



# 第5章 语法制导翻译技术



重庆大学 葛亮

知识点：语法制导定义、翻译方案  
S-属性定义、L-属性定义  
S-属性定义的翻译  
L-属性定义的翻译

# 语法制导翻译技术

## 语法制导翻译概述

- 5. 1 语法制导定义及翻译方案
  - 5. 2 **S**-属性定义的自底向上翻译
  - 5. 3 **L**-属性定义的自顶向下翻译
  - 5. 4 **L**-属性定义的自底向上翻译
  - 5. 5 通用的语法制导翻译方法
- 小结

# 语法制导翻译概述

- 源程序  $\xrightarrow[\text{语义相同}]{\text{等价转换}}$  目标程序
- 语义分析涉及到语言的语义
- 形式语义学
  - ◆ 采用形式系统的方法对形式语言及其程序进行语义定义的学问
  - ◆ 开始于20世纪60年代初
- 语法制导翻译技术
  - ◆ 多数编译程序普遍采用的一种技术
  - ◆ 比较接近形式化

# 语法制导翻译的整体思路

- 首先，根据翻译目标来确定每个产生式的语义；
- 其次，根据产生式的含义，分析每个符号的语义；
- 再次，把这些语义以属性的形式附加到相应的文法符号上（即把语义和语言结构联系起来）；
- 然后，根据产生式的语义给出符号属性的求值规则（即语义规则），从而形成语法制导定义。
- 翻译：  
根据语法分析过程中所使用的产生式，执行与之相应的语义规则，完成符号属性值的计算，从而完成翻译。

# 语法制导翻译示例

$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \text{digit}$	$F.val = \text{digit.val}$

例如：考虑算术表达式文法

- **翻译目标：计算表达式的值**
- 根据翻译目标确定每个产生式的语义；
  - ◆  $E \rightarrow E_1 + T$ ：表达式的值由两个子表达式的值相加得到
  - ◆  $F \rightarrow \text{digit}$ ：表达式的值即数字的值
- 根据产生式的语义，分析每个符号的语义；
  - ◆  $E$ 、 $T$ 、 $F$ 、 $\text{digit}$ 、 $+$ 、 $*$ 、 $($ 、 $)$
- 把这些语义以**属性**的形式附加到相应的文法符号上；
  - ◆  $E.val$ 、 $T.val$ 、 $F.val$ 、 $\text{digit.val}$
- 根据产生式的语义，给出符号属性的求值规则(即**语义规则**)，从而形成**语法制导定义**。
  - ◆  $E \rightarrow E_1 + T$ 对应的求值规则： $E.val = E_1.val + T.val$
  - ◆ 语法制导定义：产生式 语义规则

# 语法制导翻译示例（续）

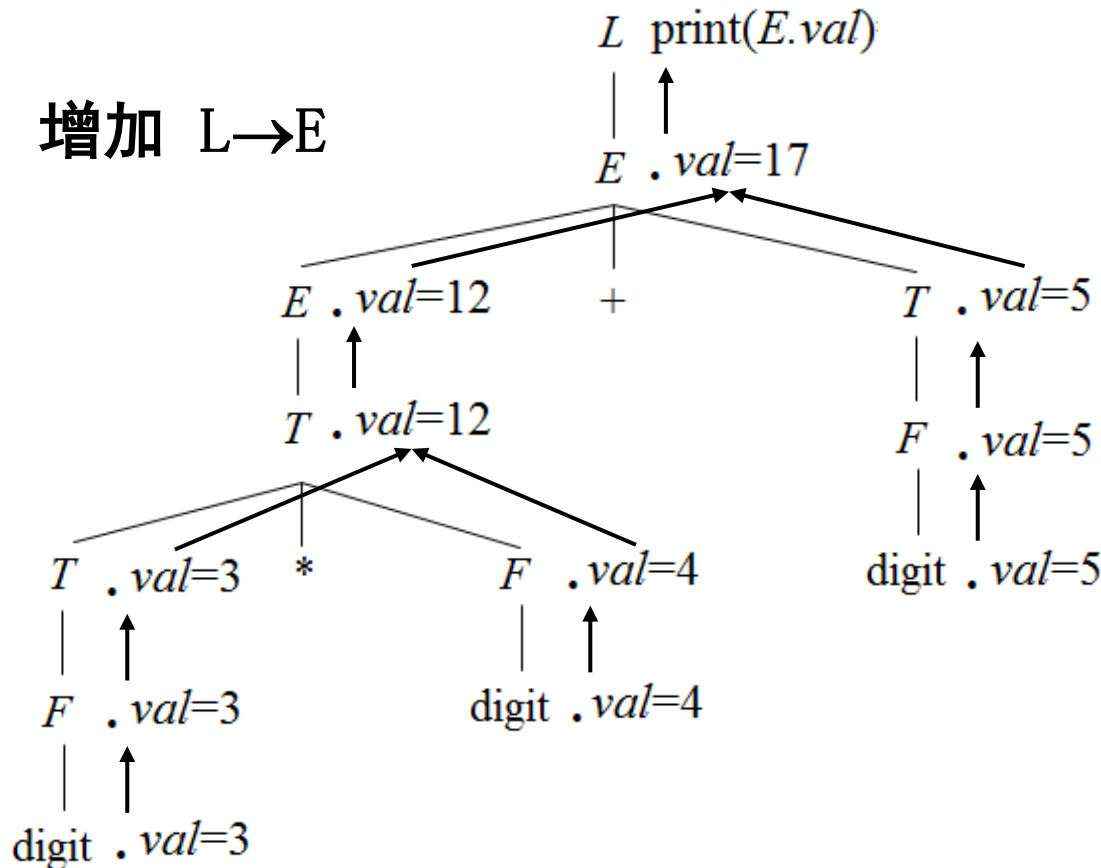
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \text{digit}$	$F.val = \text{digit.val}$

例如：考虑算术表达式文法

- 翻译目标：计算并打印表达式的值
- 根据语法分析过程中使用的产生式，执行相应的语义规则，完成相应的属性求值，从而完成翻译。
- 例如：3\*4+5

◆ 拓广文法：增加  $L \rightarrow E$

◆ 分析树：



# 翻译目标决定语义规则

- 翻译目标决定产生式的含义、决定文法符号应该具有的属性，也决定了产生式的语义规则。
- 例如：考虑算术表达式文法
  - ◆ 翻译目标：检查表达式的类型
  - ◆  $E \rightarrow E_1 + T$  的语义：表达式的类型由两个子表达式的类型综合得到
  - ◆ 分析每个符号的语义，并以属性的形式记录： **$E.type$** 、 **$E_1.type$** 、 **$T.type$**
  - ◆ 求值规则：  
if ( **$E_1.type == integer$** ) && ( **$T.type == integer$** )  
     **$E.type = integer$** ;  
else ...

# 翻译结果依赖于语义规则

## ■ 翻译目标

### ◆ 生成代码

- 可以为源程序产生中间代码
- 可以直接生成目标机指令

### ◆ 对输入符号串进行解释执行

### ◆ 向符号表中存放信息

### ◆ 给出错误信息

## ■ 翻译的结果依赖于语义规则

### ◆ 使用语义规则进行计算所得到的结果就是对输入符号串进行翻译的结果。

### ◆ 如： $E \rightarrow E + T$ 的翻译结果可以是：计算表达式的值、检查表达式的类型是否合法、为表达式创建语法树、生成代码等等。



# 语法制导翻译的一般步骤

输入符号串

————→ 分析树

————→ 依赖图

————→ 语义规则的计算顺序

————→ 计算结果

# 语义规则的执行时机

- 可以用一个或多个子程序（称为**语义动作**）所要完成的功能描述产生式的语义。
- 在**语法分析过程中**使用某个产生式时，在**适当的时机**执行相应的语义动作，完成所需要的翻译。
- 把语义动作**插入到产生式中适当的位置**，从而形成**翻译方案**。
- 语法制导定义是对翻译的高层次的说明，它隐蔽了一些实现细节，无须指明翻译时语义规则的计算次序。
- 翻译方案指明了语义规则的计算次序，规定了语义动作的执行时机。

# 5.1 语法制导定义及翻译方案

5.1.1 语法制导定义

5.1.2 依赖图

5.1.3 计算次序

5.1.4 S属性定义和L属性定义

5.1.5 翻译方案

# 5.1.1 语法制导定义

- 对上下文无关文法的推广
- 每个文法符号都可以有一个**属性集**，其中可以包括两类属性：**综合属性**和**继承属性**。
  - ◆ 左部符号的综合属性是从该产生式右部文法符号的属性值计算出来的；在分析树中，一个内部结点的**综合属性**是从其子结点的属性值计算出来的。
  - ◆ 出现在产生式右部的某文法符号的**继承属性**是从其所在产生式的左部非终结符号和/或右部文法符号的属性值计算出来的；在分析树中，一个结点的**继承属性**是从其兄弟结点和/或父结点的属性值计算出来的。
  - ◆ 分析树中某个结点的**属性值**是由与在这个结点上所用产生式相应的**语义规则决定的**。
- 和产生式相联系的**语义规则建立了属性之间的关系**，这些关系可用**有向图**（即：**依赖图**）来表示。

# 语法制导定义

在一个语法制导定义中，对应于每一个文法产生式  $A \rightarrow \alpha$ ，都有与之相联系的一组语义规则，其形式为：  
 $b = f(c_1, c_2, \dots, c_k)$

这里， $f$ 是一个函数，而且

- (1) 如果 $b$ 是 $A$ 的一个综合属性，则  $c_1$ 、 $c_2$ 、...、 $c_k$ 是产生式右部文法符号的属性或者 $A$ 的继承属性；
- (2) 如果 $b$ 是产生式右部某个文法符号的一个继承属性，则  $c_1$ 、 $c_2$ 、...、 $c_k$ 是 $A$ 或产生式右部任何文法符号的属性。

■ 属性 $b$ 依赖于属性 $c_1$ 、 $c_2$ 、...、 $c_k$ 。

# 语义规则

## ■ 一般情况：

- ◆ 语义规则函数可写成表达式的形式。
- ◆ 比如：  $E.val = E_1.val + T.val$

## ■ 某些情况下：

- ◆ 一个语义规则的唯一目的就是产生某个副作用，如打印一个值、向符号表中插入一条记录等；
- ◆ 这样的语义规则通常写成过程调用或程序段。
- ◆ 看成是相应产生式左部非终结符号的虚拟综合属性。
- ◆ 虚属性和符号 ‘=’ 都没有表示出来。
- ◆ 比如： `print(E.val)`

## ■ 语义规则函数都不具有副作用的语法制导定义称为属性文法

# 简单算术表达式求值的语法制导定义

产生式	语义规则
$L \rightarrow E$	<b>print(E.val)</b>
$E \rightarrow E_1 + T$	<b>E.val = E<sub>1</sub>.val + T.val</b>
$E \rightarrow T$	<b>E.val = T.val</b>
$T \rightarrow T_1 * F$	<b>T.val = T<sub>1</sub>.val * F.val</b>
$T \rightarrow F$	<b>T.val = F.val</b>
$F \rightarrow (E)$	<b>F.val = E.val</b>
$F \rightarrow \text{digit}$	<b>F.val = digit.lexval</b>

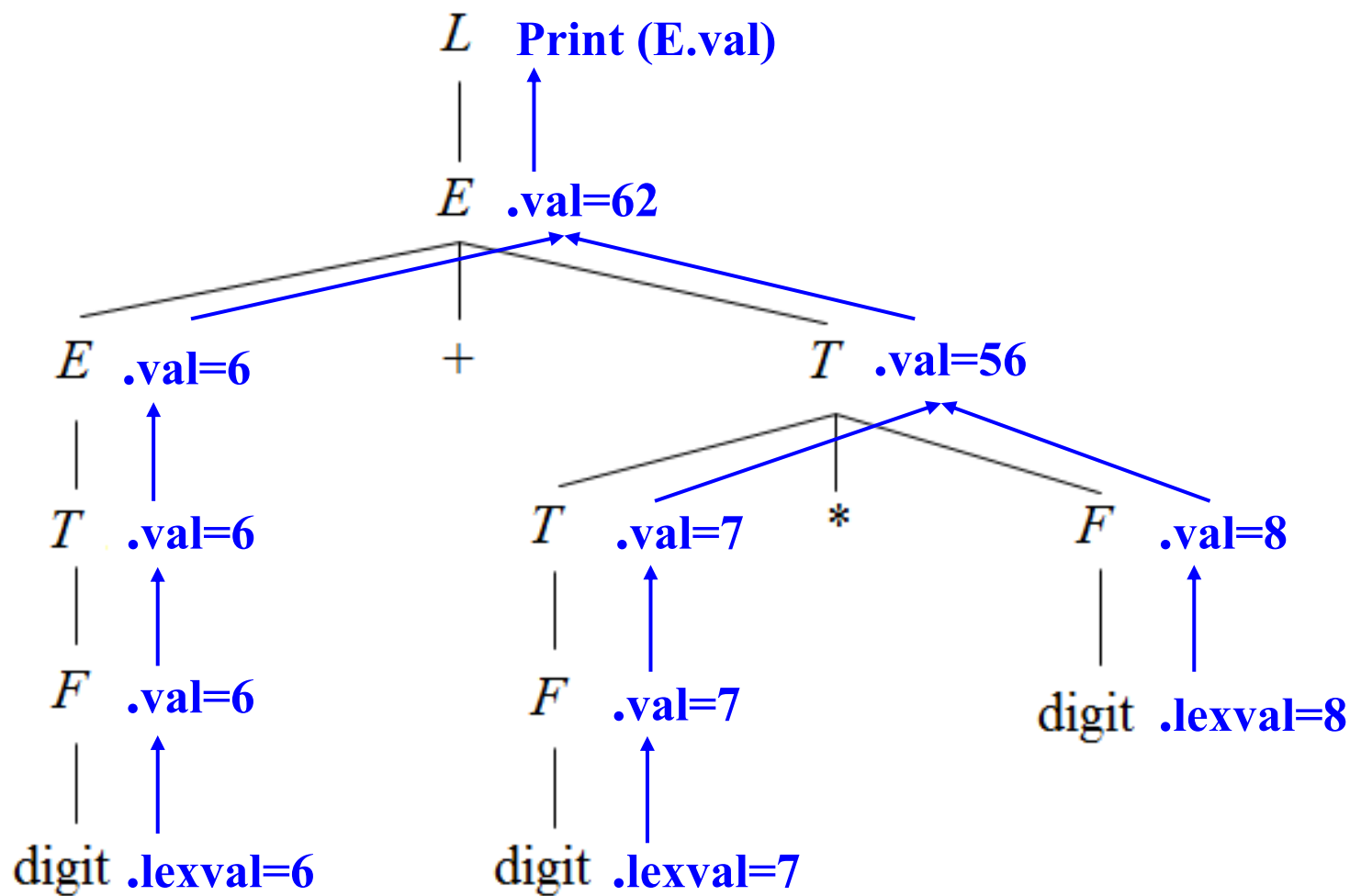
- **综合属性**val与每一个非终结符号E、T、F相联系
- 表示相应非终结符号所代表的子表达式的整数值
- $L \rightarrow E$ 的语义规则是一个过程，打印出由E产生的算术表达式的值，可以认为是非终结符号**L**的一个**虚拟综合属性**。

# 综合属性

- 分析树中，如果一个结点的某一属性由其子结点的属性确定，则这种属性为该结点的**综合属性**。
- 如果一个语法制导定义仅仅使用综合属性，则称这种语法制导定义为**S-属性定义**。
- 对于**S-属性定义**，通常采用**自底向上**的方法对其分析树加注释，即从树叶到树根，按照语义规则计算每个结点的属性值。
- 简单台式计算机的语法制导定义是**S-属性定义**



# 6+7\*8的分析树加注释的过程



- 属性值的计算可以在语法分析过程中进行。

# 继承属性

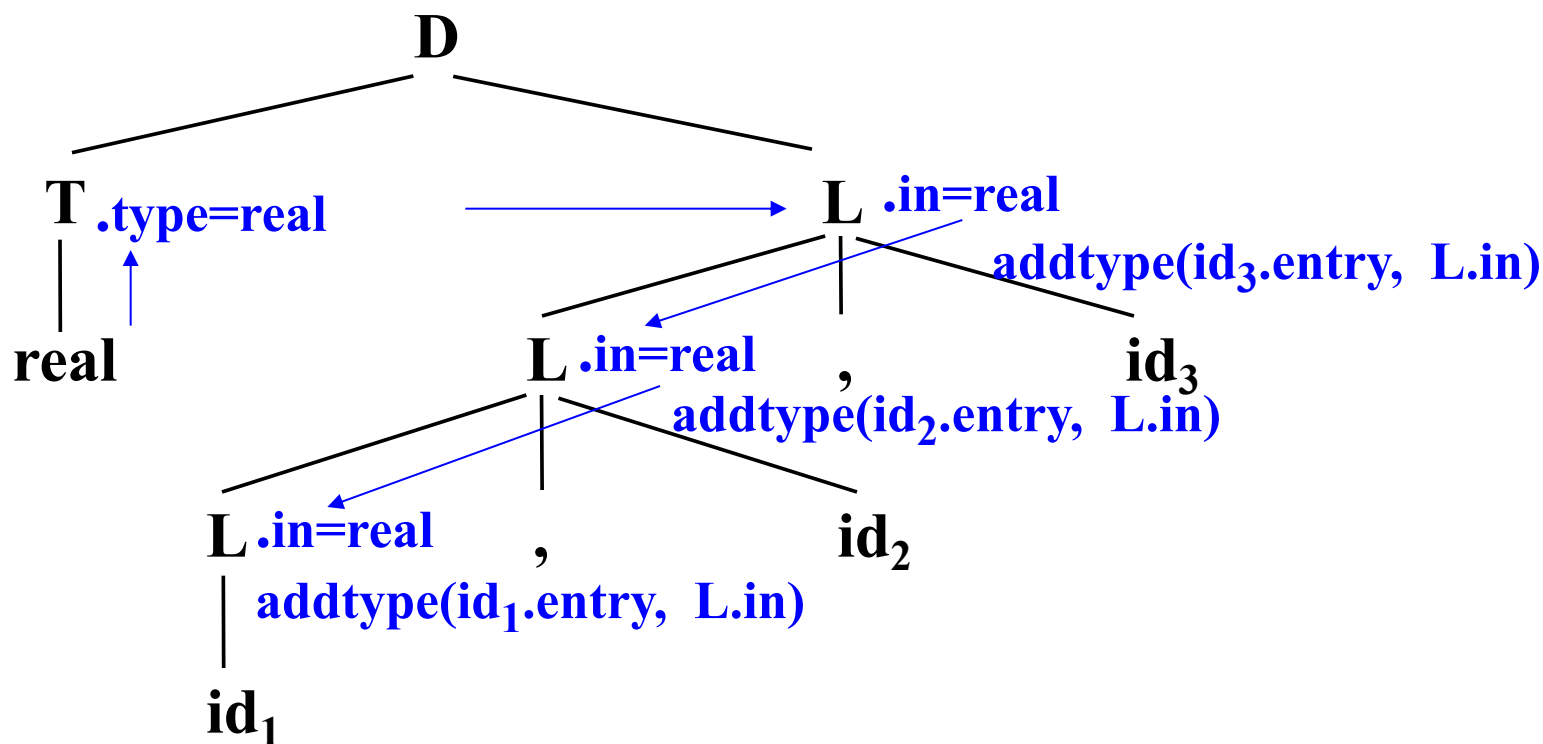
- 分析树中，一个结点的**继承属性**值由该结点的父结点和/或它的兄弟结点的属性值决定。
- 可用继承属性表示程序设计语言结构中**上下文之间的依赖关系**
  - ◆ 可以跟踪一个标识符的类型
  - ◆ 可以跟踪一个标识符，了解它是出现在赋值号的右边还是左边，以确定是需要该标识符的值还是地址。

# 用继承属性L.in传递类型信息的语法制导定义

产生式	语义规则
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in$ $addtype(id.entry, L.in)$
$L \rightarrow id$	$addtype(id.entry, L.in)$

- D产生的声明语句包含了类型关键字int或real，后跟一个标识符表。
- T有综合属性type，其值由声明中的关键字确定。
- L的继承属性L.in，
  - ◆ 产生式 $D \rightarrow TL$       L.in 表示从其兄弟结点T继承下来的类型信息。
  - ◆ 产生式 $L \rightarrow L_1, id$        $L_1.in$  表示从其父结点L继承下来的类型信息

# 语句 $\text{real id}_1, \text{id}_2, \text{id}_3$ 的注释分析树



L产生式的语义规则使用继承属性L.in把类型信息在分析树中向下传递；

并通过调用过程addtype，把类型信息填入标识符在符号表中相应的表项中。

## 5.1.2 依赖图

- 分析树中，结点的继承属性和综合属性之间的相互依赖关系可以由依赖图表示。
- 为每个包含过程调用的语义规则引入一个**虚拟综合属性** $b$ ，以便把语义规则统一为 $b=f(c_1, c_2, \dots, c_k)$ 的形式。
- 依赖图中：
  - ◆ 为每个属性设置一个结点
  - ◆ 如果属性 $b$ 依赖于 $c$ ，那么从属性 $c$ 的结点有一条有向边连到属性 $b$ 的结点。

## 算法5.1 构造依赖图

输入：一棵分析树

输出：一张依赖图

方法：

for (分析树中每一个结点n)

for (结点n处的文法符号的每一个属性a)

为a在依赖图中建立一个结点;

for (分析树中每一个结点n)

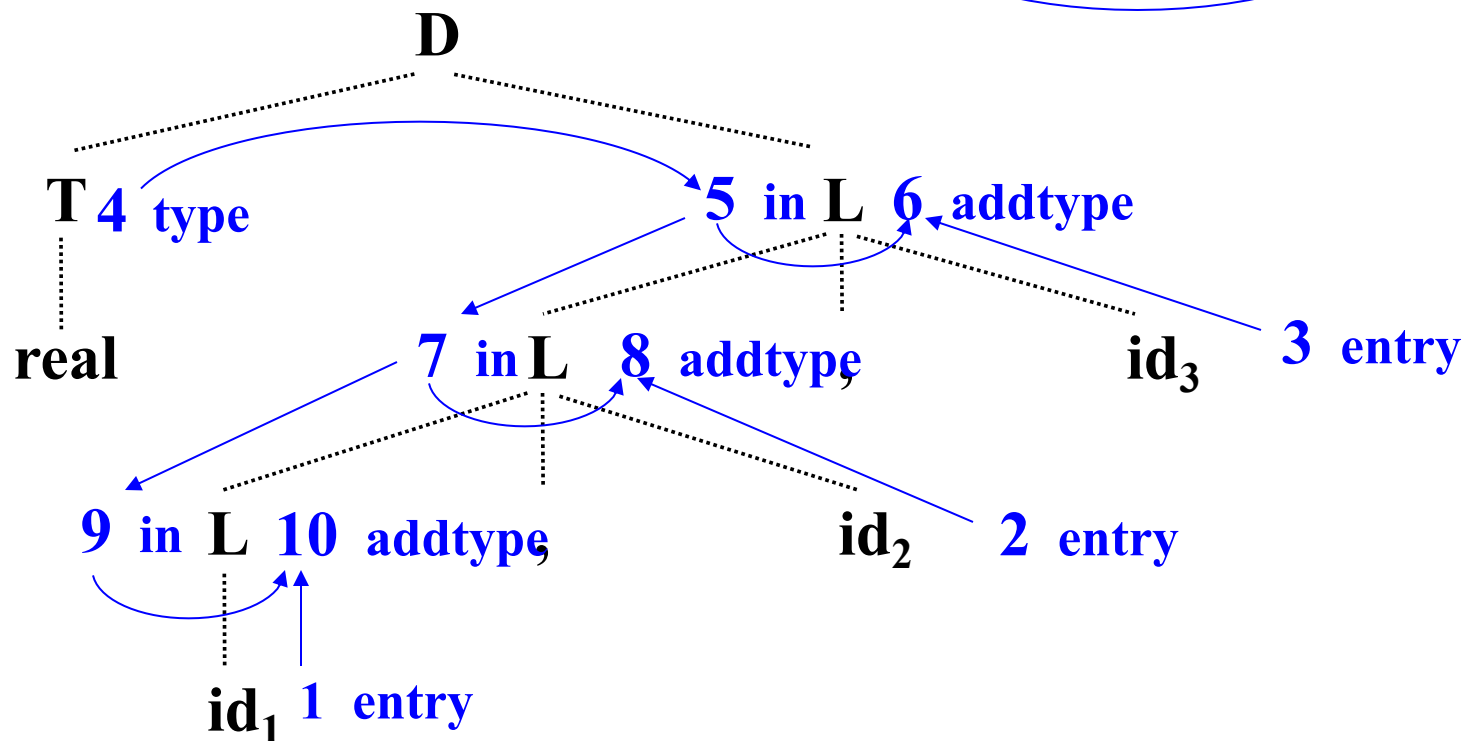
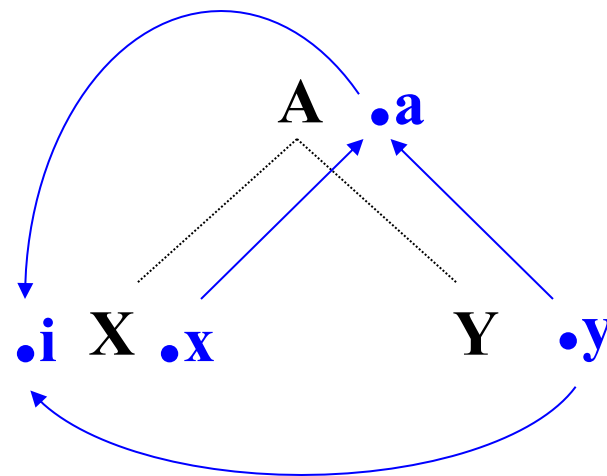
for (结点n处所用产生式对应的每一个语义规则

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

for (i=1; i≤k; i++)

从 $c_i$ 结点到b结点构造一条有向边;

Liang GE CAU

$$X.i = g(A.a, Y.y)$$


## 5.1.3 计算次序

### ■ 有向非循环图的拓扑排序

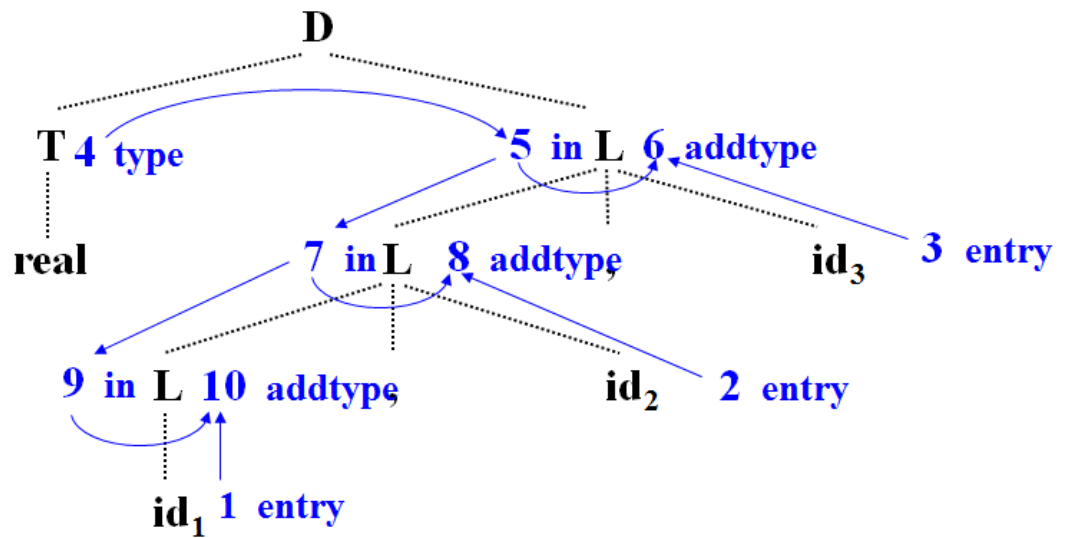
- ◆ 图中结点的一种排序  $m_1, m_2, \dots, m_k$
- ◆ 有向边只能从这个序列中前面的结点指向后面的结点
- ◆ 如果  $m_i \rightarrow m_j$  是从  $m_i$  指向  $m_j$  的一条边，那么在序列中  $m_i$  必须出现在  $m_j$  之前。

### ■ 依赖图的任何拓扑排序

- ◆ 给出了分析树中结点的语义规则计算的有效顺序
- ◆ 在拓扑排序中，一个结点上语义规则  $b=f(c_1, c_2, \dots, c_k)$  中的属性  $c_1, c_2, \dots, c_k$  在计算  $b$  时都是可用的。



# 计算顺序



**type=real;**

**in<sub>5</sub>=type;**

**addtype(id<sub>3</sub>.entry, in<sub>5</sub>);**

**in<sub>7</sub>=in<sub>5</sub>;**

**addtype(id<sub>2</sub>.entry, in<sub>7</sub>);**

**in<sub>9</sub>=in<sub>7</sub>;**

**addtype(id<sub>1</sub>.entry, in<sub>9</sub>);**

## ■ 拓扑排序:

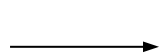
- **1、2、3、4、5、6、7、8、9、10**
- **4、5、3、6、7、2、8、9、1、10**

■  **$a_n$ 代表依赖图中与序号n的结点有关的属性a**

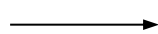
# 语法制导翻译过程

- 基础文法用于建立输入符号串的分析树；
- 为分析树构造依赖图；
- 对依赖图进行拓扑排序；
- 从这个序列得到语义规则的计算顺序；
- 照此计算顺序进行求值，得到对输入符号串的翻译。

输入符号串



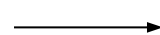
分析树



依赖图



语义规则的计算顺序



计算结果

## 5.1.4 S属性定义和L属性定义

- S属性定义：仅涉及综合属性的语法制导定义
- L属性定义：一个语法制导定义是**L属性定义**，如果
  - ◆ 与每个产生式  $A \rightarrow X_1 X_2 \dots X_n$  相应的每条语义规则计算的属性都是**A的综合属性**，或是  $X_j$  ( $1 \leq j \leq n$ ) 的**继承属性**，而该继承属性**仅依赖于**：
    - A的继承属性；
    - 产生式中 $X_j$ 左边的符号 $X_1$ 、 $X_2$ 、...、 $X_{j-1}$ 的属性；
- 每一个S属性定义都是L属性定义

# 语法制导定义示例：

例：非L属性定义 →

产生式	语义规则
$A \rightarrow LM$	$L.i = 1(A.i)$ $M.i = m(L.s)$ $A.s = f(M.s)$
$A \rightarrow QR$	$R.i = r(A.i)$ $Q.i = q(R.s)$ $A.s = f(Q.s)$

例：L属性定义 →

产生式	语义规则
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in$ $addtype(id.entry, L.in)$
$L \rightarrow id$	$addtype(id.entry, L.in)$

# 属性计算顺序——深度优先遍历分析树

```
void deepfirst (n: node)
{
    for (n的每一个子结点m, 从左到右) {
        计算m的继承属性;
        deepfirst(m);
    };
    计算n的综合属性;
}.
```

- 以分析树的根结点作为**实参**
- L属性定义的属性都可以用**深度优先的顺序**计算。
  - ◆ **进入**结点前, 计算它的继承属性
  - ◆ 从结点**返回**时, 计算它的综合属性

## 5.1.5 翻译方案

- 上下文无关文法的一种便于翻译的书写形式
- 属性与文法符号相对应
- 语义动作括在花括号中，并插入到产生式右部某个合适的位置上
- 给出了使用语义规则进行属性计算的顺序
- 分析过程中翻译的注释

# 翻译方案示例

一个简单的翻译方案:

$E \rightarrow TR$

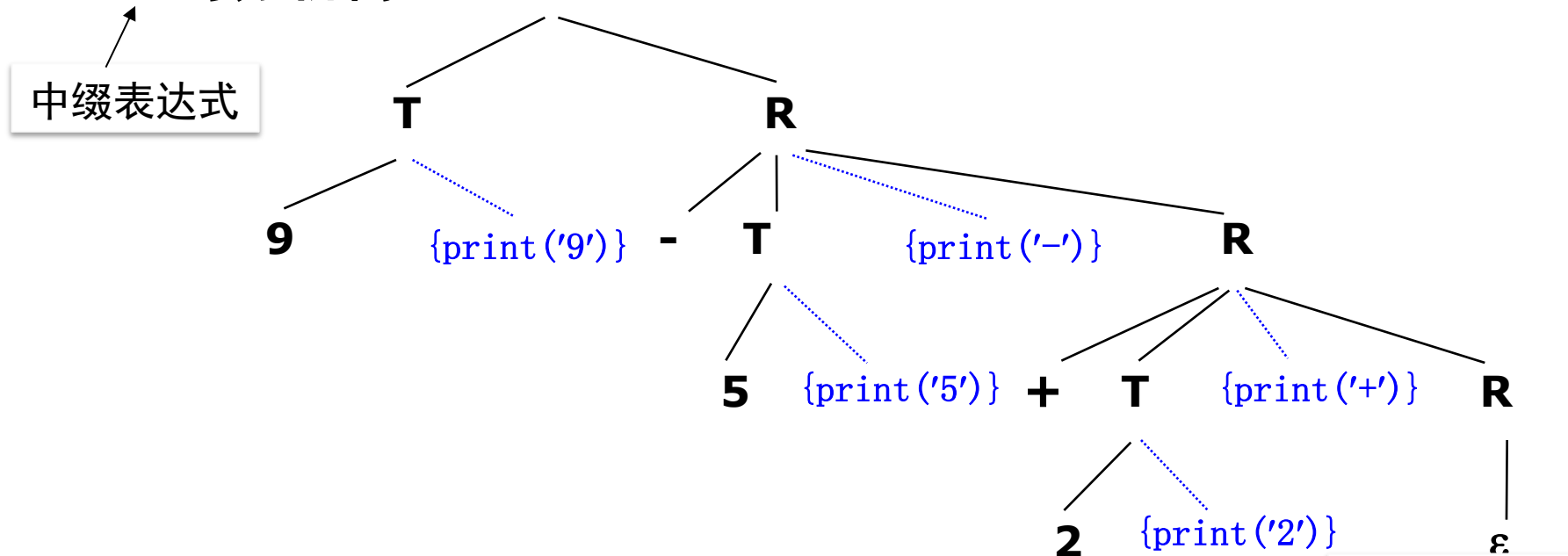
$R \rightarrow +T \{ \text{print}('+') \} R_1$

$\quad \quad \quad | -T \{ \text{print}('-') \} R_1$

$\quad \quad \quad | \varepsilon$

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

9-5+2的分析树:



语义动作作为相应产生式左部符号对应结点的子结点

深度优先遍历树中结点, 执行其中的动作, 打印出**95-2+**

# 翻译方案的设计

## ■ 对于S属性定义：

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

例：产生式      语义规则

$T \rightarrow T_1 * F$        $T.val = T_1.val * F.val$

如下安排产生式和语义动作：

$T \rightarrow T_1 * F \{ T.val = T_1.val * F.val \}$



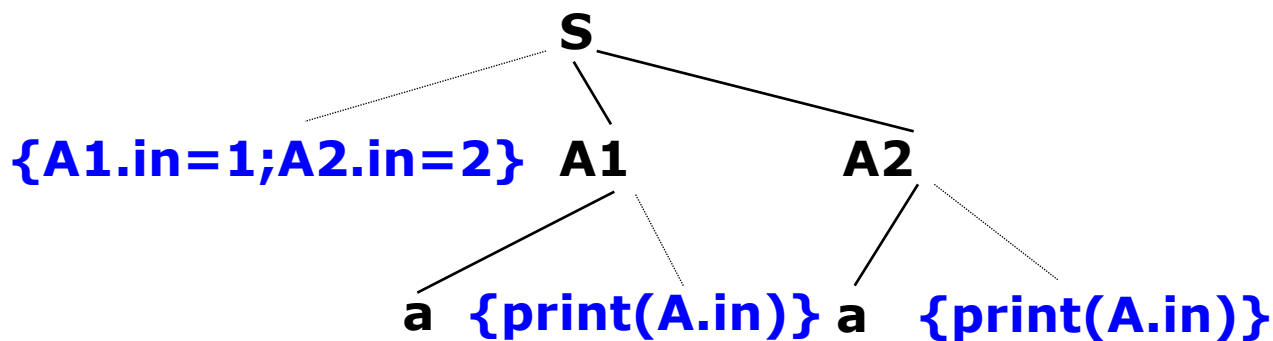
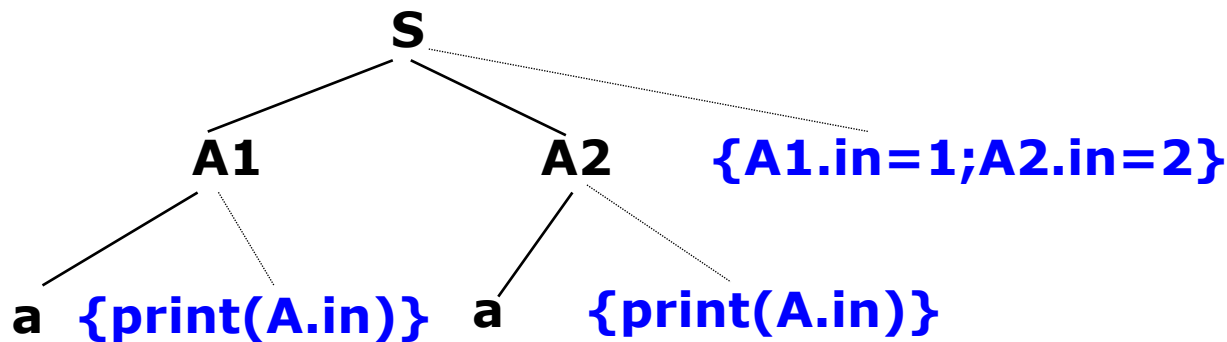
# 为L属性定义设计翻译方案的原则

- 产生式右部文法符号的**继承属性**必须在这个符号以前的动作中计算出来
  - ◆ 计算该继承属性的动作必须出现在相应文法符号之前
- 一个动作不能引用这个动作右边的文法符号的综合属性
- 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之后才能计算
  - ◆ 这种属性的计算动作放在产生式右端末尾

**示例：** 考虑如下翻译方案：

$S \rightarrow A_1 A_2 \{ A_1.in=1; A_2.in=2 \}$

$A \rightarrow a \{ \text{print}(A.in) \}$



# L属性定义翻译方案设计举例

## ■ 语法制导定义

产生式	语义规则
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in$ $addtype(id.entry, L.in)$
$L \rightarrow id$	$addtype(id.entry, L.in)$

## ■ 翻译方案

$D \rightarrow T \{ L.in = T.type \} L$

$T \rightarrow int \{ T.type = integer \}$

$T \rightarrow real \{ T.type = real \}$

$L \rightarrow \{ L_1.in = L.in \} L_1, id \{ addtype(id.entry, L.in) \}$

$L \rightarrow id \{ addtype(id.entry, L.in) \}$

# 练习

- 文法G的产生式如下：

$$S \rightarrow (L)$$

$$S \rightarrow a$$

$$L \rightarrow L, S$$

$$L \rightarrow S$$

- 试写出一个语法制导定义，输出配对括号个数。

## ■ 为S, L引入属性h

产生式

语法制导定义

$S \rightarrow (L)$

$S.h = L.h + 1$

$S \rightarrow a$

$S.h = 0$

$L \rightarrow L_1, S$

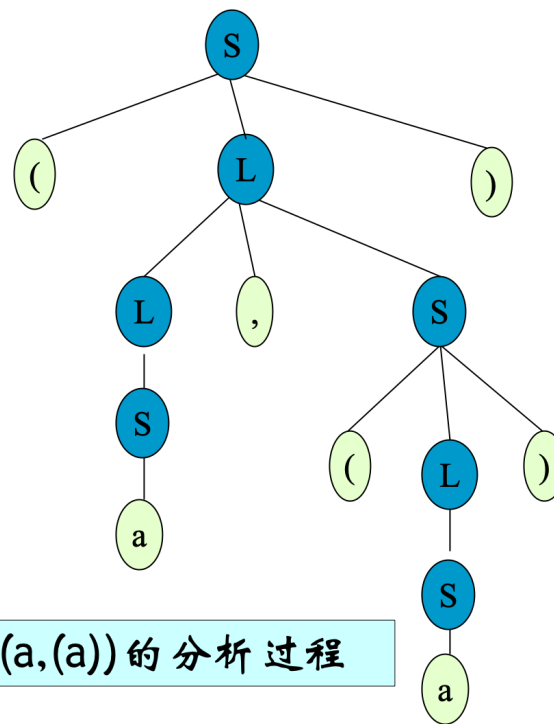
$L.h = L_1.h + S.h$

$L \rightarrow S$

$L.h = S.h$

$S' \rightarrow S$

print (S.h)



# 练习

- 文法G的产生式如下：

$$S \rightarrow (L)$$

$$S \rightarrow a$$

$$L \rightarrow L, S$$

$$L \rightarrow S$$

- 试写出一个翻译方案，打印每个a的嵌套深度。

■ 为S, L引入属性d, 翻译方案如下

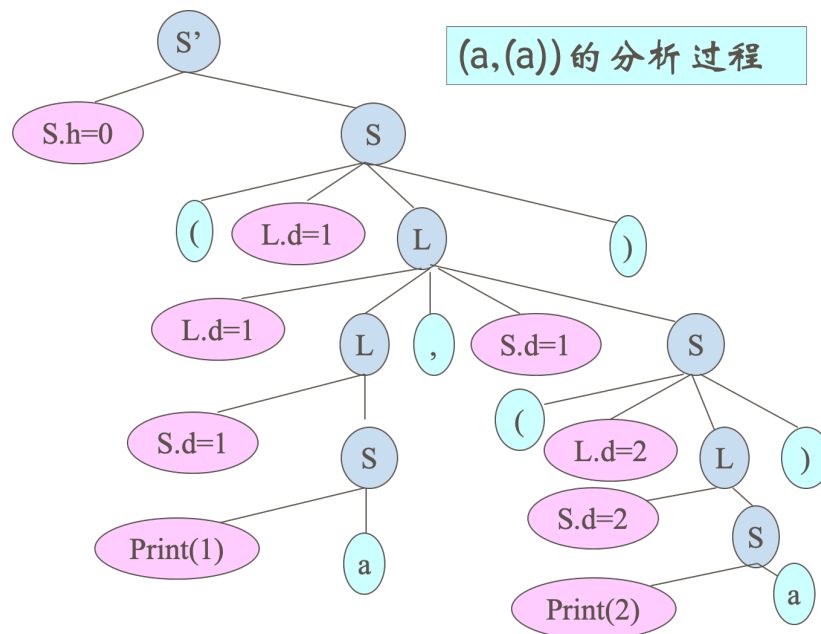
$S' \rightarrow \{S.d=0\} S$

$S \rightarrow (\{L.d=S.d+1\} L)$

$S \rightarrow a \{print(S.d)\}$

$L \rightarrow \{L_1.d=L.d\} L_1, \{S.d=L.d\} S$

$L \rightarrow \{S.d=L.d\} S$



## 5.2 S-属性定义的自底向上翻译

### ■ S属性定义：

- ◆ 只用综合属性的语法制导定义

5.2.1 为表达式构造语法树的语法制导定义

5.2.2 S属性定义的自底向上实现

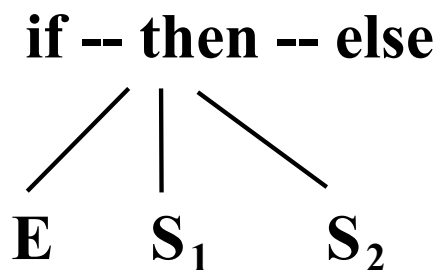


## 5.2.1 为表达式构造语法树的语法制导定义

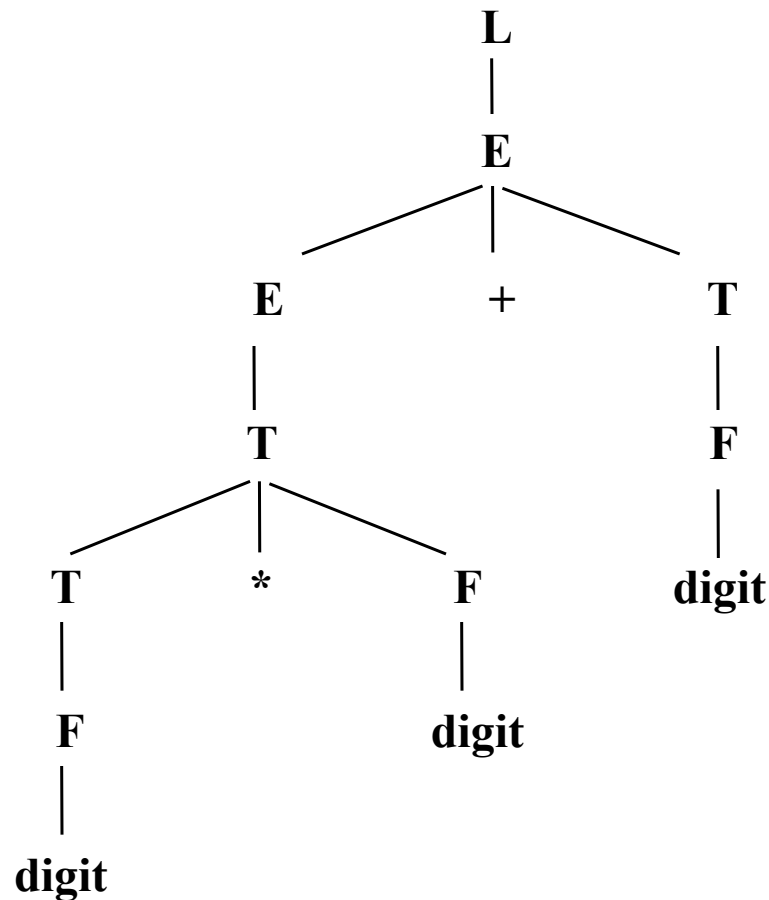
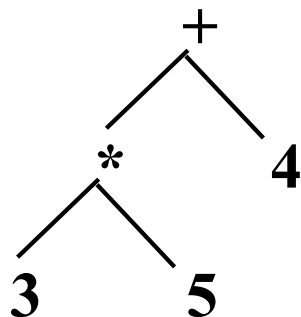
- 抽象语法：把语法规则中对语义无关紧要的具体规定去掉，剩下下来的本质性的东西称为抽象语法。
- 如：
  - ◆ 赋值语句： $x=y$ 、 $x:=y$ 、或  $y \rightarrow x$
  - ◆ 抽象形式：`assignment(variable,expression)`
- 语法树：
  - ◆ 分析树的抽象（或压缩）形式。
  - ◆ 也称为语法结构树或结构树。
  - ◆ 内部结点表示运算符号，其子结点表示它的运算分量。

# 语法树示例

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



- 表达式  $3*5+4$  的语法树



# 构造表达式的语法树

## ■ 表达式的语法树的形式

- ◆ 每一个运算符或运算分量都对应树中的一个结点
- ◆ 运算符结点的子结点分别是与该运算符的各个运算分量相应的子树的根。
- ◆ 每一个结点可包含若干个域：
  - 标识域、指针域、属性值域等

## ■ 在运算符结点中

- ◆ 一个域标识运算符
- ◆ 其它各域包含指向与各运算分量相应的结点的指针
- ◆ 称运算符为该结点的标号

# 构造函数

## ■ makenode (op, left, right)

- ◆ 建立一个运算符结点，标号是 op；
- ◆ 域left和right是指向其左右运算分量结点的指针。

## ■ makeleaf (id, entry)

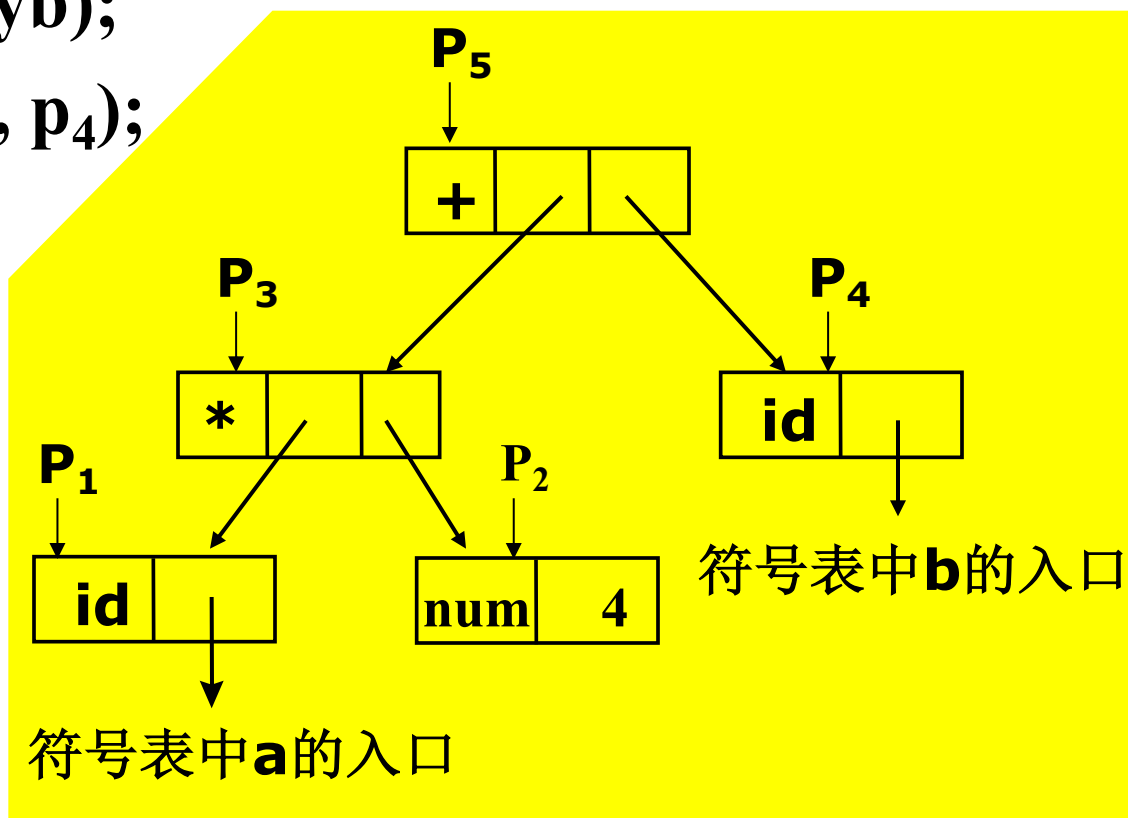
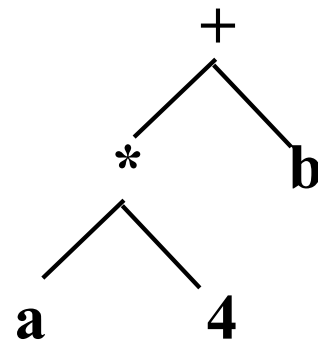
- ◆ 建立一个标识符结点，标号是 id；
- ◆ 域entry是指向该标识符在符号表中的相应条目的指针。

## ■ makeleaf (num, val)

- ◆ 建立一个数结点，标号为 num；
- ◆ 域val用于保存该数的值。

# 建立表达式 $a * 4 + b$ 的语法树

```
p1=makeleaf(id, entrya);  
p2=makeleaf(num, 4);  
p3=makenode('*', p1, p2);  
p4=makeleaf(id, entryb);  
p5=makenode('+', p3, p4);
```



# 构造表达式语法树的语法制导定义

- **翻译目标：为表达式创建语法树**
- **产生式语义：创建与产生式左部符号代表的子表达式对应的子树，即创建子树的根结点。**
- **文法符号的属性：记录所建结点，E.nptr、T.nptr、F.nptr 指向相应子树根结点的指针**
- **产生式的语义动作举例：**

$E \rightarrow E_1 + T$      $E.nptr = \text{makenode}('+', E_1.nptr, T.nptr)$

$T \rightarrow F$          $T.nptr = F.nptr$

$F \rightarrow id$          $F.nptr = \text{makeleaf}(id, id.entry)$

$F \rightarrow num$        $F.nptr = \text{makeleaf}(num, num.val)$

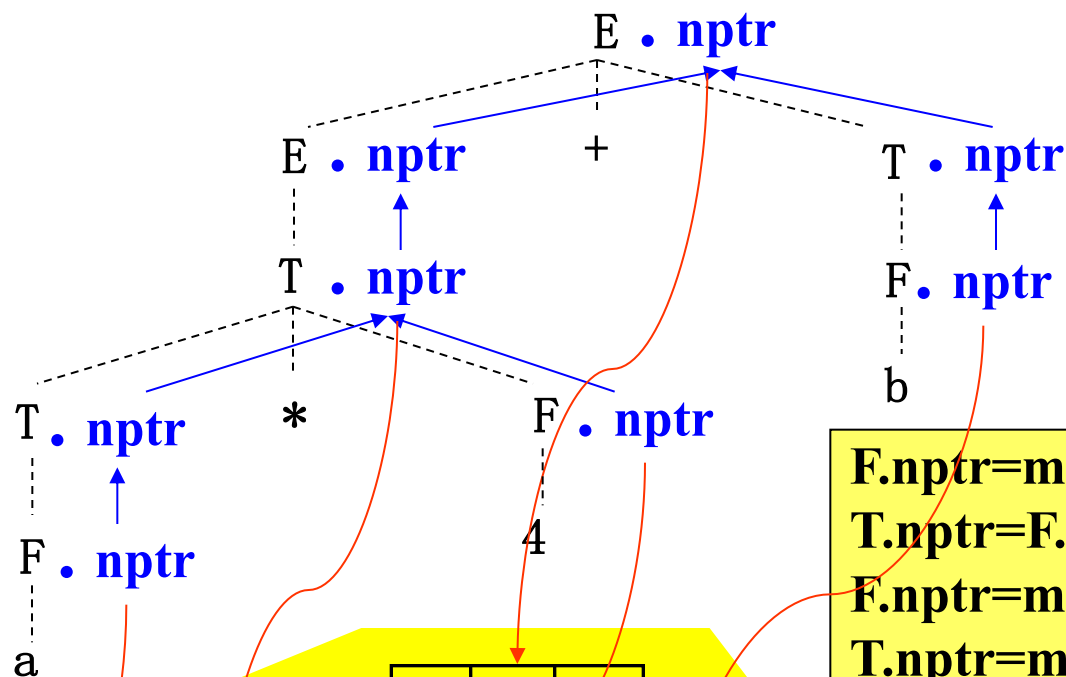
# 构造表达式语法树的语法制导定义（续）

产生式	语义规则
$E \rightarrow E_1 + T$	$E.nptr = \text{makenode}('+', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow T_1 * F$	$T.nptr = \text{makenode}('*', T_1.nptr, F.nptr)$
$T \rightarrow F$	$T.nptr = F.nptr$
$F \rightarrow (E)$	$F.nptr = E.nptr$
$F \rightarrow \text{id}$	$F.nptr = \text{makeleaf}(\text{id}, \text{id.entry})$
$F \rightarrow \text{num}$	$F.nptr = \text{makeleaf}(\text{num}, \text{num.val})$

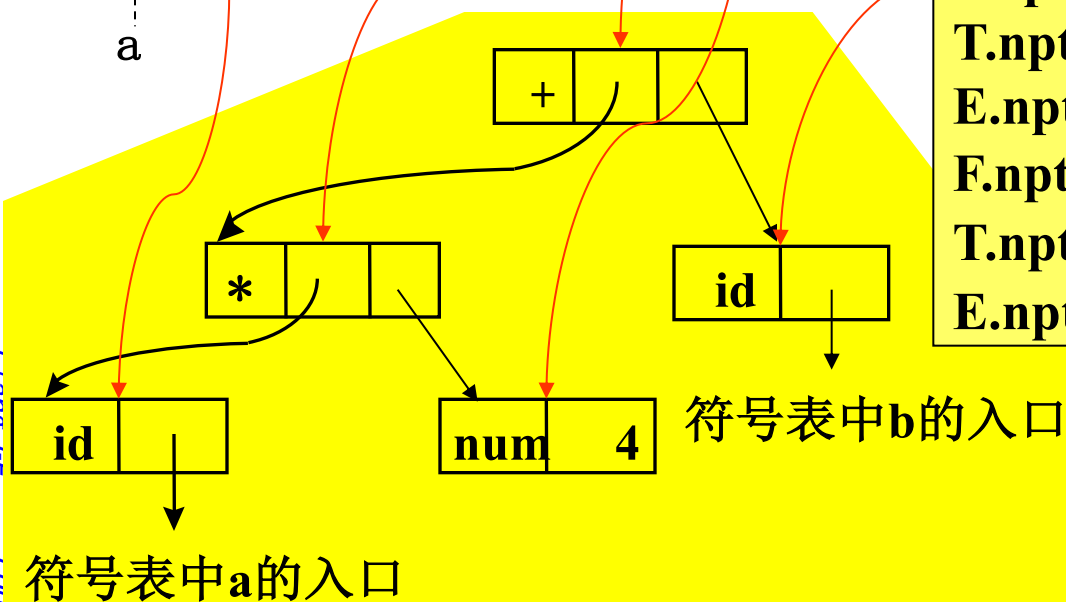
- 为了记录在构造过程中建立的子树，为每个非终结符号引入一个综合属性 nptr。
- nptr 是一个指针，指向语法树中相应非终结符号产生的表达式子树的根结点。

# 表达式 $a * 4 + b$ 的语法树的构造

$E \rightarrow E_1 + T$   
 $E \rightarrow T$   
 $T \rightarrow T_1 * F$   
 $T \rightarrow F$   
 $F \rightarrow (E)$   
 $F \rightarrow id$   
 $F \rightarrow num$



$F.nptr = makeleaf(id, entrya)$   
 $T.nptr = F.nptr$   
 $F.nptr = makeleaf(num, 4)$   
 $T.nptr = makenode('*', T_1.nptr, F.nptr)$   
 $E.nptr = T.nptr$   
 $F.nptr = makeleaf(id, entryb)$   
 $T.nptr = F.nptr$   
 $E.nptr = makenode('+', E_1.nptr, T.nptr)$





# 表达式的有向非循环图(dag)

## ■ dag与语法树相同的地方：

- ◆ 表达式的每一个子表达式都有一个结点
- ◆ 一个内部结点表示一个运算符号，且它的子结点表示它的运算分量。

## ■ dag与语法树不同的地方：

- ◆ dag中，对应一个公共子表达式的结点具有多个父结点
- ◆ 语法树中，公共子表达式被表示为重复的子树

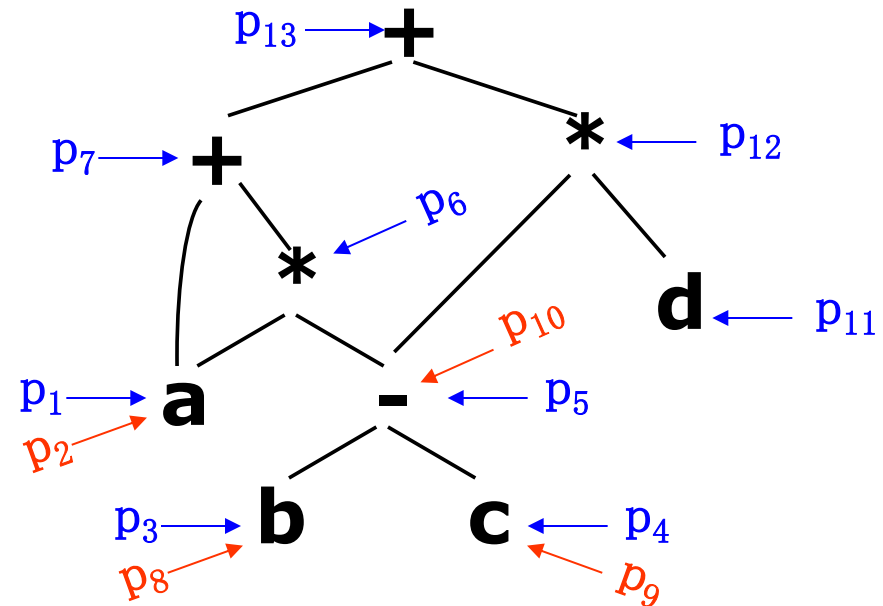
## ■ 为表达式创建dag的函数makenode和makeleaf

- ◆ 建立新结点之前先检查是否已经存在一个相同的结点
- ◆ 若已存在，返回一个指向先前已构造好的结点的指针；
- ◆ 否则，创建一个新结点，返回指向新结点的指针。

# 为表达式 $a + a * (b - c) + (b - c) * d$ 构造dag

## 函数调用

- $p_1 = \text{makeleaf}(\text{id}, a);$
- $p_2 = \text{makeleaf}(\text{id}, a);$
- $p_3 = \text{makeleaf}(\text{id}, b);$
- $p_4 = \text{makeleaf}(\text{id}, c);$
- $p_5 = \text{makenode}('-', p_3, p_4);$
- $p_6 = \text{makenode}('*', p_2, p_5);$
- $p_7 = \text{makenode}('+', p_1, p_6);$
- $p_8 = \text{makeleaf}(\text{id}, b);$
- $p_9 = \text{makeleaf}(\text{id}, c);$
- $p_{10} = \text{makenode}('-', p_8, p_9);$
- $p_{11} = \text{makeleaf}(\text{id}, d);$
- $p_{12} = \text{makenode}('*', p_{10}, p_{11});$
- $p_{13} = \text{makenode}('+', p_7, p_{12});$



## 5.2.2 S-属性定义的自底向上实现

### ■ 已知

- ◆ LR分析方法中，分析程序使用一个**栈**来存放已经分析过的子树的信息。
- ◆ 分析树中某结点的**综合属性**由其子结点的属性值计算得到
- ◆ LR分析程序在分析输入符号串的**同时**可以计算综合属性

### ■ 考虑

- ◆ 如何**保存**文法符号的综合**属性值**？
- ◆ 保存属性值的**数据结构**怎样与**分析栈**相联系？
- ◆ **怎样保证**：每当进行归约时，由栈中正在归约的产生式右部符号的属性值计算其左部符号的综合属性值。

# 修改分析栈

目的：使之能够保存综合属性

做法：在分析栈中**增加一个域**，存放综合属性值

例：带有综合属性域的分析栈

top →

Sz	Z.z
Sy	Y.y
Sx	X.x
---	---

state    val

- 栈由一对数组state和val实现
- state元素是指向LR(1)分析表中状态的指针（或索引）
- 如果state[i]保存对应符号A的状态，val[i]中就存放分析树中与结点A对应的属性值。

■ **假设**综合属性刚好在每次归约前计算

$A \rightarrow XYZ$  对应的语义规则是  $A.a = f(X.x, Y.y, Z.z)$

# 修改分析程序

- 对于终结符号
  - ◆ 其综合属性值由词法分析程序产生
  - ◆ 当分析程序执行移进操作时，其属性值随状态符号一起入栈。
- 为每个语义规则编写一段代码，以计算属性值
- 对每一个产生式 $A \rightarrow XYZ$ 
  - ◆ 把属性值的计算与归约动作联系起来
  - ◆ 归约前，执行与产生式相关的代码段
  - ◆ 归约：右部符号的相应状态及其属性出栈  
左部符号的相应状态及其属性入栈
- LR分析程序中应增加计算属性值的代码段

# 例：用LR分析程序实现表达式求值

产生式	语义规则
$L \rightarrow E$	$\text{print}(E.\text{val})$
$E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} = E.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit.lexval}$

代码段

$\text{print}(\text{val}[\text{top}])$

$\text{val}[\text{ntop}] = \text{val}[\text{top}-2] + \text{val}[\text{top}]$

$\text{val}[\text{ntop}] = \text{val}[\text{top}-2] * \text{val}[\text{top}]$

$\text{val}[\text{ntop}] = \text{val}[\text{top}-1]$

栈指针变量  $\text{top}$  和  $\text{ntop}$  的控制：

- ◆ 当用  $A \rightarrow \beta$  归约时，若  $|\beta|=r$ ，在执行相应的代码段之前， $\text{ntop} = \text{top} - r + 1$ 。
- ◆ 在每一个代码段被执行之后， $\text{top} = \text{ntop}$

# 对 $3*5+4$ 进行分析的动作序列 (分析表见表4-8)

步骤	输入	分析栈	分析动作
(1)	$3*5+4\$$	state: 0 val: -	移进 5
(2)	$*5+4\$$	state: 0 5 val: - 3	归约, 用 $F \rightarrow \text{digit}$ goto[0,F]=3
(3)	$*5+4\$$	state: 0 3 val: - 3	归约, 用 $T \rightarrow F$ goto[0,T]=2
(4)	$*5+4\$$	state: 0 2 val: - 3	移进 7
(5)	$5+4\$$	state: 0 2 7 val: - 3 -	移进 5
(6)	$+4\$$	state: 0 2 7 5 val: - 3 - 5	归约, 用 $F \rightarrow \text{digit}$ goto[7,F]=10
(7)	$+4\$$	state: 0 2 7 10 val: - 3 - 5	归约, 用 $T \rightarrow T * F$ goto[0,T]=2

步骤	输入	分析栈	分析动作
(8)	+4\$	state: 0 2 val: - 15	归约, 用 $E \rightarrow T$ goto[0,E]=1
(9)	+4\$	state: 0 1 val: - 15	移进 6
(10)	4\$	state: 0 1 6 val: - 15 -	移进 5
(11)	\$	state: 0 1 6 5 val: - 15 - 4	归约, 用 $F \rightarrow \text{digit}$ goto[6,F]=3
(12)	\$	state: 0 1 6 3 val: - 15 - 4	归约, 用 $T \rightarrow F$ goto[6,T]=9
(13)	\$	state: 0 1 6 9 val: - 15 - 4	归约, 用 $E \rightarrow E+T$ goto[0,E]=1
(14)	\$	state: 0 1 val: - 19	接受



## 5.3 L-属性定义是自顶向下翻译

- 在自顶向下的分析过程中实现L属性定义的翻译
- 预测分析方法对文法的要求
  - ◆ 不含左递归
  - ◆  $A \rightarrow \alpha \mid \beta$   
 $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \phi$

5.3.1 消除翻译方案中的左递归

5.3.2 预测翻译程序的设计

## 5.3.1 消除翻译方案中的左递归

### ■ 例：考虑对简单表达式求值的语法制导定义

产生式	语义规则
$L \rightarrow E$	$\text{print}(E.\text{val})$
$E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} = E.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.\text{val}$

翻译方案：

- (1)  $L \rightarrow E \{ \text{print}(E.\text{val}) \}$
- (2)  $E \rightarrow E_1 + T \{ E.\text{val} = E_1.\text{val} + T.\text{val} \}$
- (3)  $E \rightarrow T \{ E.\text{val} = T.\text{val} \}$
- (4)  $T \rightarrow T_1 * F \{ T.\text{val} = T_1.\text{val} * F.\text{val} \}$
- (5)  $T \rightarrow F \{ T.\text{val} = F.\text{val} \}$
- (6)  $F \rightarrow (E) \{ F.\text{val} = E.\text{val} \}$
- (7)  $F \rightarrow \text{digit} \{ F.\text{val} = \text{digit}.\text{val} \}$

消除左递归的方法：

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

替换为：  $A \rightarrow \beta R$

$R \rightarrow \alpha R \mid \varepsilon$

由(2)和(3)有：

(2')  $E \rightarrow T \{ E.\text{val} = T.\text{val} \} M$

(3')  $M \rightarrow +T \{ E.\text{val} = E_1.\text{val} + T.\text{val} \} M_1$

(3'')  $M \rightarrow \varepsilon$

继承属性M. i：表示在M之前已经推导出的子表达式的值  
综合属性M. s：表示在M完全展开之后得到的表达式的值

为(3'')设置把M.i传递给M.s的语义动作，得到：

(3'')  $M \rightarrow \varepsilon \{M.s = M.i\}$

对于(2')， $E \rightarrow T \{E.val = T.val\}$  M 通过M的属性 M.s 和 M.i 完成E和T的综合属性的传递 $E.val = T.val$ ，得到：

(2')  $E \rightarrow T \{M.i = T.val\}$   
 $M \{E.val = M.s\}$

对于(3')  $M \rightarrow +T \{E.val = E_1.val + T.val\}$   $M_1$

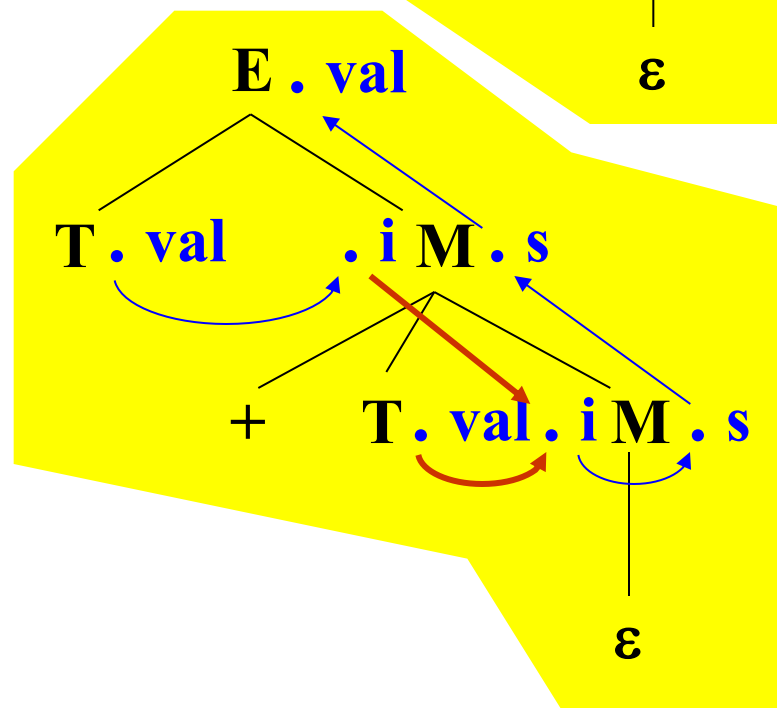
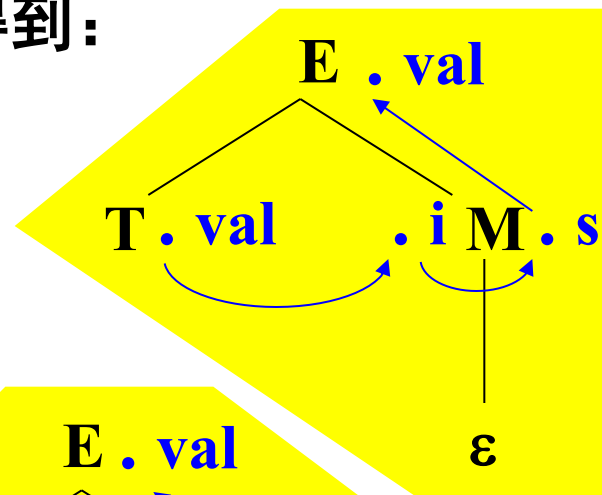
$M_1.i$ 的语义规则为： $M_1.i = M.i + T.val$

$M.s$ 的语义规则为： $M.s = M_1.s$

于是得到：

(3')  $M \rightarrow +T \{M_1.i = M.i + T.val\}$   
 $M_1 \{M.s = M_1.s\}$

同样，通过引入非终结符号N，  
可以得到(4)和(5)的变换结果



# 翻译方案

## 表达式3\*5+4的翻译过程

$L \rightarrow E \{ \text{print}(E.\text{val}) \}$

$E \rightarrow T \{ M.i = T.\text{val} \}$

$M \{ E.\text{val} = M.s \}$

$M \rightarrow +$

$T \{ M_1.i = M.i + T.\text{val} \}$

$M_1 \{ M.s = M_1.s \}$

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

$T \rightarrow F \{ N.i = F.\text{val} \}$

$N \{ T.\text{val} = N.s \}$

$N \rightarrow *$

$F \{ N_1.i = N.i * F.\text{val} \}$

$N_1 \{ N.s = N_1.s \}$

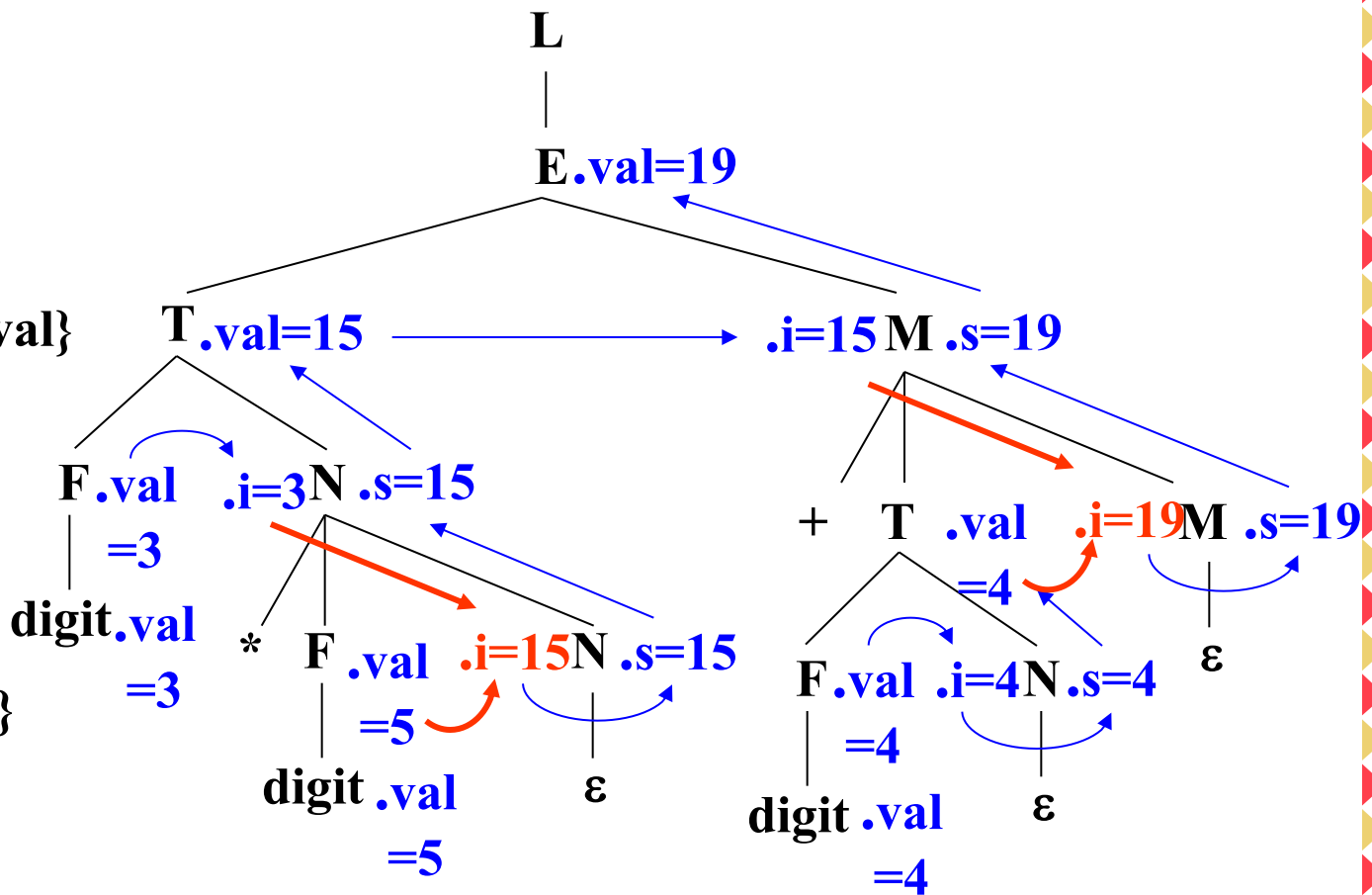
$N \rightarrow \epsilon \{ N.s = N.i \}$

$F \rightarrow ($

$E$

$) \{ F.\text{val} = E.\text{val} \}$

$F \rightarrow \text{digit} \{ F.\text{val} = \text{digit}.\text{lexval} \}$



# 消除翻译方案中左递归的一般方法

翻译方案:

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

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

消除基本文法中的左递归:

$$A \rightarrow XR$$

$$R \rightarrow YR \mid \varepsilon$$

为R设置继承属性R. i和综合属性R. s

R. i: 表示在R之前已经扫描过的符号串的属性值

R. s: 表示在R完全展开为终结符号之后得到的符号串的属性值。

翻译方案转换为:

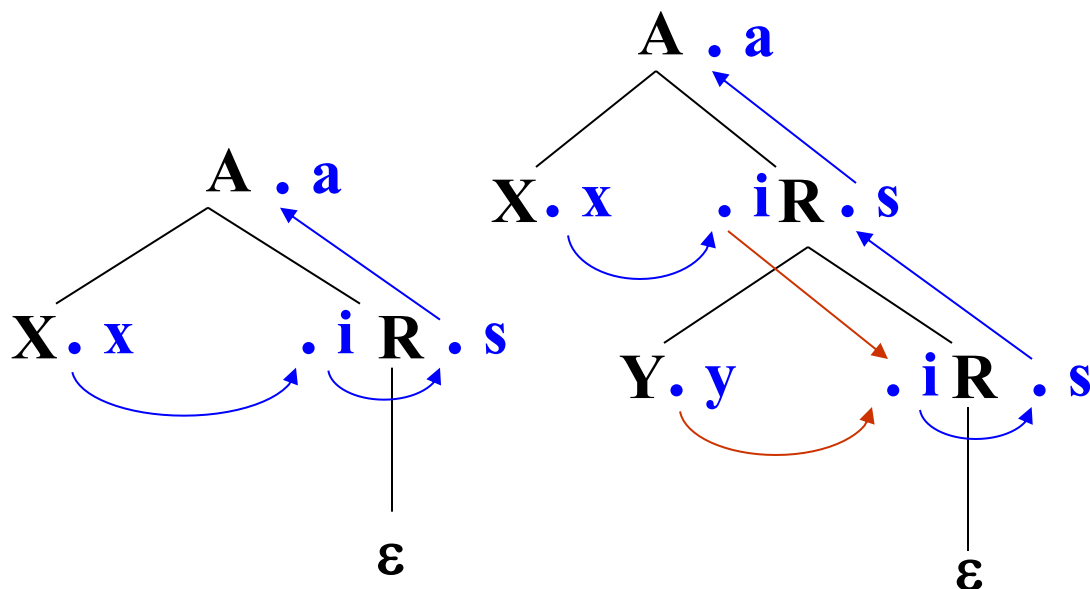
$$A \rightarrow X \quad \{R. i = f(X. x)\}$$

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

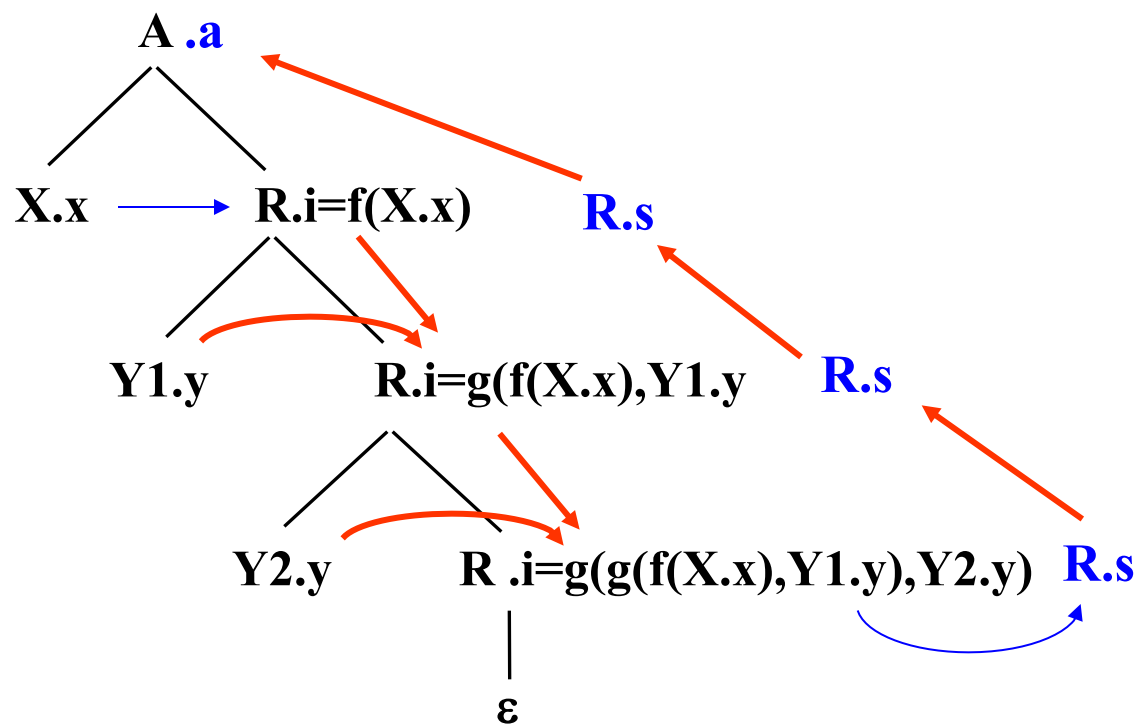
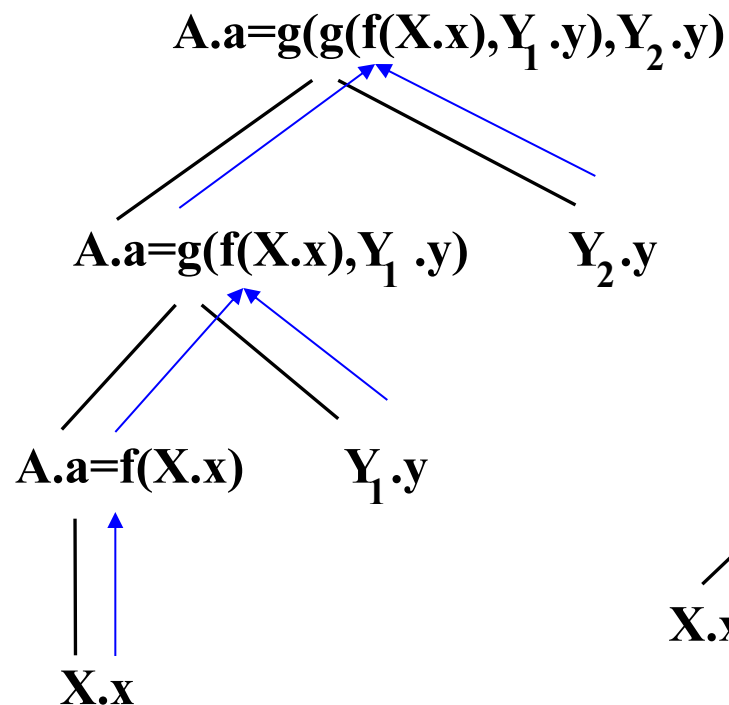
$$R \rightarrow Y \quad \{R_1. i = g(R. i, Y. y)\}$$

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

$$R \rightarrow \varepsilon \quad \{R. s = R. i\}$$



# 计算属性的两种方法



## 5.3.2 预测翻译程序的设计

### ■ 从翻译方案出发构造自顶向下的语法制导翻译程序

#### 算法5.2：构造语法制导的预测翻译程序

输入：基础文法适合于预测分析的语法制导翻译方案

输出：语法制导翻译程序

方法：（修改预测分析程序的构造技术）

(1) 为每个非终结符号A建立一个函数（可以是递归函数）

- ◆ A的每一个继承属性对应函数的一个形参
- ◆ A的综合属性作为函数的返回值
- ◆ A产生式中的每个文法符号的每个属性都对应一个局部变量

(2) A的函数的代码由多个分支组成

## 算法5.2：构造语法制导的预测翻译程序（续）

### (3) 与每个产生式相关的程序代码

- ◆ 按照从左到右的顺序考虑产生式右部的记号、非终结符号和语义动作
- ◆ 对带有综合属性 $x$ 的记号 $X$ 
  - 把属性 $x$ 的值保存于为 $X.x$ 声明的变量中
  - 产生一个匹配记号 $X$ 的调用
  - 推进扫描指针
- ◆ 对非终结符号 $B$ 
  - 产生一个函数调用语句 $c=B(b_1, b_2, \dots, b_k)$
  - $b_i (i=1, 2, \dots, k)$ 是对应于 $B$ 的继承属性的变量
  - $c$ 是对应于 $B$ 的综合属性的变量
- ◆ 对每一个语义动作
  - 把动作代码复制到分析程序中
  - 用代表属性的变量代替翻译方案中引用的属性



示例：

## 为简单表达式求值的翻译方案构造翻译程序

- 为每个非终结符号构造一个函数

`void fxL(void)`

`int fxE(void)`

`int fxM(int in)`

`int fxT(void)`

`int fxN(int in)`

`int fxF(void)`

## 与 $E \rightarrow TM$ 、 $M \rightarrow +TM | \varepsilon$ 相应的分析过程

**// $E \rightarrow TM$**

```
void proc_E(void) {  
    proc_T();  
    proc_M();  
};
```

**// $M \rightarrow +TM | \varepsilon$**

```
void proc_M(void) {  
    if (lookahead == '+') {  
        match( '+' );  
        proc_T();  
        proc_M();  
    }  
};
```

# 实现翻译方案的函数

$E \rightarrow T \{M.i = T.val\}$   
 $M \{E.val = M.s\}$

```
int fxE(void) {  
    int eval, tval, mi, ms;  
    tval=fxT();  
    mi=tval;  
    ms=fxM(mi);  
    eval=ms;  
    return eval;  
}
```

# 实现翻译方案的函数

```
int fxM(int in) {  
    int tval, i1, s1, s;  
    char addoplexeme;  
    if (lookahead == '+') { // 产生式  $M \rightarrow +TM$   
        addoplexeme = lexval;  
        match( '+' );  
        tval = fxT();  
        i1 = in + tval;  
        s1 = fxM(i1);  
        s = s1;  
    };  
    else s = in; // 产生式  $M \rightarrow \epsilon$   
    return s  
}
```

$M \rightarrow +$

$T \{ M_1.i = M.i + T.val \}$

$M_1 \{ M.s = M_1.s \}$

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

## 5.4 L属性定义的自底向上翻译

- 在自底向上的分析过程中实现L属性定义的翻译
- 可以实现任何基于LL(1)文法的L属性定义
- 可以实现许多（不是全部）基于LR(1)文法的L属性定义

**5.4.1 移走翻译方案中嵌入的语义规则**

**5.4.2 直接使用分析栈中的继承属性**

**5.4.3 变换继承属性的计算规则**

**5.4.4 改写语法制导定义为S属性定义**

## 5.4.1 移走翻译方案中嵌入的语义规则

- 自底向上地处理继承属性
- 等价变换：
  - 使所有嵌入的动作都出现在产生式的右端末尾
- 方法：
  - ◆ 在基础文法中引入新的产生式，形如： $M \rightarrow \epsilon$
  - ◆  $M$ ：标记非终结符号，用来代替嵌入在产生式中的动作
  - ◆ 把被 $M$ 替代的动作放在产生式 $M \rightarrow \epsilon$ 的末尾

**示例：** 去掉如下翻译方案中嵌入的动作：

$E \rightarrow TR$

$R \rightarrow +T \{\text{print}('+\')$

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

- 标记非终结符号M和N，及产生式  $M \rightarrow \varepsilon$  和  $N \rightarrow \varepsilon$
- 用M和N替换出现在R产生式中的动作

### ■ 新的翻译方案

$E \rightarrow TR$

$R \rightarrow +TMR \mid -TNR \mid \varepsilon$

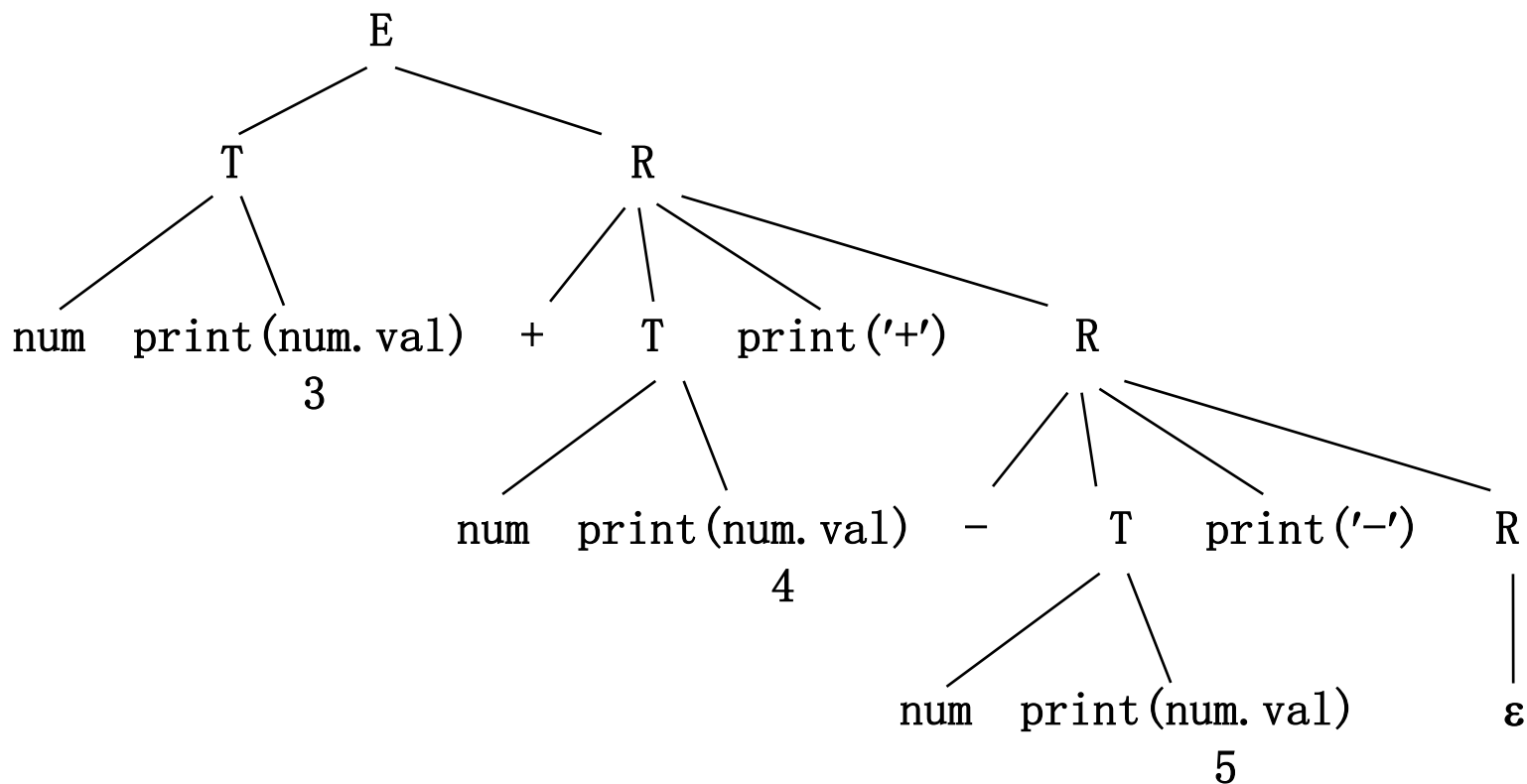
$T \rightarrow \text{num} \{\text{print}(\text{num.val})\}$

$M \rightarrow \varepsilon \{\text{print}('+\')$

$N \rightarrow \varepsilon \{\text{print}('-'\})\}$

# 变换前、后的翻译方案是等价的

变换前，表达式 $3+4-5$ 的分析树：

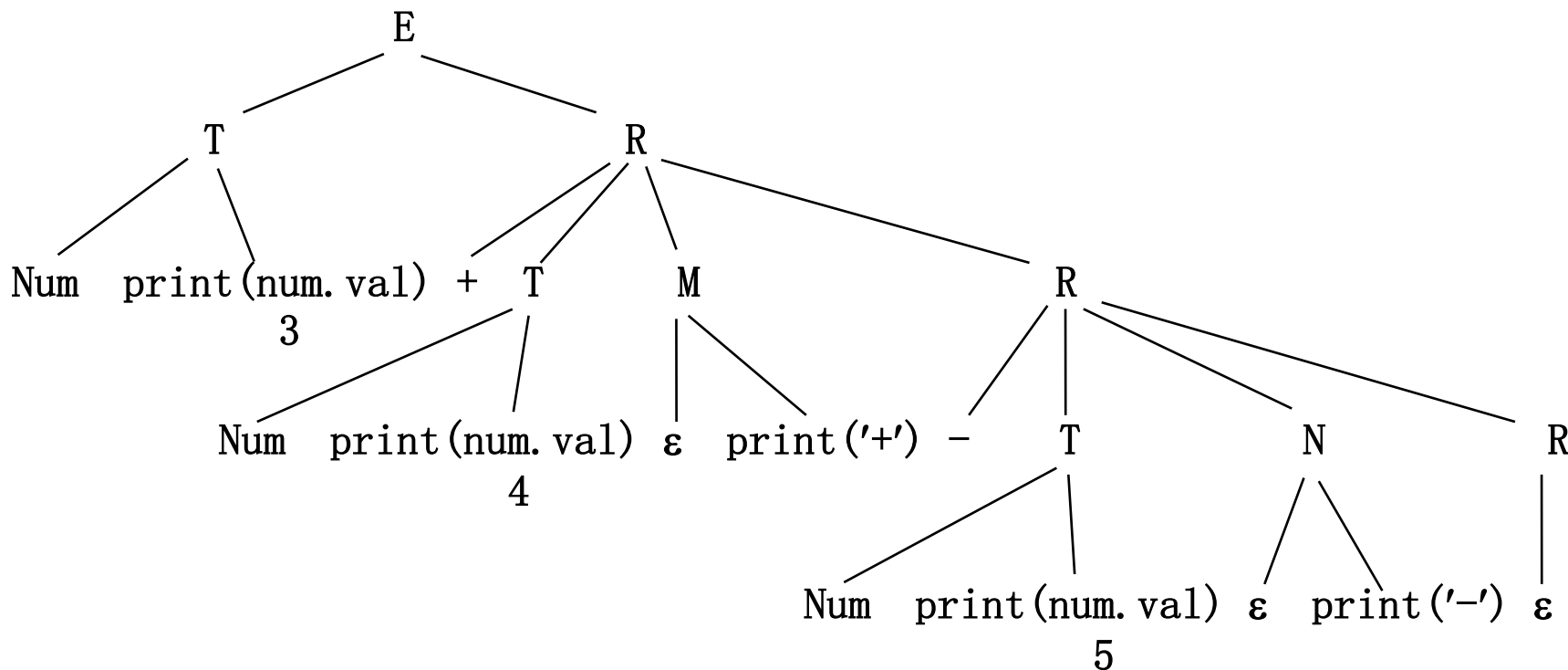


- 深度优先的顺序进行遍历
- `print(num1.val) print(num2.val) print('+') print(num3.val) print('-')`
- 动作执行的结果是： $34+5-$



# 变换前、后的翻译方案是等价的（续）

变换后，表达式 $3+4-5$ 的分析树：



- 深度优先的顺序进行遍历
- `print(num1.val) print(num2.val) print('+') print(num3.val) print('-')`
- 动作执行的结果是：34+5-

## 5.4.2 直接使用分析栈中的继承属性

- LR分析程序对产生式 $A \rightarrow XY$ 的归约
- 考虑分析过程中属性的计算

$Y_k$	$Y_k \cdot y$
$X_n$	$X_n \cdot x$
$Y_2$	$Y_2 \cdot y$
$X_2$	$X_2 \cdot x$
$X_1$	$X_1 \cdot s$
top → ...	
state	val

$$X \rightarrow X_1 X_2 \dots X_n$$

$$Y \rightarrow Y_1 Y_2 \dots Y_k$$

$$Y.i = X.s$$

# 复制规则的重要作用

输入符号串: real p, q, r

翻译方案:

$D \rightarrow T \{ \mathbf{L.in = T.type} \}$

$L$

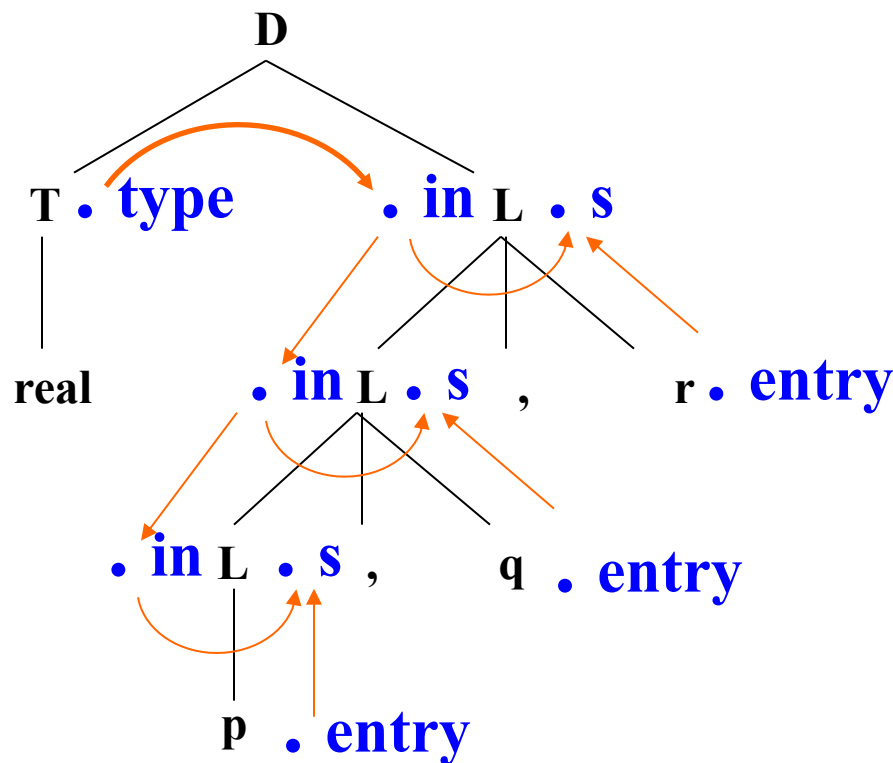
$T \rightarrow \text{int} \{ T.type = \text{integer} \}$

$T \rightarrow \text{real} \{ T.type = \text{real} \}$

$L \rightarrow \{ \mathbf{L_1.in = L.in} \}$

$L_1, \text{id} \{ \text{addtype}(\text{id.entry}, L.in) \}$

$L \rightarrow \text{id} \{ \text{addtype}(\text{id.entry}, L.in) \}$



# 例：应用继承属性，用复制规则传递标识符的类型

输入	栈	分析动作
real p,q,r\$	state: val:	移进
p,q,r\$	state: real val: real	归约，用T→real
p,q,r\$	state: T val: real	移进
,q,r\$	state: T val: real	归约，用L→id
,q,r\$	state: T val: real	移进
q,r\$	state: T val: real	移进
,r\$	state: T val: real	归约，用L→L,id
,r\$	state: T val: real	移进
r\$	state: T val: real	移进
\$	state: T val: real	归约，用L→L,id
\$	state: T val: real	归约，用D→TL
\$	state: D val: -	接受

# 计算属性值的代码段

产生式	代码段
$D \rightarrow TL$	
$T \rightarrow \text{int}$	<code>val[ntop]=integer</code>
$T \rightarrow \text{real}$	<code>val[ntop]=real</code>
$L \rightarrow L, \text{id}$	<code>addtype(val[top], val[top-3])</code>
$L \rightarrow \text{id}$	<code>addtype(val[top], val[top-1])</code>

- `top`和`ntop`分别是归约前和归约后的栈顶指针
- 当用产生式  $L \rightarrow \text{id}$  归约时, `L.in` 的位置?
- 当用产生式  $L \rightarrow L, \text{id}$  进行归约时, `L.in` 的位置?
- 和 `L.in` 有关的动作?

## 5.4.3 变换继承属性的计算规则

- 要想从栈中取得继承属性，**当且仅当**文法允许属性值在栈中存放的位置可以预测。

例：属性值在栈中的位置不可预测的语法制导定义

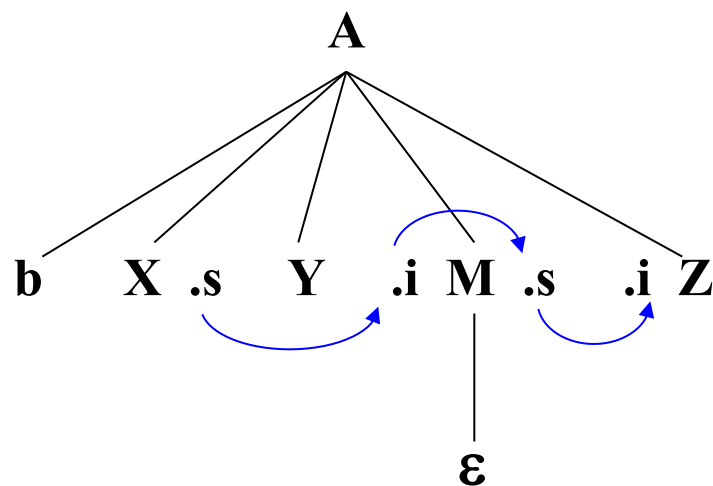
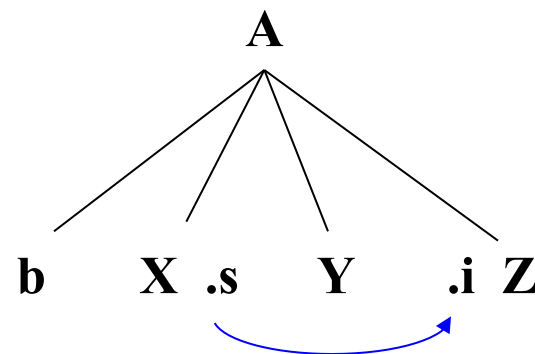
	产生式	语义规则
(1)	$A \rightarrow aXZ$	$Z.i = X.s$
(2)	$A \rightarrow bXYZ$	$Z.i = X.s$
(3)	$X \rightarrow x$	$X.s = 5$
(4)	$Y \rightarrow y$	$Y.s = 7$
(5)	$Z \rightarrow z$	$Z.s = g(Z.i)$

当用  $Z \rightarrow z$  进行归约时，  
 $Z.i$  可能在  $val[top-1]$  处  
也可能在  $val[top-2]$  处

# 模拟继承属性的计算

- 引入**标记非终结符号**，对原语法制导定义进行等价变换

	产生式	语义规则
(1)	$A \rightarrow aXZ$	$Z.i = X.s$
(2')	$A \rightarrow bXYMZ$	$M.i = X.s;$ $Z.i = M.s$
(3)	$X \rightarrow x$	$X.s = 5$
(4)	$Y \rightarrow y$	$Y.s = 7$
(5)	$Z \rightarrow z$	$Z.s = g(Z.i)$
(6)	$M \rightarrow \varepsilon$	$M.s = M.i$



# 用标记非终结符号模拟 非复制规则的语义规则

例：考虑如下的产生式及语义规则：

$A \rightarrow aXY$

$Y.i = f(X.s)$

$Y \rightarrow y$

$Y.s = g(Y.i)$

---	a	X	y
---		X.s	

↑  
top

## ■ 引入标记非终结符号N

$A \rightarrow aXNY$

$N.i = X.s; Y.i = N.s$

$N \rightarrow \epsilon$

$N.s = f(N.i)$

$Y \rightarrow y$

$Y.s = g(Y.i)$

---	a	X	N	y
---		X.s	N.s	

↑  
top

- 所有继承属性均由复制规则实现
- 继承属性在栈中的位置可以预测



## 算法5.3：L属性定义的自底向上分析和翻译

输入：基础文法是LL(1)文法的L属性定义

输出：在分析过程中计算所有属性值的分析程序

方法：

假设：

每个**非终结符号** $A$  都有一个继承属性  $A.i$

每一个**文法符号** $X$  都有一个综合属性  $X.s$

(1) 对每个产生式  $A \rightarrow X_1 X_2 \dots X_n$

引入 $n$ 个新的**标记非终结符号** $M_1, M_2, \dots, M_n$

用产生式  $A \rightarrow M_1 X_1 M_2 X_2 \dots M_n X_n$  代替原来的产生式

$X_j$  的继承属性与标记非终结符号  $M_j$  相联系

属性  $X_j.i$  (也就是  $M_j.s$ ) 总是在  $M_j$  处计算，且发生在开始做归约到  $X_j$  的动作之前。

## 算法5.3：L属性定义的自底向上分析和翻译（续）

(2) 在自底向上分析过程中，各个属性的值都可以被计算出来

**第一种情况：**用  $M_j \rightarrow \varepsilon$  进行归约

■ 已知：

- ◆ 每个标记非终结符号在文法中是唯一的
- ◆  $M_j$  属于哪个形式为  $A \rightarrow M_1 X_1 M_2 X_2 \cdots M_n X_n$  的产生式
- ◆ 计算属性  $X_j.i$  需要哪些属性、以及它们的位置

state	...	$M_A$	$M_1$	$X_1$	$M_2$	$X_2$	...	$M_{j-1}$	$X_{j-1}$	$M_j$
val	...	$A.i$	$X_1.i$	$X_1.s$	$X_2.i$	$X_2.s$	...	$X_{j-1}.i$	$X_{j-1}.s$	$M_j.s$

$\begin{array}{ccccccccccc} \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow \\ \text{top-2}(j-1) & & \text{top-2}(j-1)+2 & & \text{top-2}(j-2)+2 & & & & & & \text{top} \\ & \uparrow & & \uparrow & & & & & \uparrow & & \text{ntop} \\ & \text{top-2}(j-1)+1 & & \text{top-2}(j-2)+1 & & & & & \text{top-1} & & \end{array}$

# 算法5.3：L属性定义的自底向上分析和翻译（续）

第二种情况：用  $A \rightarrow M_1 X_1 M_2 X_2 \cdots M_n X_n$  进行归约

■ 已知：

- ◆  $A.i$  的值、及其位置
- ◆ 计算  $A.s$  所需要的属性值均已在栈中已知的位置  
即各有关  $X_j$  的位置上

state	...	$M_A$	$M_1$	$X_1$	$M_2$	$X_2$	...	$M_n$	$X_n$
val	...	$A.i$	$X_1.i$	$X_1.s$	$X_2.i$	$X_2.s$	...	$X_n.i$	$X_n.s$

$\uparrow$                        $\uparrow$                        $\uparrow$                        $\uparrow$

$top-2n$        $top-2n+2$        $top-2n+4$                        $top$

state	...	$M_A$	$A$
val	...	$A.i$	$A.s$

$\uparrow$                        $\uparrow$

$top-1$        $top$

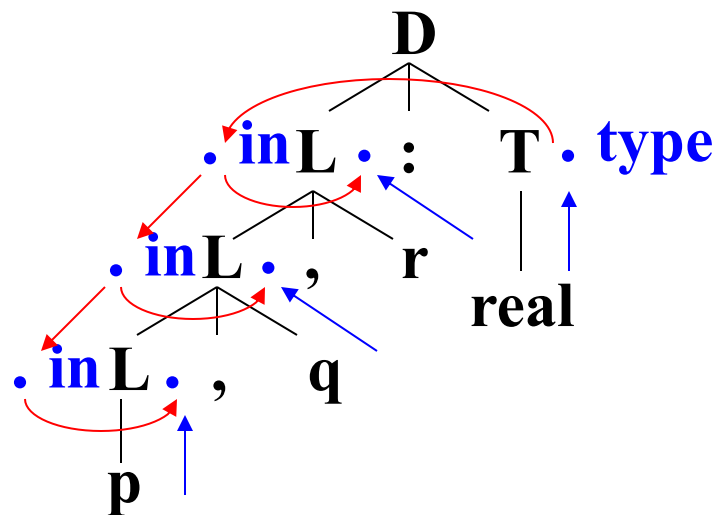
## 5.4.4 改写语法制导定义为S属性定义

例：PASCAL的变量声明语句可由如下文法产生：

$D \rightarrow L:T$

$T \rightarrow \text{integer} \mid \text{real}$

$L \rightarrow L, \text{id} \mid \text{id}$



### ■ 问题：

标识符由 $L$ 产生，而类型不在 $L$ 的子树中

归约从左向右进行，类型信息从右向左传递

只用综合属性不能使类型和标识符联系在一起

# 解决方法

- 改写文法，使类型作为标识符表的最后一个元素

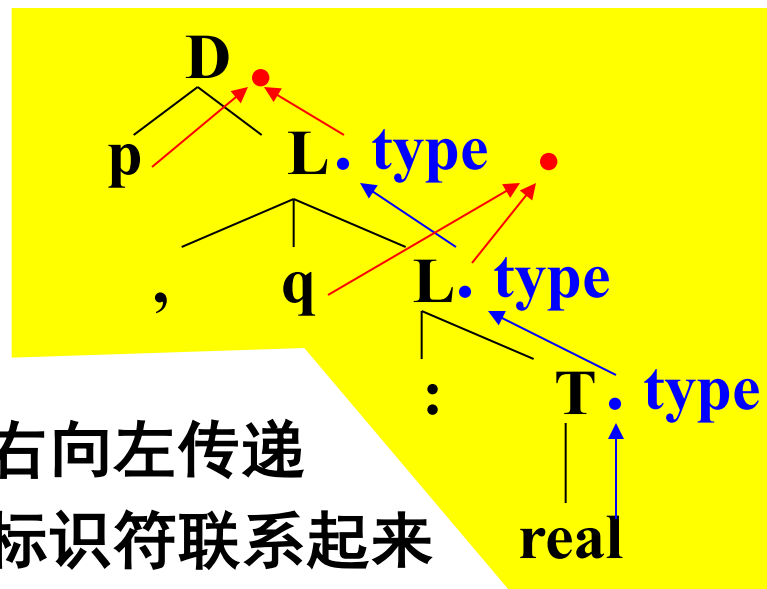
$D \rightarrow idL$

$L \rightarrow ,idL \mid :T$

$T \rightarrow integer \mid real$

- 改写后：

归约从右向左进行，类型信息从右向左传递  
仅用综合属性即可把类型信息和标识符联系起来



$D \rightarrow idL$        $addtype(id.entry, L.type)$

$L \rightarrow ,idL_1$        $L.type = L_1.type;$   
                          $addtype(id.entry, L_1.type)$

$L \rightarrow :T$        $L.type = T.type$

$T \rightarrow integer$        $T.type = integer$

$T \rightarrow real$        $T.type = real$

## 5.5 通用的语法制导翻译方法

- 按照一般的语法制导翻译步骤完成翻译。

输入符号串

——> 分析树

——> 依赖图

——> 语义规则的计算顺序

——> 计算结果

- 根据语法制导定义，为每个非终结符号构造一个翻译函数。
- 分析树的结点编号作为翻译函数的一个形参，函数根据在该结点处所用的产生式及其语义规则定义的属性之间的依赖关系以适当的顺序访问其诸子结点。
- 在遍历分析树的过程中，调用相应的函数来计算属性值，从而实现非L属性定义的翻译。

## 算法5.4 根据语法制导定义构造语法制导翻译程序

输入：语法制导定义

输出：语法制导翻译程序

方法：

为每一个非终结符号A建立一个函数，该函数可以是递归的。

(1) 设计函数头：

- ◆ 分析树结点作为函数的形参，A的每一个继承属性对应函数的一个形参，A的综合属性作为函数的返回值。
- ◆ 为A产生式中的每个文法符号的每一个属性都声明一个相应的局部变量。

(2) 函数体结构：

- ◆ 如果非终结符号A有多个候选式，则A的函数体首先要根据当前结点处使用的产生式来确定应执行的分支代码，即A的函数代码可由多个分支组成。

## 算法5.4 根据语法制导定义构造语法制导翻译程序(续)

### (3) 设计分支代码:

- ◆ 依据语法制导定义中与A的每个候选产生式相关的语义规则来设计相应的分支程序代码
- ◆ 根据属性之间的依赖关系确定访问子结点的顺序
- ◆ 子结点可以是内部结点、或者叶子结点。

① 若子结点是叶子结点，并且对应的记号X有综合属性x，则把它的值保存于为属性X.x声明的变量中。

② 若子结点是内部结点，且对应于非终结符号B

- ◆ 如果B有继承属性B.i，则先根据语义规则生成计算B.i值的代码，即将语义规则中出现的属性替换为相应的变量；
- ◆ 然后，产生一个函数调用语句  $c=B(n, b_1, b_2, \dots, b_k)$ ，其中n是B对应的分析树结点， $b_i(i=1, 2, \dots, k)$ 是对应于B的继承属性的变量，c是对应于B的综合属性的变量。



示例:

为表5-2中的语法制导定义构造语法制导翻译程序

产生式	语义规则
$A \rightarrow LM$	$L.i = 1(A.i)$ $M.i = m(L.s)$ $A.s = f(M.s)$
$A \rightarrow QR$	$R.i = r(A.i)$ $Q.i = q(R.s)$ $A.s = f(Q.s)$

■ 函数:

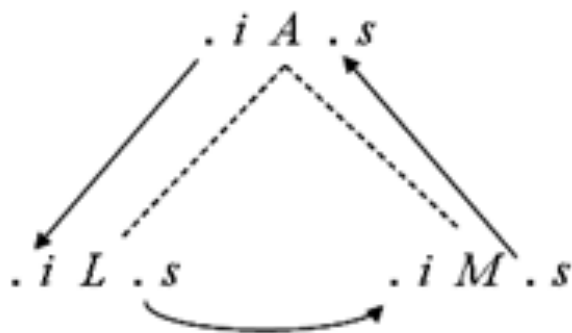
$fxA(n, ai)$

$fxL(n, li)$

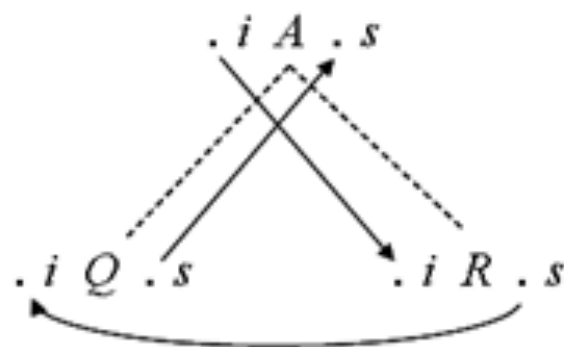
$fxM(n, mi)$

$fxQ(n, qi)$

$fxR(n, ri)$



(a) 产生式  $A \rightarrow LM$  的依赖图

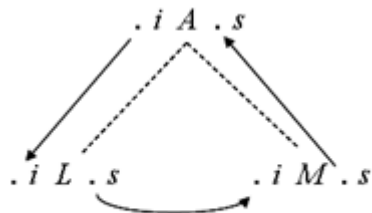


(b) 产生式  $A \rightarrow QR$  的依赖图

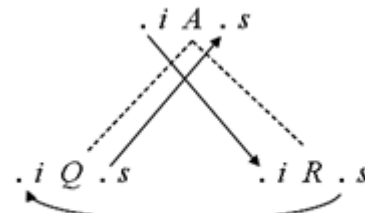
# 翻译函数 $fxA(n, ai)$ 示意

```
float fxA(n, ai) {
    float as, li, ls, mi, ms, qi, qs, ri, rs;
    switch ( 在结点n处使用的产生式 )
    {
    case 'A→LM':
        li=l(ai);
        ls=fxL(child(n, 1), li);
        mi=m(ls);
        ms=fxM(child(n, 2), mi);
        as=f(ms);
        return as;
    }
```

```
case 'A→QR':
    ri=r(ai);
    rs=fxR(child(n, 2), ri);
    qi=q(rs);
    qs=fxQ(child(n, 1), qi);
    as=f(qs);
    return as;
default:
    error();
}
}
```



(a) 产生式  $A \rightarrow LM$  的依赖图



(b) 产生式  $A \rightarrow QR$  的依赖图

# 小结

- 综合属性、继承属性
- 语法制导定义
  - ◆ S-属性定义
  - ◆ L-属性定义（继承属性应满足的**限制条件**）
- 翻译方案
  - ◆ 构造S-属性定义的翻译方案（语义动作放在产生式右尾）
  - ◆ 构造L-属性定义的翻译方案（语义动作插入**产生式之中**）
- S-属性定义的自底向上翻译（**改造LR分析程序**）
  - ◆ 分析栈的扩充
  - ◆ 分析控制程序的修改

# 小结 (续)

## ■ L-属性定义的预测翻译

- ◆ 为每个非终结符号构造递归函数
- ◆ 形参、返回值、局部变量
- ◆ 函数体、分支程序
- ◆ 分支程序段的设计

## ■ L-属性定义的自底向上翻译

- ◆ 通过复制规则引用继承属性
- ◆ 引入标记非终结符号及相应的产生式
- ◆ 修改语义动作、使继承属性由复制规则实现

## ■ 通用的语法制导翻译方法