第六章 属性文法和语法制导翻译

同济大学计算机系

程序语言语义的形式化描述
——形式语义学

- 1962年美国斯坦福大学麦克阿瑟（Mcarthur）教授在国际信息加工联合会年会上作了著名的报告——“**通往计算机的数学科学**”，系统地论述了程序设计语言语义形式化的重要性，以及和程序正确性、语言的正确实施等的关系，并提出在形式语言研究中使用抽象语法和状态向量等基本方法——形式语义学 。

语义分析方法

- 丹麦的科学家曾经运用指称语义学理论成功地实现了Ada语言的编译系统 。
- 形式语义学方法缺点：符号系统比较复杂，其描述文本不易读，不能借助这些形式系统自动完成语义处理任务。
- 目前实际应用中比较流行的语义描述和语义处理方法是**属性文法**和**语法制导翻译**的方法

内容线索

- **属性文法**
- **基于属性文法的处理方法**
- **语法制导翻译**

属性文法

- Knuth在1968年提出
- 在上下文无关文法的基础上，在描述语义动作时，为每个文法符号（终结符和非终结符）配备若干相关的“值”，如“类型”，“地址”等，称为**属性**。
- 对文法的每个产生式配备一组属性计算规则称为**语义规则**，它的描述形式为b:=f(c1,c2,…ck)，其中b,c1,c2…ck为文法符号的属性，f是一个函数。
- 每个文法符号联系于一组属性，且对每个产生式都给出其语义规则的文法称为**属性文法**。

属性和语义规则

- 属性代表与文法符号相关信息，如类型、值、代码序列、符号表内容等；
- 属性可以进行计算和传递；
- 在一个属性文法中，对应于每个产生式$A \rightarrow \alpha$都有一组与之相关联的语义规则，每条规则的形式为：

$$b:=f(c_1,c_2,\cdots,c_k)$$

这里，f是一个函数

（1）b是A的一个属性，并且$c_1,c_2,\cdots,c_k$是产生式右边文法符号的属性，则b是A的**综合属性**；

（2）b是产生式右边某个文法符号X的一个属性，并且$c_1,c_2,\cdots,c_k$ 是A或产生式右边任何文法符号的属性， b是X的**继承属性**；

- 属性b依赖于属性$c_1,c_2,\cdots,c_k$。

简单台式计算器的属性文法

产生式	语义规则
$L \rightarrow En$	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* F.val
$T \rightarrow F$	T.val :=F.val
$F \rightarrow (E)$	F.val :=E.val
$F \rightarrow digit$	F.val :=digit.lexval

记号表示

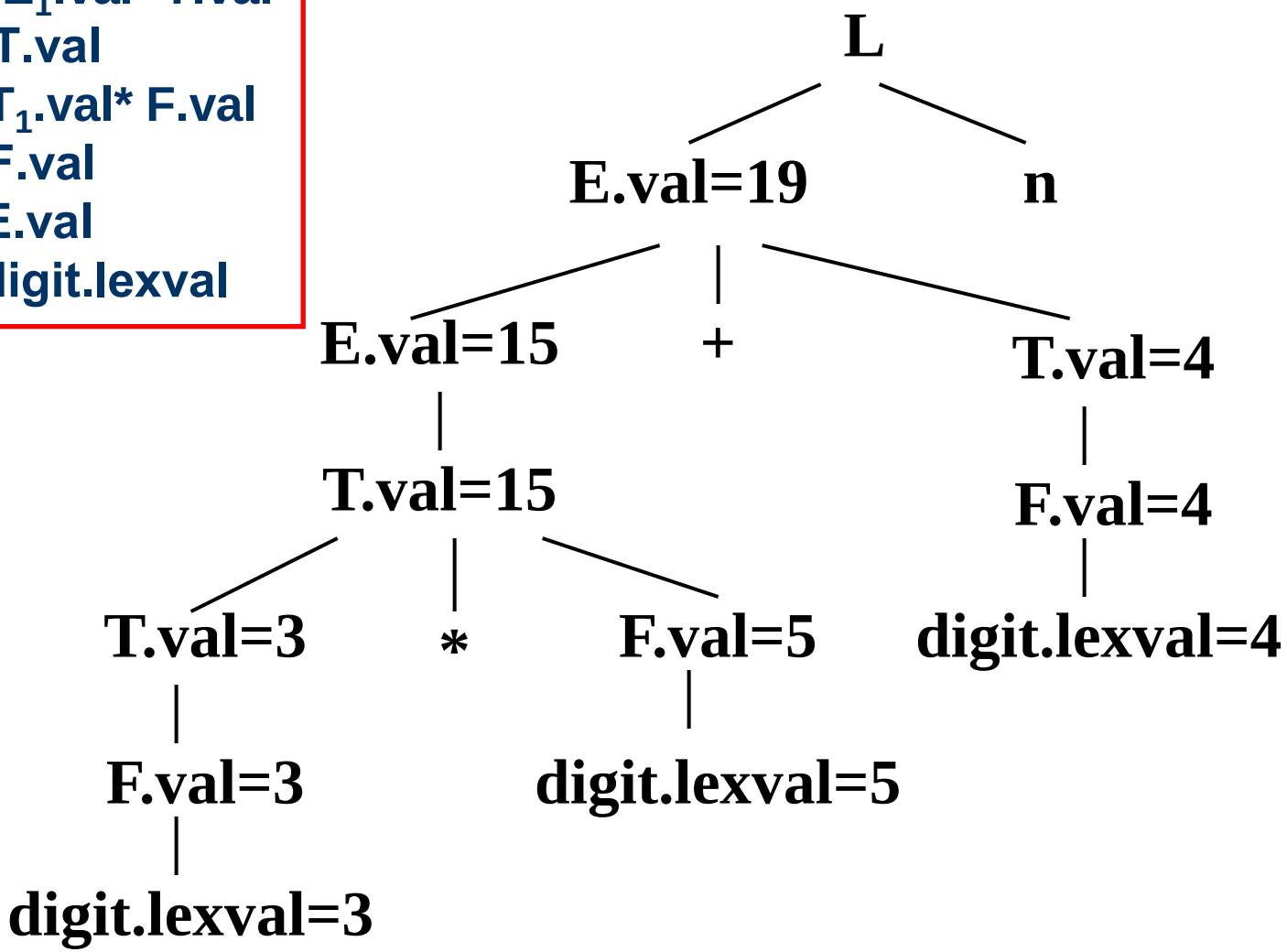
- 对于某个文法符号$X \in V_T \cup V_N$，用X . type（X的类型），X . cat（X的种别），X .val（X的值或地址）等表示它的**属性**。
- 用**下标**（上角标）区分同一产生式中**相同符号的多次出现**。

综合属性

- 在语法树中，一个结点的**综合属性**的值由其**子结点**的属性值确定。
- 使用自底向上的方法在每一个结点处使用语义规则计算综合属性的值
- 仅仅使用综合属性的属性文法称**S－属性文法**

产生式	语义规则
$L \rightarrow En$	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* F.val
$T \rightarrow F$	T.val :=F.val
$F \rightarrow (E)$	F.val :=E.val
$F \rightarrow digit$	F.val :=digit.lexval

3*5+4n的带注释的语法树



继承属性

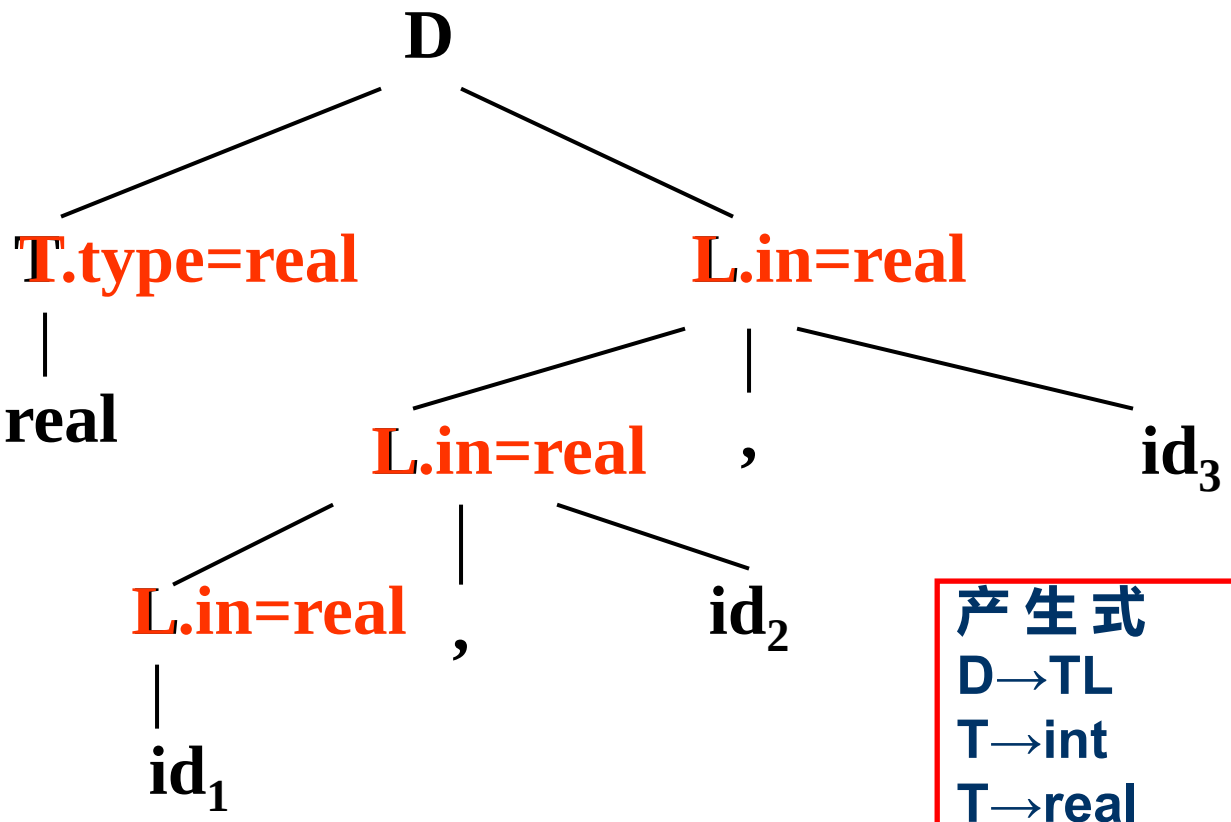
- 在语法树中，一个结点的**继承属性**由此结点的**父结点和/或兄弟结点**的某些属性确定
- 用继承属性来表示程序设计语言结构中的上下文依赖关系很方便

带继承属性L.in的属性文法

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

把标识符的类型填入符号表中，符号表入口由id.entry指明

句子real id_1，id_2，id_3的带注释的语法树



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

说明

- 终结符只有综合属性，由词法分析器提供
- 非终结符既可有综合属性也可有继承属性，文法开始符号的所有继承属性作为属性计算前的初始值
- 对出现在产生式右边的继承属性和出现在产生式左边的综合属性都必须提供一个计算规则。属性计算规则中只能使用相应产生式中的文法符号的属性
- 出现在产生式左边的继承属性和出现在产生式右边的综合属性不由所给的产生式的属性计算规则进行计算，它们由其它产生式的属性规则计算或者由属性计算器的参数提供
- 语义规则所描述的工作可以包括属性计算、静态语义检查、符号表操作、代码生成等等。

例. 考虑非终结符A，B和C，其中，

	综合属性	继承属性
A	b	a
B	c	
C		d

产生式A→BC可能有语义规则

C.d:=B.c+1

A.b:=A.a+B.c

而属性A.a和B.c在其它地方计算

某个文法符号的继承属性一般情况下依赖于它左边的兄弟节点的属性或产生式左边符号的属性。

内容线索

- ✓ 属性文法
- ■ **基于属性文法的处理方法**
- ■ 语法制导翻译

概述

- **基于属性文法的处理过程**

输入串 ⟶ 语法树 ⟶ 依赖图 ⟶ 语义规则计算次序

- **由源程序的语法结构所驱动的处理办法就是语法制导翻译法**
  - 依赖图
  - 树遍历

  } 多遍扫描

  - **一遍扫描：** 无需显式构造语法树或依赖图，一遍完成属性文法的语义规则计算

依赖图

- 在一棵语法树中的结点的**继承属性**和**综合属性**之间的相互依赖关系可以由称作**依赖图**的一个有向图来描述
- 为每一个包含过程调用的语义规则引入一个**虚综合属性b**，这样把每一个语义规则都写成

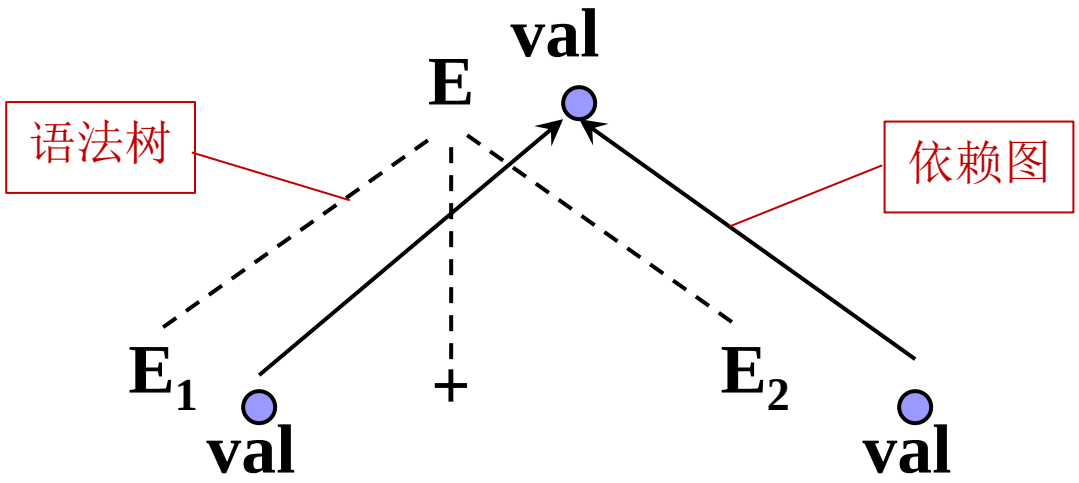
$$b:=f(c_1,c_2,\cdots,c_k)$$

  的形式
- 依赖图中**为每一个属性设置一个结点**，如果属性b依赖于属性c，则从属性c的结点有一条有向边连到属性b的结点。

依赖图构造算法

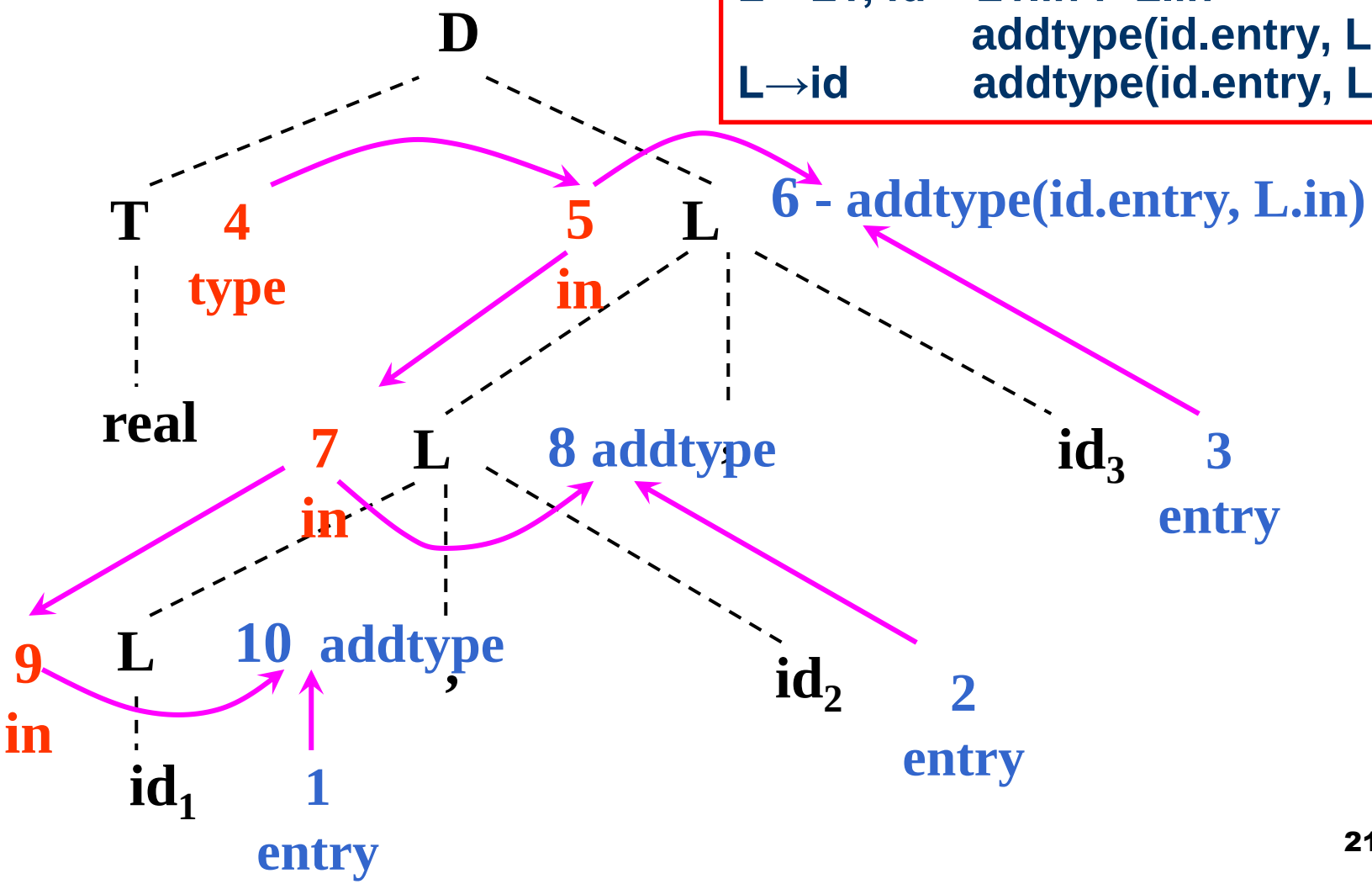
for 语法树中每一结点n do
  for 结点n的文法符号的每一个属性a do
    为a在依赖图中建立一个结点;
for 语法树中每一个结点n do
  for 结点n所用产生式对应的每一个语义规则
    $b:=f(c_1,c_2,\cdots,c_k)$ do
    for i:=1 to k do
      从c_i结点到b结点构造一条有向边;

$E \rightarrow E_1 + E_2$ $E.val := E_1.val + E_2.val$



句子real id_1, id_2, id_3的带注释的语法树的依赖图

产生式	语义规则
D→TL	L.in := T.type
T→int	T.type := integer
T→real	T.type := real
L→L1, id	L1.in :=L.in addtype(id.entry, L.in)
L→id	addtype(id.entry, L.in)



说明

- 一条求值规则只有在其各变元值均已求得的情况下才可以使用；
- 如果一属性文法不存在属性之间的循环依赖关系，那么称该文法为良定义的，我们**只处理良定义的属性文法**。

例如：p=f(c1), c1=g(c2), c2=h(p)

存在循环依赖

说明

- 从依赖图的拓扑排序中，可以得到计算语义规则的顺序，按这个顺序来计算就得到了输入符号串的翻译。
- 属性文法说明的翻译是很精确的。

树遍历的属性计算方法

- 通过树遍历的方法计算属性的值
  - 假设语法树已建立，且树中已带有开始符号的继承属性和终结符的综合属性
  - 以某种次序遍历语法树，直至计算出所有属性
    - 深度优先，从左到右的遍历
    - 可能会遍历多次

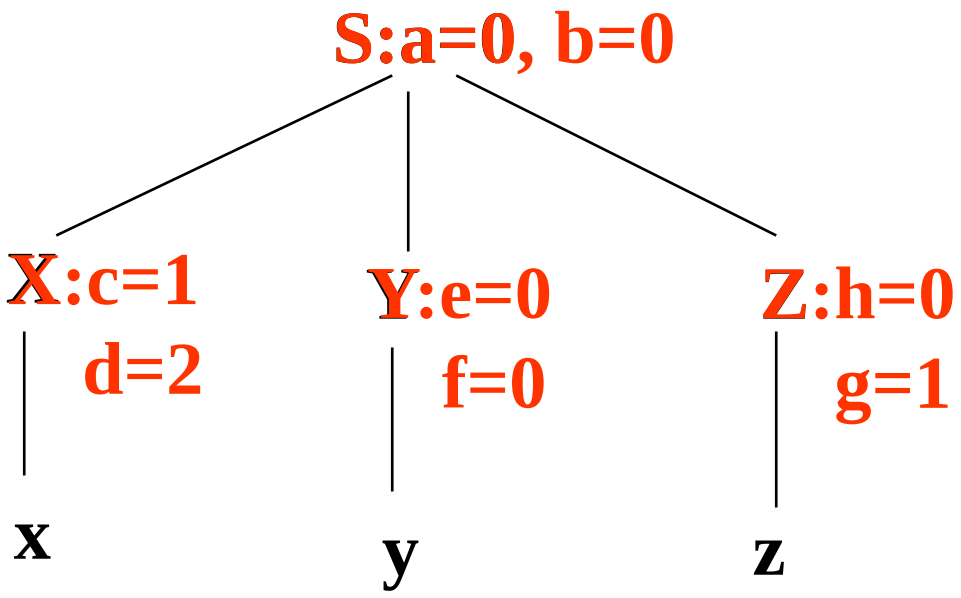
```
While 还有未被计算的属性 do
   VisitNode(S)      /*S是开始符号*/

procedure VisitNode (N:Node) ;
begin
  if N是一个非终结符 then
     /*假设它的产生式为N→X1…Xm*/
     for i :=1 to  m  do
     if Xi∈VN then /*即Xi是非终结符*/
          begin
            计算Xi的所有能够计算的继承属性;
            VisitNode (Xi)
     end;
     计算N的所有能够计算的综合属性
end
```

例 考虑属性文法G。其中，S有继承属性a，综合属性b；X有继承属性c、综合属性d；Y有继承属性e、综合属性f；Z有继承属性h、综合属性g

产 生 式	语 义 规 则
S→XYZ	Z.h := S.a
	X.c := Z.g
	S.b := X.d -2
	Y.e := S.b
X→x	X.d :=2*X.c
Y→y	Y.f := Y.e*3
Z→z	Z.g :=Z.h+1

假设S.a的初始值为0，输入串为xyz

产 生 式	语义规则
S→XYZ	Z.h := S.a
	X.c := Z.g
	S.b := X.d -2
	Y.e := S.b
X→x	X.d :=2*X.c
Y→y	Y.f := Y.e*3
Z→z	Z.g :=Z.h+1



第一次遍历求得：Z.h和Z.g
第二次遍历求得：X.c, X.d和S.b
第三次遍历求得：Y.e和Y.f

一遍扫描的处理方法

- 一遍扫描的处理方法是在语法分析的同时计算属性值
  - 所采用的语法分析方法
  - 属性的计算次序
- L - 属性文法适合于一遍扫描的自上而下分析
- S - 属性文法适合于一遍扫描的自下而上分析

内容线索

- ✓ 属性文法
- ✓ 基于属性文法的处理方法
- ■ 语法制导翻译

语法制导翻译

- 直观上说就是为每个产生式配上一个翻译子程序（称语义动作或语义子程序），并且在**语法分析**的同时执行它。
- 语义动作一方面规定了产生式**产生的符号串**的意义，另一方面又按照这种意义规定了生成中间代码应做的**基本动作**。
- 定义：在语法分析过程中，当一个产生式**获得匹配**（自上而下分析）和**用于归约**（自下而上分析）时，此产生式对应的语义子程序进入工作，完成既定翻译任务，产生中间代码。

例. 定义算术表达式E的“值”的语义动作:

1. $E \rightarrow E^{(1)}+E^{(2)}$

   {E.VAL := $E^{(1)}$.VAL+$E^{(2)}$.VAL}

2. $E \rightarrow 0$ {E.VAL := 0}

3. $E \rightarrow 1$ {E.VAL := 1}

上述语义动作虽然不产生中间代码，但产生式1、2、3所产生的句子有了具体意义，而且，能按照其语义动作，在分析句子的同时一步一步地算出每个句子的“值”。

语法制导翻译的作用

- 如果语义动作不是简单的计值程序，而是某种中间代码的产生程序，那么随着语法分析的进展，这种代码也逐步生成。
- 语法制导翻译的作用
  - 产生中间代码
  - 产生目标指令
  - 对输入串进行解释执行

非语法制导翻译方法

- 与语法制导翻译方法相对的是非语法制导翻译，即翻译程序将不受输入语言的文法的控制，如形式语义学的翻译方法。

语法制导翻译的例子

- 一个语法制导翻译的基础是一个文法，其中翻译成分依附在每一产生式上。

例.下列翻译模式，它定义翻译，即对每个输入x，其输出是x的逆转。定义此翻译的规则是

产生式	翻译式
(1)$S \rightarrow 0S$	(1)$s=s0$
(2)$S \rightarrow 1S$	(2)$s=s1$
(3)$S \rightarrow \varepsilon$	(3)$s=\varepsilon$

输入输出对可由(α,β)表示，其中α是输入句子形式，而β是输出句子形式。

(S,S)开始用产生式s→0s来扩展得到(0S,S0).

再用一次规则(1)，得到(00S,S00)。

再用规则(2)，就得到(001S,S100)。

然后应用规则(3)并得到(001,100)。

产生式	翻译式
(1)s→0s (2)s→1s (3)s→ε	(1)s=s0 (2)s=s1 (3)s=ε

例.语法制导的一个具体实现

$E \rightarrow T^1 + T^2$

{ if T^1.type = int and T^2.type= int

then E.type :=int

else error}

$E \rightarrow T^1$ or T^2

{ if T^1.type = bool and T^2.type=bool

then E.type :=bool

else error}

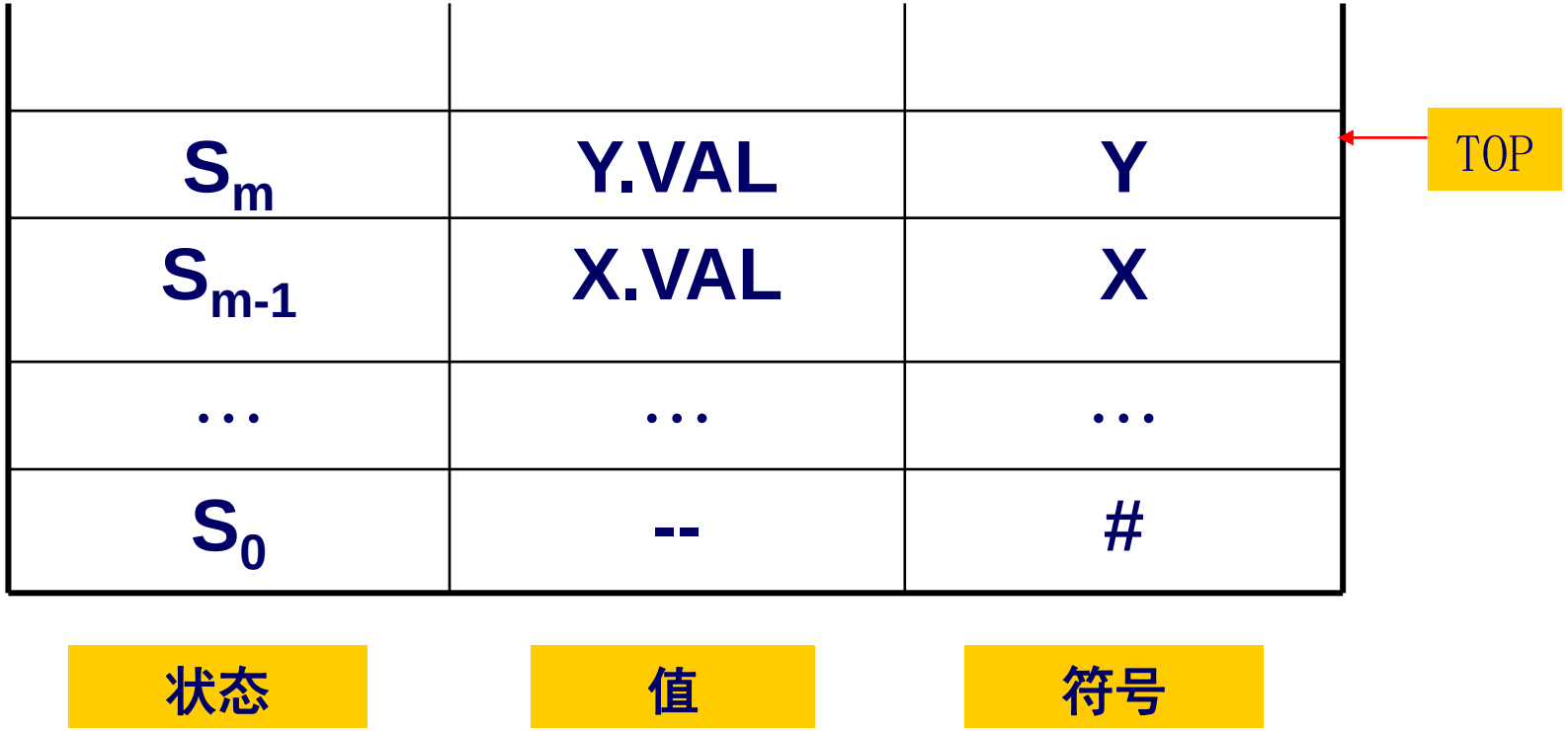
$T \rightarrow n$ { T.type := int}

$T \rightarrow b$ { T.type := bool}

LR(0)分析表

状态	action					GOTO	
	+	or	n	b	#	E	T
0			s4	s3		1	2
1					acc		
2	s5	s7					
3	r4	r4	r4	r4	r4		
4	r3	r3	r3	r3	r3		
5			s4	s3			6
6	r1	r1	r1	r1	r1		
7			s4	s3			8
8	r2	r2	r2	r2	r2		

LR分析器的栈加入语义值。



E→T1 + T2 {
if T1.type = int and T2.type= int
E.type :=int
else error}
E →T1 or T2 {
if T1.type = bool and T2.type=bool
E.type :=bool
else error }
T → n { T.type := int}
T → b { T.type := bool}

LR(0)分析表

状态	action					GOTO	
	+	or	n	b	#	E	T
0			s4	s3		1	2
1					acc		
2	s5	s7					
3	r4	r4	r4	r4	r4		
4	r3	r3	r3	r3	r3		
5			s4	s3			6
6	r1	r1	r1	r1	r1		
7			s4	s3			8
8	r2	r2	r2	r2	r2		

输入串： n + n

4	T	int
5	+	---
1	T	int
0	#	--

输入串
n + b

3	T	bool
5	+	---
4	T	int
0	#	--

E→T1 + T2 {
if T1.type = int and T2.type= int
E.type :=int
else error}
E →T1 or T2 {
if T1.type = bool and T2.type=bool
E.type :=bool
else error }
T → n { T.type := int}
T → b { T.type := bool}

LR(0)分析表

	action					GOTO	
状态	+	or	n	b	#	E	T
0			s4	s3		1	2
1					acc		
2	s5	s7					
3	r4	r4	r4	r4	r4		
4	r3	r3	r3	r3	r3		
5			s4	s3			6
6	r1	r1	r1	r1	r1		
7			s4	s3			8
8	r2	r2	r2	r2	r2		