有多个域
 - 叶子结点中只存放词法值；
 - 内部结点中存放op（操作符）值和参数（通常指向其它结点）



抽象语法树的例子

- 产生式 $S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$ 的语法树

if-then-else



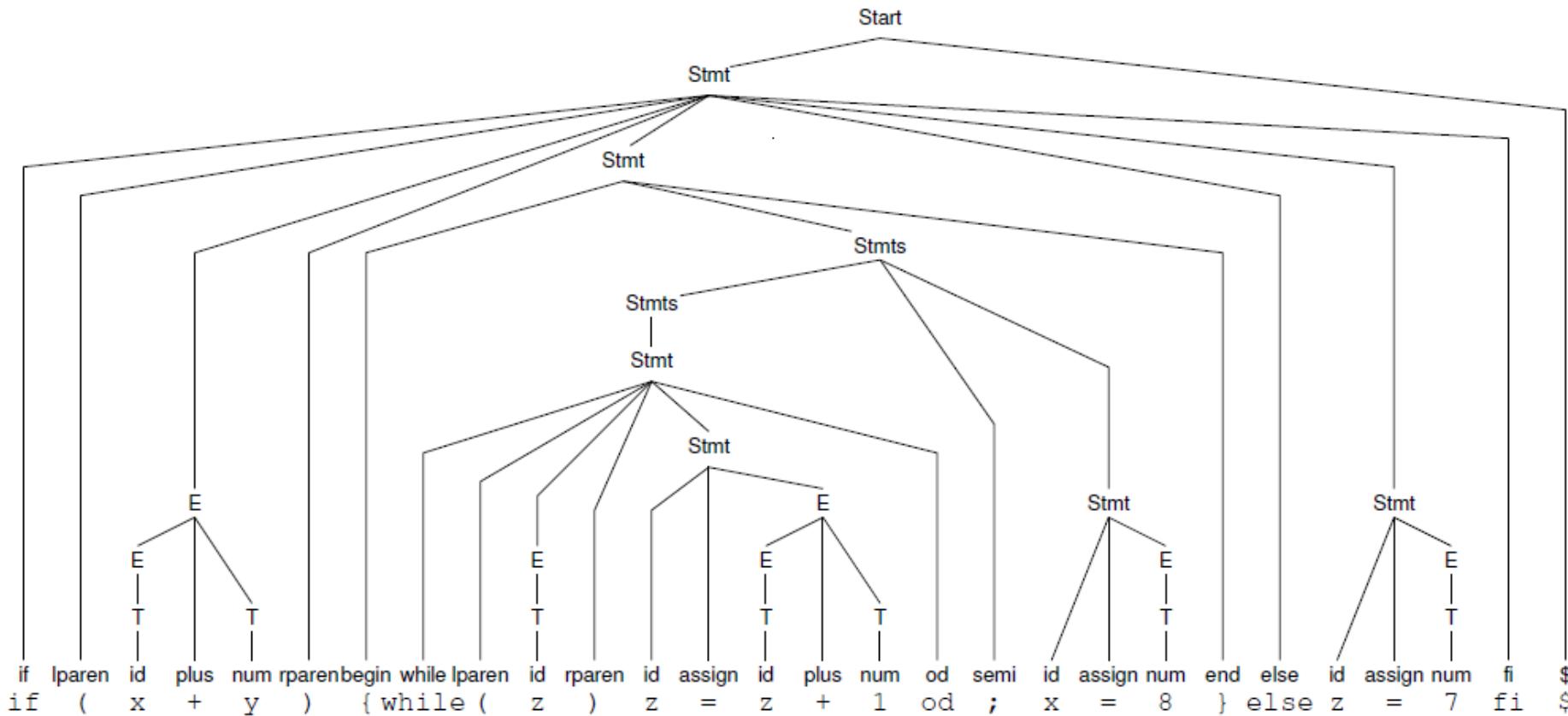
- 赋值语句的语法树

assignment



- 在语法树中，运算符号和关键字都不在叶结点，而是在内部结点中出现。

具体语法树 vs. 抽象语法树



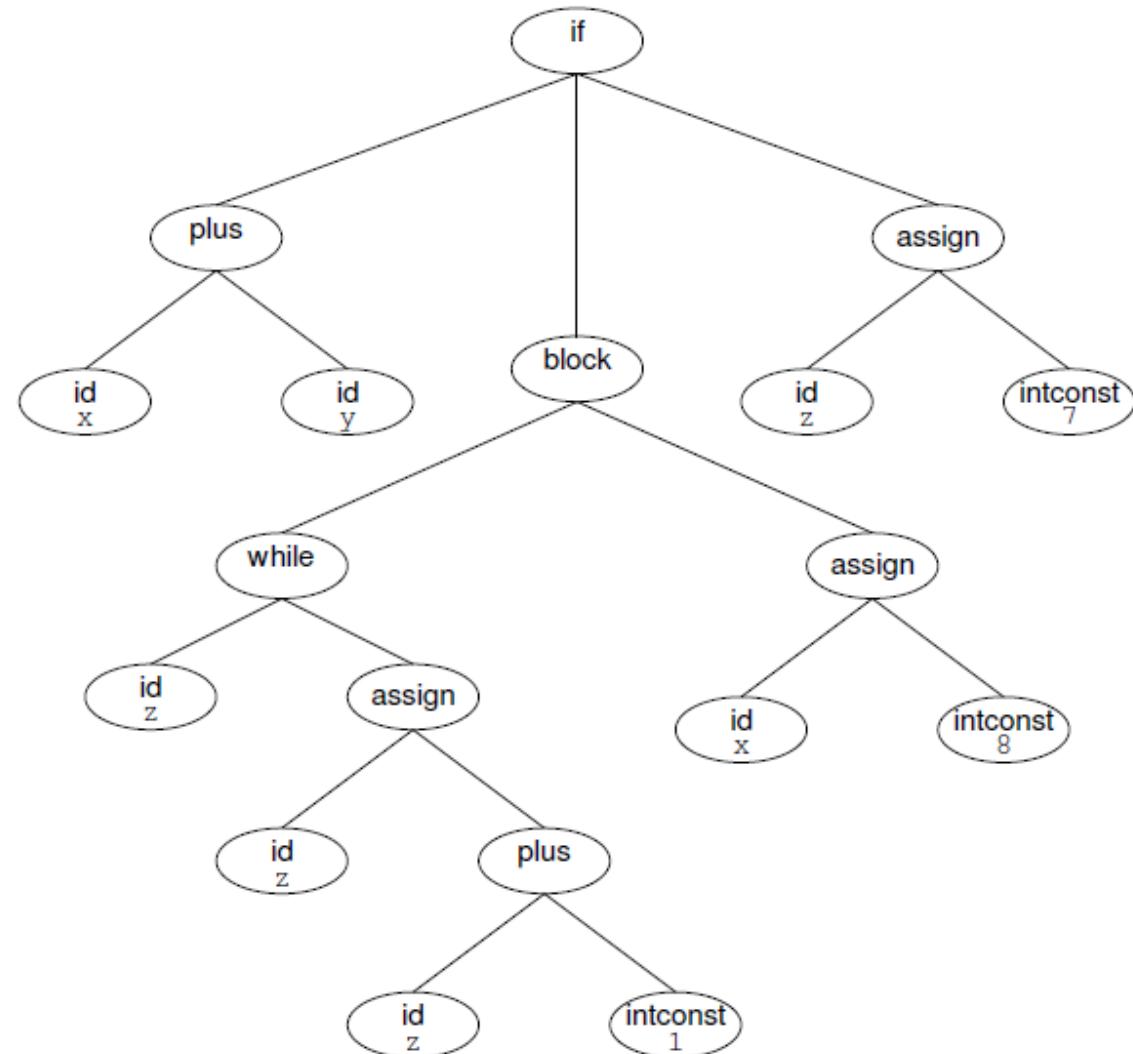
if (x+y) { while (z) z=z+1 od; x =8 } else z = 7 fi \$
 的具体语法树（分析树）



具体语法树 vs. 抽象语法树

```
if (x+y) {  
while (z)  
z=z+1 od;  
x =8 }  
else z = 7 fi $
```

的抽象语法树
(AST)





构造简单表达式的语法树的SDD

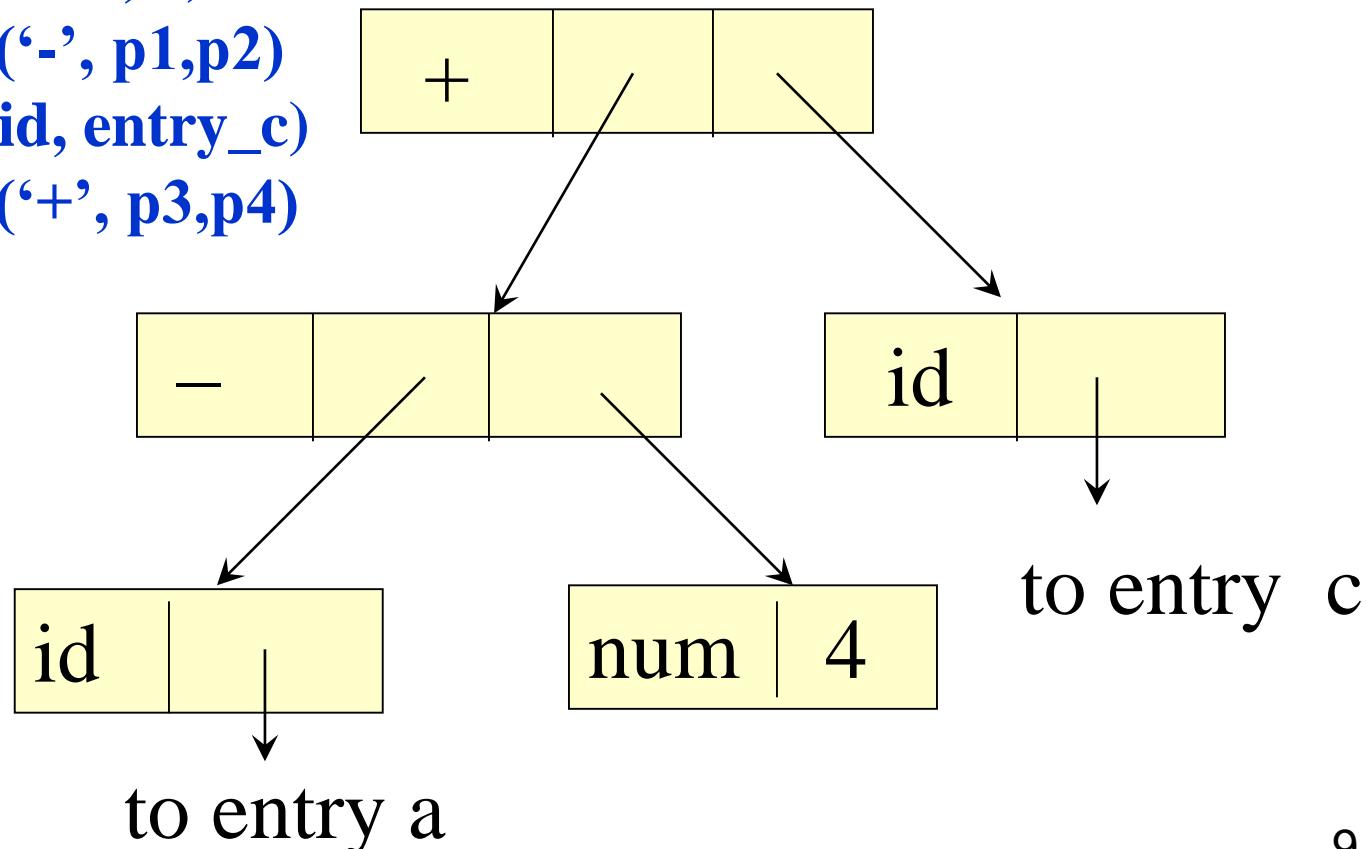
- 属性E.node指向E对应的语法树的根结点

产生式	语义规则
$E \rightarrow E_1 + T$	$E.\text{node} = \text{new Node} ('+', E_1.\text{node}, T.\text{node})$
$E \rightarrow E_1 - T$	$E.\text{node} = \text{new Node} ('-', E_1.\text{node}, T.\text{node})$
$E \rightarrow T$	$E.\text{node} = T.\text{node}$
$T \rightarrow (E)$	$T.\text{node} = E.\text{node}$
$T \rightarrow \text{id}$	$T.\text{node} = \text{new Leaf} (\text{id}, \text{id.name})$
$T \rightarrow \text{num}$	$T.\text{node} = \text{new Leaf} (\text{num}, \text{num.val})$

例：a-4+c的语法树的构造过程

• 构造步骤：

- p1=new Leaf(id, entry_a)
- p2=new Leaf(num, 4)
- p3=new Node('-', p1, p2)
- p4=new Leaf(id, entry_c)
- p5=new Node('+', p3, p4)



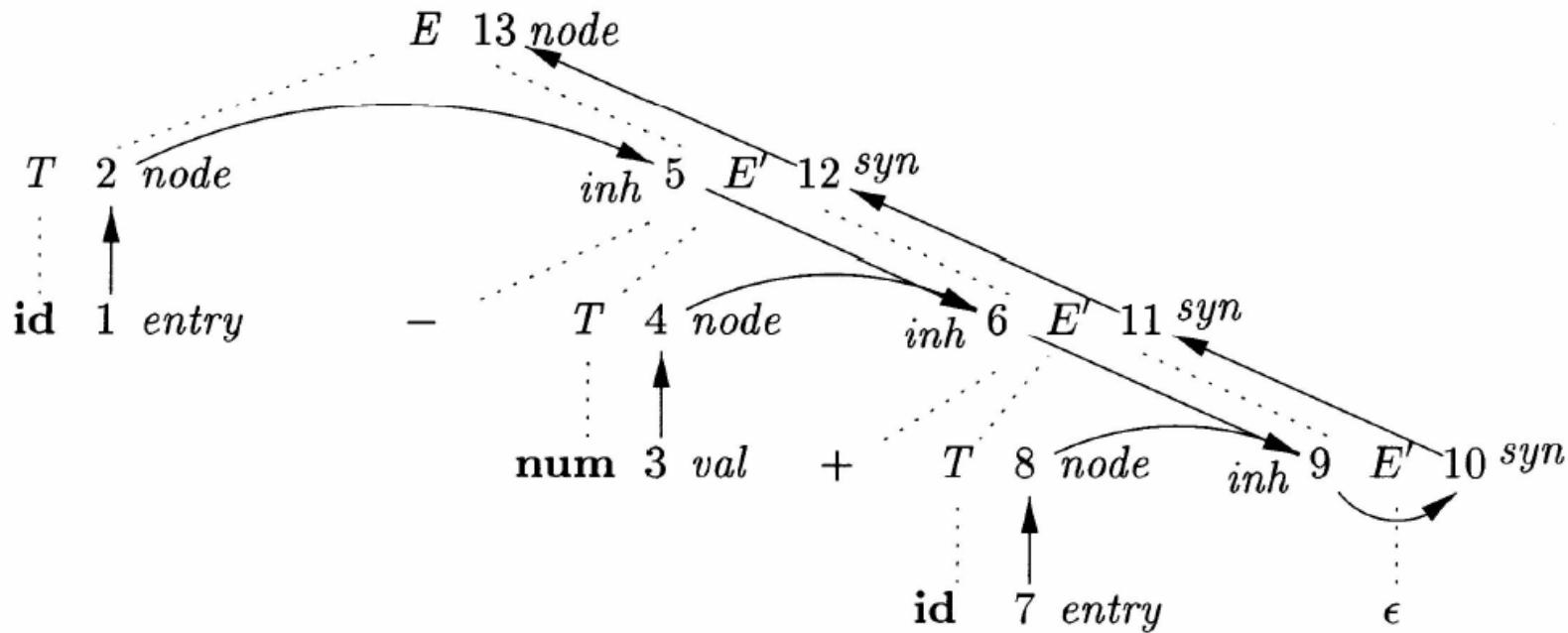


在自顶向下分析过程中构造AST

产生式	语义规则
1) $E \rightarrow T E'$	$E.\text{node} = E'.\text{syn}$ $E'.\text{inh} = T.\text{node}$
2) $E' \rightarrow + T E'_1$	$E'_1.\text{inh} = \mathbf{new} \ Node(' + ', E'.\text{inh}, T.\text{node})$ $E'.\text{syn} = E'_1.\text{syn}$
3) $E' \rightarrow - T E'_1$	$E'_1.\text{inh} = \mathbf{new} \ Node(' - ', E'.\text{inh}, T.\text{node})$ $E'.\text{syn} = E'_1.\text{syn}$
4) $E' \rightarrow \epsilon$	$E'.\text{syn} = E'.\text{inh}$
5) $T \rightarrow (E)$	$T.\text{node} = E.\text{node}$
6) $T \rightarrow \mathbf{id}$	$T.\text{node} = \mathbf{new} \ Leaf(\mathbf{id}, \mathbf{id}.\text{entry})$
7) $T \rightarrow \mathbf{num}$	$T.\text{node} = \mathbf{new} \ Leaf(\mathbf{num}, \mathbf{num}.\text{val})$

自顶向下的AST构造过程

- 对于这个SDD，各属性值的计算过程实际上和原来S属性定义中的计算过程一致
- 继承属性可以把值从一个结构传递到另一个并列的结构；也可把值从父结构传递到子结构





类型结构

□ 简化的类型表达式文法

- $T \rightarrow B\ C$ $B \rightarrow \text{int} \mid \text{float}$
- $C \rightarrow [\text{num}]C \mid \epsilon$

□ 生成类型表达式的SDD

产生式	语义规则
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow \text{int}$	$B.t = \text{integer}$
$B \rightarrow \text{float}$	$B.t = \text{float}$
$C \rightarrow [\text{num}]C_1$	$C.t = \text{array}(\text{num}.val, C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$



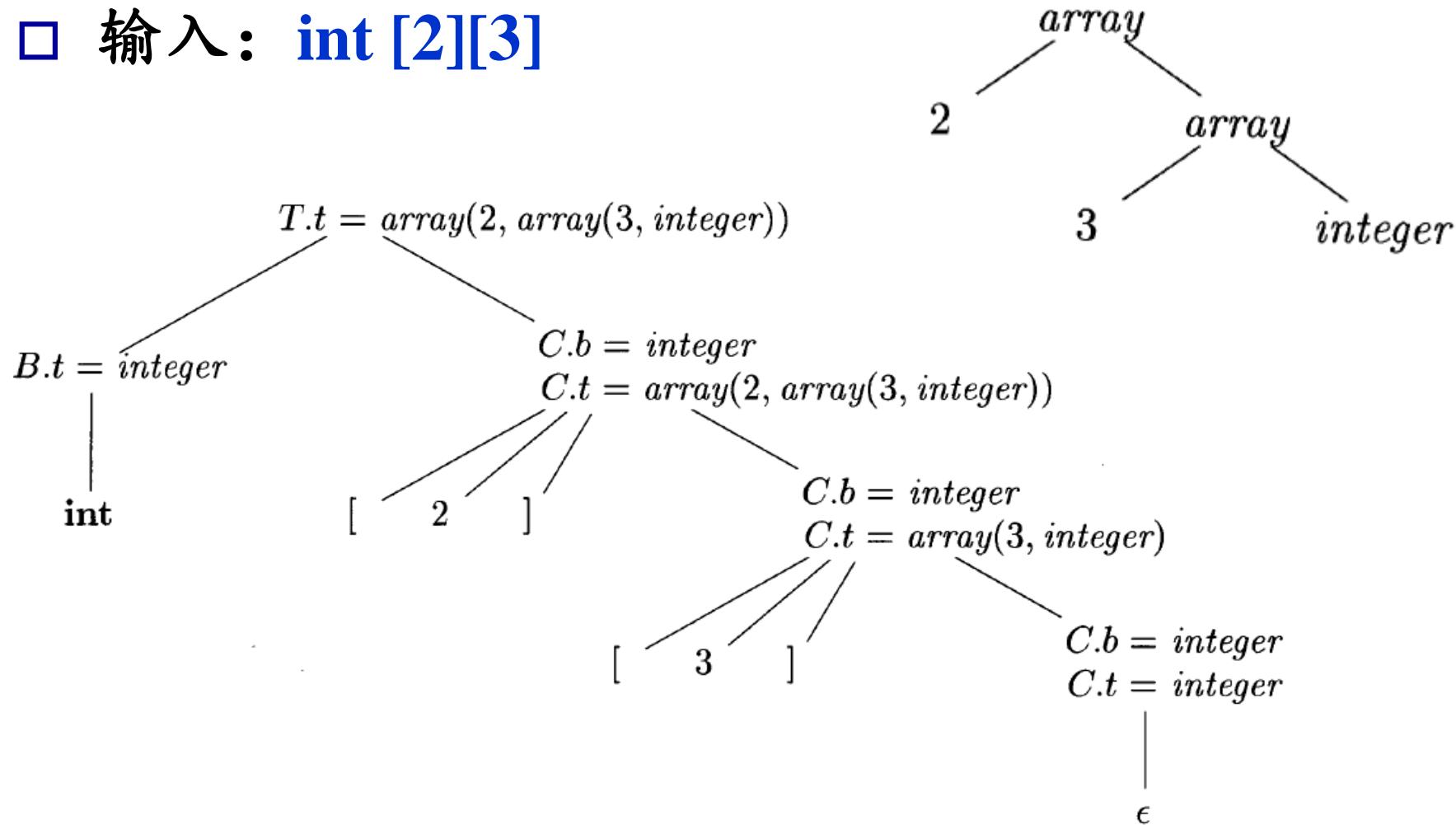
类型的含义

- 类型包括两个部分： $T \rightarrow B\ C$
 - 基本类型 B
 - 分量 C
- 分量形如[3][4]
 - 表示 3×4 的二维数组
- int [3][4]
- 数组构造算符array
 - `array(3,array(4,integer))`表示 3×4 的二维数组



类型表达式的生成过程

□ 输入： int [2][3]





主要内容

- 语法制导翻译概述
- 语法制导定义 (SDD)
 - SDD的求值顺序
- 语法制导翻译的应用
 - 抽象语法树的构造
 - 类型结构
- 语法制导的翻译方案 (SDT)
- L属性的SDD的实现方法



语法制导的翻译方案 (SDT)

- 语法制导的翻译方案 (**syntax-directed translation scheme**) 是对语法制导定义的补充
 - 也称作语法制导的翻译模式
 - 把SDD的语义规则改写为计算属性值的程序片段用花括号 { } 括起来，插入到产生式右部的任何合适的位置上
 - 这是一种语法分析和语义动作交错的表示法，它表达在按深度优先遍历分析树的过程中何时执行语义动作
- 原来的不含语义动作的文法称作是基础文法



SDT的实现方法

- SDT的基本实现方法：
 - 建立语法分析树
 - 将语义动作看作是虚拟的结点
 - 从左到右、深度优先地遍历分析树，在访问虚拟结点时执行相应动作
- 通常情况下在语法分析过程中实现，不需要真的构造语法分析树
- 可以用SDT实现两类重要的SDD
 - 基础文法是LR的，SDD是S属性的
 - 基础文法是LL的，SDD是L属性的



翻译方案示例：

- 一个简单的SDT(只包含+/-操作的表达式)

$E \rightarrow TR$

$R \rightarrow addop\ T\ \{print(addop.lexeme)\}\ R_1 | \epsilon$

$T \rightarrow num\ \{print(num.val)\}$

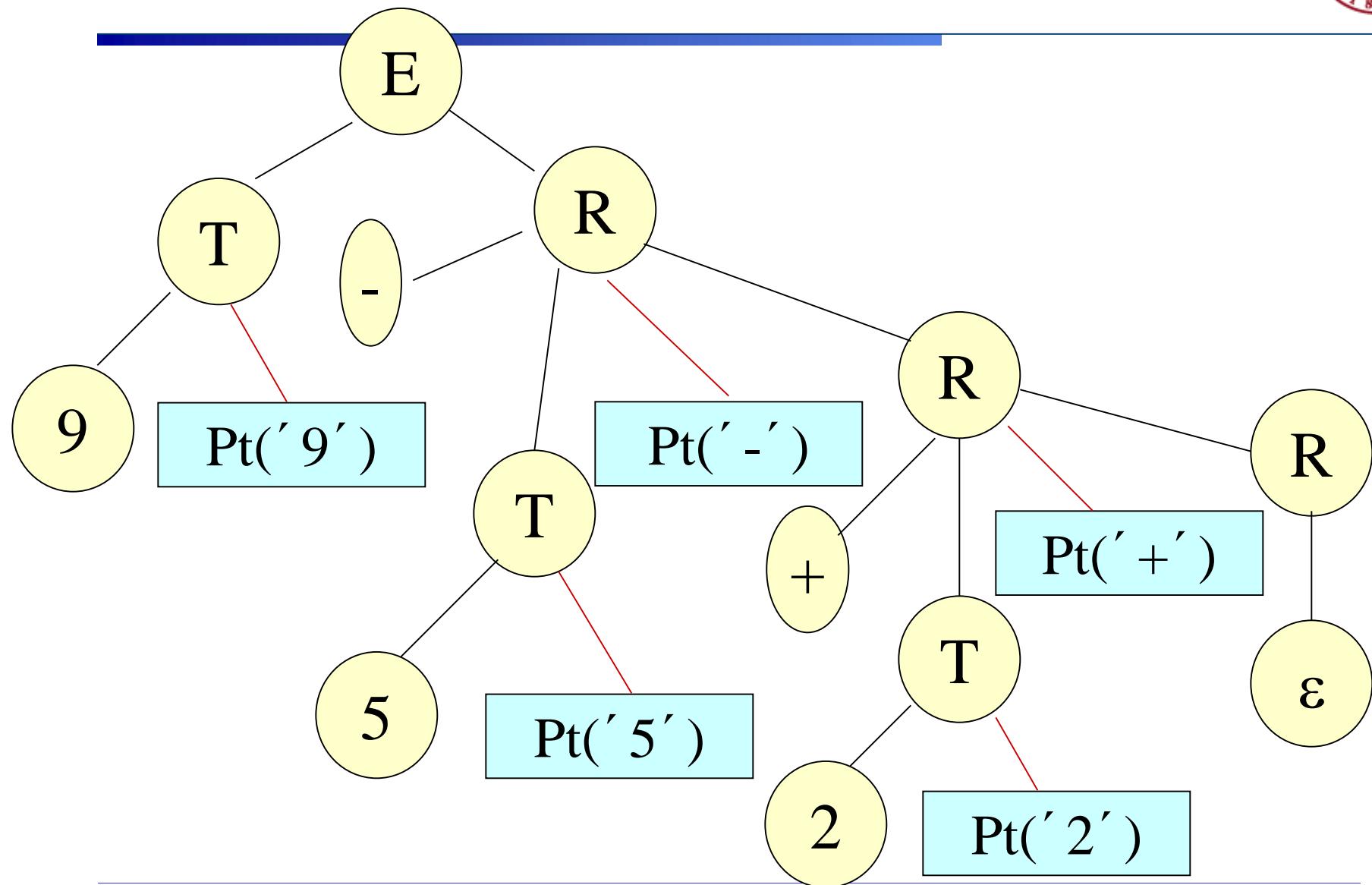
- 把语义动作看成终结符号，输入 **9-5+2**，其分析树见下页，当按深度优先遍历它，执行遍历中访问的语义动作，将输出

9 5 - 2 +

- 输出的是输入表达式**9-5+2**的后缀式



9-5+2的带语义动作的分析树





翻译方案的设计（1/2）

- 根据语法制导定义设计翻译方案
- 需要保证语义动作不会引用还没有计算的属性值。分情况考虑：

1. 只需要综合属性的情况：

- 为每一个语义规则建立一个包含赋值的动作，并把这个动作放在相应的产生式右边的末尾。

例如： $T \rightarrow T_1 * F$ 所需动作： $T.val = T_1.val * F.val$
 $T \rightarrow T_1 * F \quad \{ T.val = T_1.val * F.val \}$



翻译方案的设计（2/2）

2. 既有综合属性又有继承属性：

- 产生式右边的符号的继承属性必须在这个符号以前的动作中计算出来。
- 一个动作不能引用这个动作右边符号的综合属性。
- 产生式左边非终结符号的综合属性只有在它所引用的所有属性都计算出来以后才能计算。计算这种属性的动作通常可放在产生式右端的末尾。



不正确的翻译方案

下面的翻译方案不满足要求：

$$\begin{array}{ll} S \rightarrow A_1 A_2 & \{ A_1.in=1; \quad A_2.in=2 \} \\ A \rightarrow a & \{ \text{print}(A.in) \} \end{array}$$

可以改成如下的形式：

$$\begin{array}{ll} S \rightarrow \{ A_1.in=1 \} A_1 \{ A_2.in=2 \} A_2 & \\ A \rightarrow a & \{ \text{print}(A.in) \} \end{array}$$



如何构造语义动作集合？

- 构造合适的语义动作集合，需要深入理解在给定分析技术（自顶向下或自底向上）中推导的过程。
- 要想书写清晰而简洁的语义动作，通常需要对文法进行重新构造，以帮助在合适的地方来计算语义值。
- 在最初获得了适合自顶向下或自底向上分析的文法之后，往往在编译器构造的这个阶段还是会对文法进行修改以提供对语义动作的支持。



后缀翻译方案

- 文法可以自底向上分析且SDD是S属性的，必然可以构造出**后缀SDT**
- **后缀SDT**: 所有动作都在产生式最右端的**SDT**
- 构造方法：
 - 将每个语义规则看作是一个赋值语义动作
 - 将所有的语义动作放在规则的最右端



后缀翻译方案的例子

□ 实现桌上计算器的后缀SDT

$L \rightarrow E \text{ n}$	{ print($E.val$); }
$E \rightarrow E_1 + T$	{ $E.val = E_1.val + T.val;$ }
$E \rightarrow T$	{ $E.val = T.val;$ }
$T \rightarrow T_1 * F$	{ $T.val = T_1.val \times F.val;$ }
$T \rightarrow F$	{ $T.val = F.val;$ }
$F \rightarrow (E)$	{ $F.val = E.val;$ }
$F \rightarrow \text{digit}$	{ $F.val = \text{digit}.lexval;$ }



后缀SDT的语法分析栈实现

- 可以在LR语法分析的过程中实现
 - 归约时执行相应的语义动作
 - 定义用于记录各文法符号的属性的union结构
 - 栈中的每个文法符号（或者说状态）都附带一个这样的union类型的值
 - 在按照产生式 $A \rightarrow XYZ$ 归约时，Z的属性可以在栈顶找到，Y的属性可以在下一个位置找到，X的属性可以在再下一个位置找到。

	X	Y	Z	状态 / 文法符号
	$X.x$	$Y.y$	$Z.z$	综合属性
				↑ 栈顶



state	val
...	...
X	X.x
Y	Y.y
Z	Z.z

top →

state	val
...	...
A	A.a

定义 $\text{A.a} = \text{f}(\text{X.x}, \text{Y.y}, \text{Z.z})$ (抽象表示) 对应的动作

```
stack[top-2].val =  
    f(stack[top-2].val, stack[top-1].val, stack[top].val);  
top = top-2;
```



后缀SDT的栈实现

产生式

语义动作

$L \rightarrow E \ n$ { print($stack[top - 1].val$);
 $top = top - 1;$ }

$E \rightarrow E_1 + T$ { $stack[top - 2].val = stack[top - 2].val + stack[top].val$;
 $top = top - 2;$ }

$E \rightarrow T$

$T \rightarrow T_1 * F$ { $stack[top - 2].val = stack[top - 2].val \times stack[top].val$;
 $top = top - 2;$ }

$T \rightarrow F$

$F \rightarrow (E)$ { $stack[top - 2].val = stack[top - 1].val$;
 $top = top - 2;$ }

$F \rightarrow \text{digit}$



产生式内部带有语义动作的SDT

- 动作左边的所有符号（以及动作）处理完成后，就立刻执行这个动作
 - $B \rightarrow X\{a\}Y$
 - 自底向上分析时，在X出现在栈顶时执行动作a
 - 自顶向下分析时，在试图展开Y或者在输入中检测到Y的时刻执行a
- 不是所有的SDT都可以在分析过程中实现
- 但是后缀SDT以及L属性对应的SDT可以在分析时完成



有问题的SDT

□ 从中缀表达式到前缀表达式的转换

- 1) $L \rightarrow E \text{ n}$
- 2) $E \rightarrow \{ \text{print}('+' \text{); } \} \ E_1 + T$
- 3) $E \rightarrow T$
- 4) $T \rightarrow \{ \text{print}('*' \text{); } \} \ T_1 * F$
- 5) $T \rightarrow F$
- 6) $F \rightarrow (E)$
- 7) $F \rightarrow \text{digit} \ \{ \text{print}(\text{digit}.lexval \text{); } \}$

- 在自顶向下和自底向上分析中都无法实现
- 对于这种一般的SDT，都可以先建立分析树(语义动作作为虚拟的结点)，然后进行前序遍历并执行动作



消除左递归时SDT的转换方法

- 如果动作不涉及属性值，可以把动作当作终结符号进行处理，然后消左递归
- 原始的产生式
 - $E \rightarrow E_1 + T \quad \{print('+'\});\}$
 - $E \rightarrow T$
- 转换后得到
 - $E \rightarrow T R$
 - $R \rightarrow + T \quad \{print ('+'\});\} R$
 - $R \rightarrow \epsilon$



左递归文法翻译方案的转换

例 把带左递归的文法的翻译方案转换成不带左递归的文法的翻译方案。

$E \rightarrow E_1 + T \quad \{ E.val = E_1.val + T.val \}$

$E \rightarrow E_1 - T \quad \{ E.val = E_1.val - T.val \}$

$E \rightarrow T \quad \{ E.val = T.val \}$

$T \rightarrow (E) \quad \{ T.val = E.val \}$

$T \rightarrow num \quad \{ T.val = num.val \}$

带左递归的文法的翻译方案



经过转换的不带有左递归文法的翻译方案

$E \rightarrow T \{ R.i = T.val \}$

$R \{ E.val = R.s \}$

$R \rightarrow +$

$T \{ R_1.i = R.i + T.val \}$

$R_1 \{ R.s = R_1.s \}$

$R \rightarrow -$

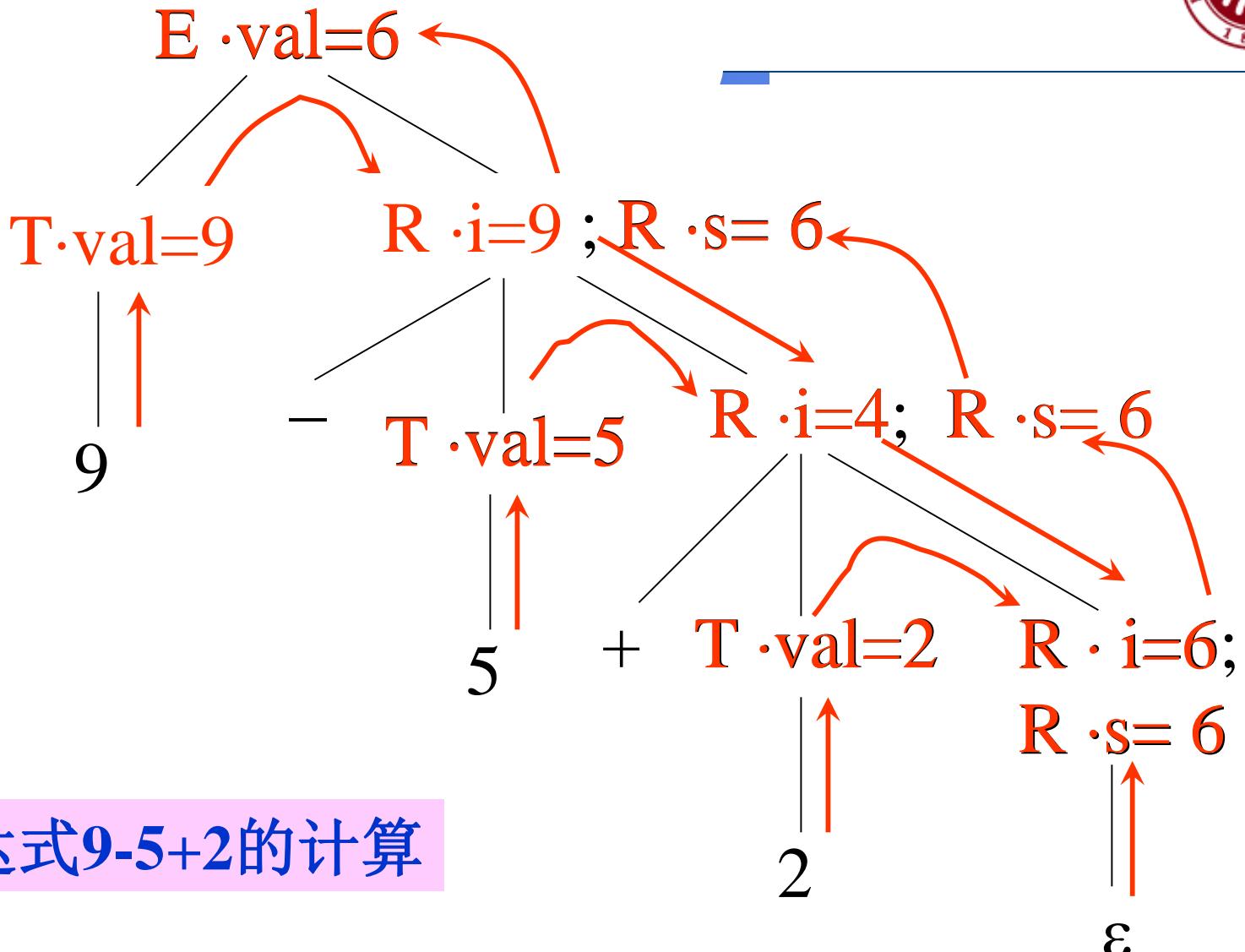
$T \{ R_1.i = R.i - T.val \}$

$R_1 \{ R.s = R_1.s \}$

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

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

$T \rightarrow \text{num} \{ T.val = \text{num}.val \}$





左递归翻译方案的一般化及消除

□ 左递归翻译方案

$A \rightarrow A_1 Y$

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

$A \rightarrow X$

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

- 假设每个文法符号有一个综合属性，用相应的小写字母表示， g 和 f 是任意函数

□ 消除左递归之后，文法转换成

$A \rightarrow X R$

$R \rightarrow Y R \mid \epsilon$

□ 消除左递归的翻译方案

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

$R \{ A.a = R.s \}$

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

$R_1 \{ R.s = R_1.s \}$

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

- 经过转换的翻译方案中使用了 R 的继承属性 i 和综合属性 s



输入: XY_1Y_2

$$A.a = g(g(f(X.x), Y_1.y), Y_2.y)$$

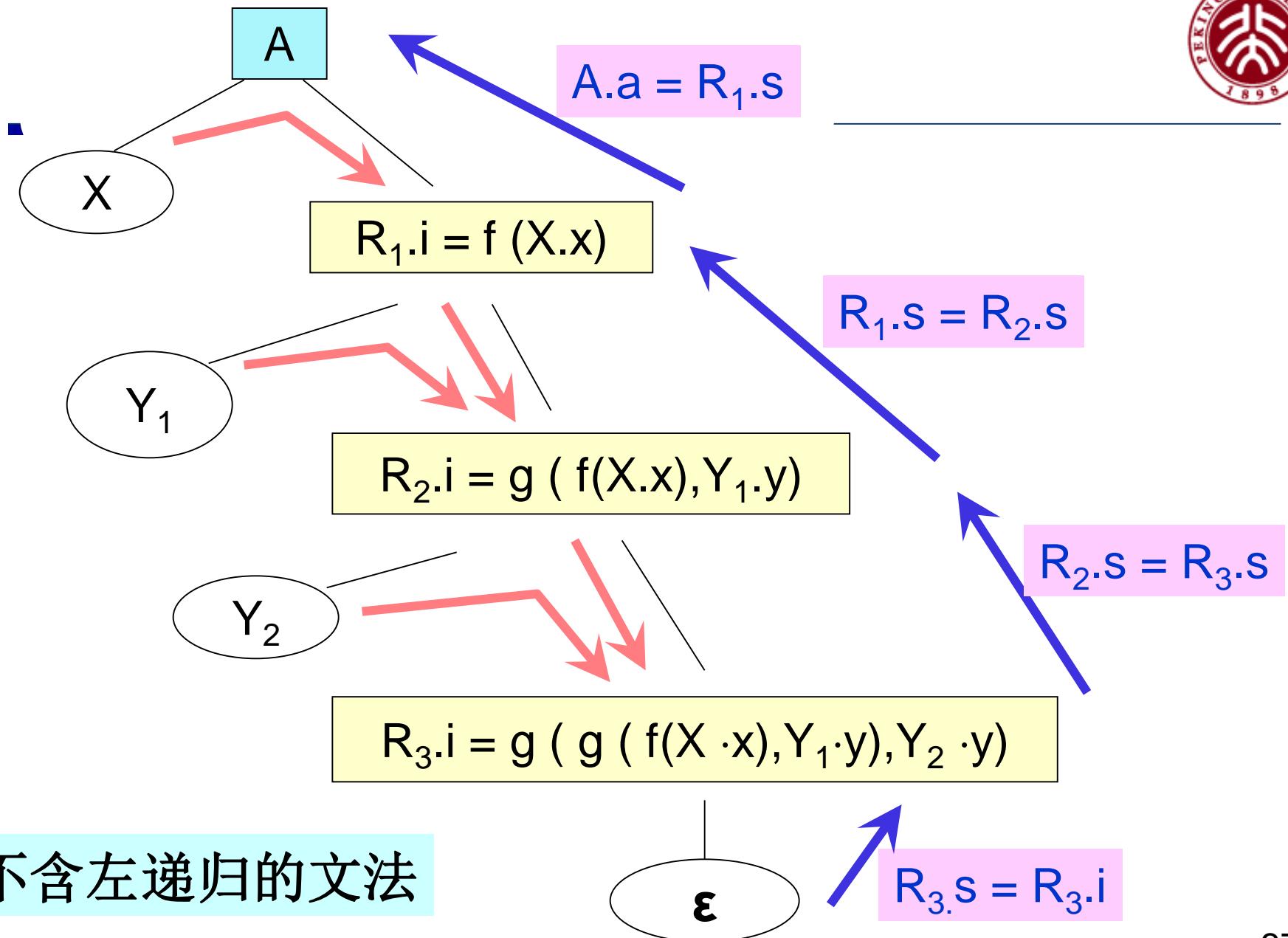
$$A.a = g(f(X.x), Y_1.y)$$

$$A.a = f(X.x)$$

X

Y₂

含左递归的文法



不含左递归的文法



L属性的SDT的例子

□ SDD

```
 $S \rightarrow \text{while } (C) S_1 \quad L1 = new();$ 
 $\qquad\qquad\qquad L2 = new();$ 
 $\qquad\qquad\qquad S_1.next = L1;$ 
 $\qquad\qquad\qquad C.false = S.next;$ 
 $\qquad\qquad\qquad C.true = L2;$ 
 $\qquad\qquad\qquad S.code = \text{label} \parallel L1 \parallel C.code \parallel \text{label} \parallel L2 \parallel S_1.code$ 
```

□ 继承属性：

- **next**: 语句结束后应该跳转到的标号
- **true**、**false**: C为真/假时应该跳转到的标号

□ 综合属性**code**表示代码



转换为SDT

```
 $S \rightarrow \text{while} ( \quad \{ L1 = new(); L2 = new(); C.false = S.next; C.true = L2; \}$ 
 $C ) \quad \{ S_1.next = L1; \}$ 
 $S_1 \quad \{ S.code = \text{label} \parallel L1 \parallel C.code \parallel \text{label} \parallel L2 \parallel S_1.code; \}$ 
```

□ 语义动作

- $L1 = \text{new}()$ 和 $L2 = \text{new}()$: 计算临时值
- $C.false = S.next; C.true = L2$: 计算C的继承属性
- $S_1.next = L1$: 计算 S_1 的继承属性
- $S.code = \dots$: 计算S的综合属型



作业

- 11月8日交
- 语法制导的翻译方案
 - Ex. 5.3.2
 - Ex. 5.4.3