

第4讲 语义分析

学习的主要内容和目标

➤ 学习的主要内容

- ◇ 语义分析的实现技术
- ◇ 符号表
- ◇ 类型检查

➤ 学习的目标

- ◇ 掌握S-属性文法和L-属性文法；
- ◇ 掌握基于语法分析的语义计算技术；
- ◇ 掌握符号表的一般组织方法；
- ◇ 理解类型检查的方法


语义分析概述

- 上下文无关文法是否描述了程序语言中所有规则？
- 有没有什么你熟悉的规则是超出了上下文无关文法的范畴？

语义分析概述

➤ 引例

```
int b=3;  
int main()  
{  
    int sum, a=2;  
    sum=a+b;  
    return 0;  
}
```



大家熟悉的
C语言

语义分析概述

➤ 引例

```
int b=3;  
int main()  
{  
    int sum, a=2;  
    sum=a+b;  
    return 0;  
}
```

先声明再使用
sum,a,b在语法分析的角度都是ID

语法分析只是检查这个位置上应该是一个标识符，但并不区分是哪一个标识符

sum=a
与
sum=b
执行的代码不相同

语义分析概述

➤ 引入词法分析

怎么知道前面定义的id就是后面使用的id

$block \rightarrow \{ decls\ stmts \}$

$decls \rightarrow decl\ decls \mid \varepsilon$

$decl \rightarrow type\ lid\ ;$

$type \rightarrow int \mid char \mid bool$

$lid \rightarrow id \mid \textcolor{red}{id}, lid$

$stmts \rightarrow stmt\ stmts \mid \varepsilon$

$stmt \rightarrow id = bexpr\ ;$

$bexpr \rightarrow expr \mid expr < expr \mid expr > expr$

$expr \rightarrow expr + expr \mid expr * expr \mid (expr) \mid \textcolor{red}{id}$

语义分析概述

➤ 常见的语义约束

- ◇ 名字的作用域分析：建立名字的定义和使用之间联系
- ◇ 类型检查：
- ◇ 唯一性检查：有的对象只能被定义一次（如枚举类型的元素不能重复出现）

语义分析概述

➤ 语义分析的主要任务

◇ 根据语言的语义约束检查程序的一致性；

✓ 语义约束怎么表示？

✓ 语义检查的实现？

语法制导定义

➤ 语义规则

$$E \rightarrow E_1 \text{ op } E_2 \quad \left\{ \begin{array}{l} \text{if } E_1.type == \text{real and } E_2.type == \text{real then real} \\ \text{else if } E_1.type == \text{int and } E_2.type == \text{int} \\ \text{then } E.type = \text{int else type_error} \end{array} \right.$$

语法制导定义

产生式

$$S \rightarrow E$$

$$E \rightarrow E_1 + T$$

$$E \rightarrow T$$

$$T \rightarrow T_1 * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow d$$

语义规则

$$\text{PRINT}(E.val)$$

$$E.val = E_1.val + T.val$$

$$E.val = T.val$$

$$T.val = T_1.val \times F.val$$

$$T.val = F.val$$

$$F.val = E.val$$

$$F.val = d.lexval$$

语法制导定义

➤ 定义

✧ 在上下文无关文法的基础上进行如下扩展：

- ✓ 为每个文法符号关联多个属性
- ✓ 为文法的每个产生式关联一个规则集合

语法制导定义

➤ 属性

◇ 刻画一个文法符号的特性

- ✓ 如符号的值
- ✓ 符号的名字串
- ✓ 符号的类型
- ✓ 符号的偏移地址
- ✓ 符号被赋予的寄存器
- ✓ 代码片断等

◇ 文法符号 X 关联属性 a 的属性值可通过 $X.a$ 访问

语法制导定义

► 语义规则

◇ 每个产生式 $A \rightarrow \alpha$ 都关联一个语义规则的集合，用于描述当前产生式上的静态约束。

✓ $b := f(c_1, c_2, \dots, c_k)$

✓ $f(c_1, c_2, \dots, c_k)$

✓ 其中， b, c_1, c_2, \dots, c_k 是该产生式中文法符号的属性

语法制导定义

➤ 属性文法

语义检查的实现?

产生式

$S \rightarrow E$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow d$

语义规则

$S.val = E.val$

$E.val = E_1.val + T.val$

$E.val = T.val$

$T.val = T_1.val \times F.val$

$T.val = F.val$

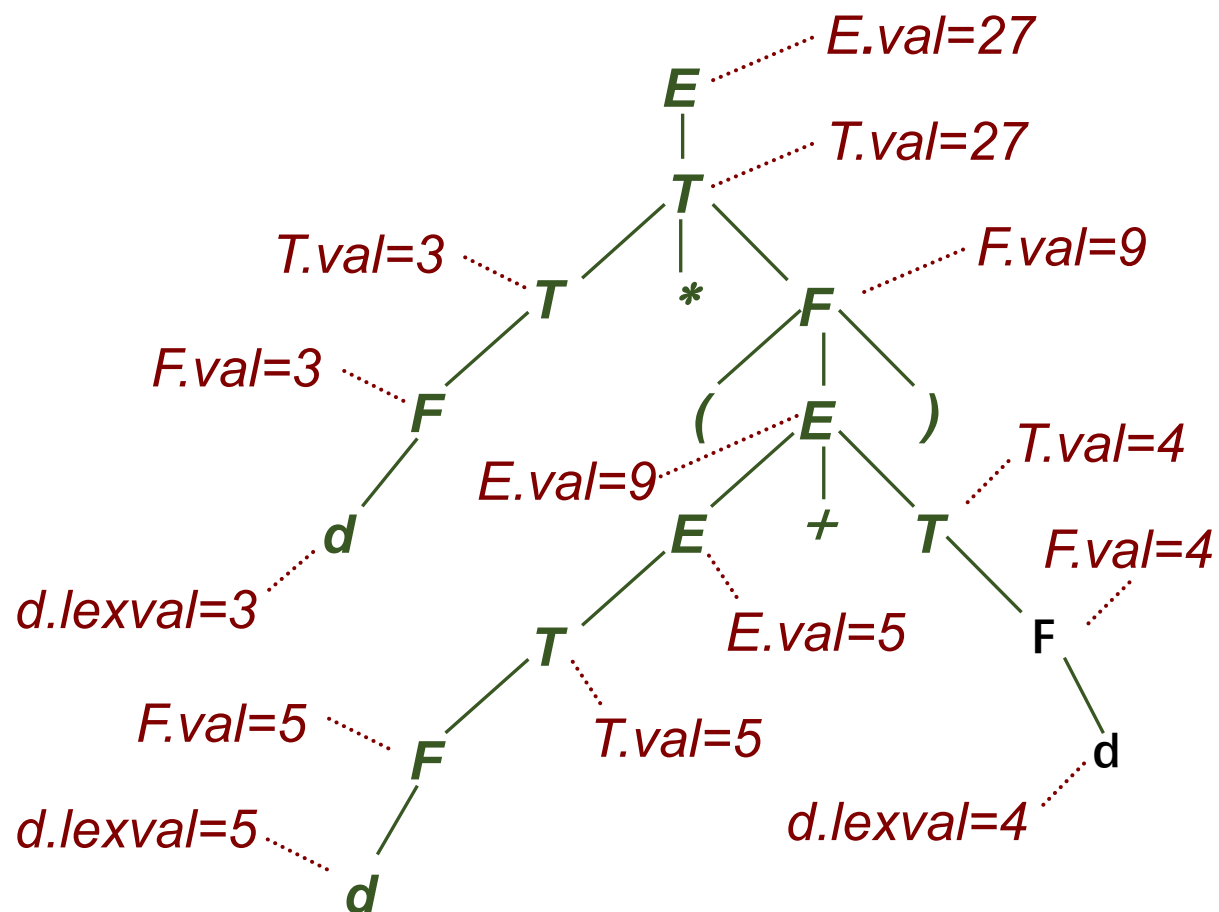
$F.val = E.val$

$F.val = d.lexval$

语法制导定义

► 综合属性

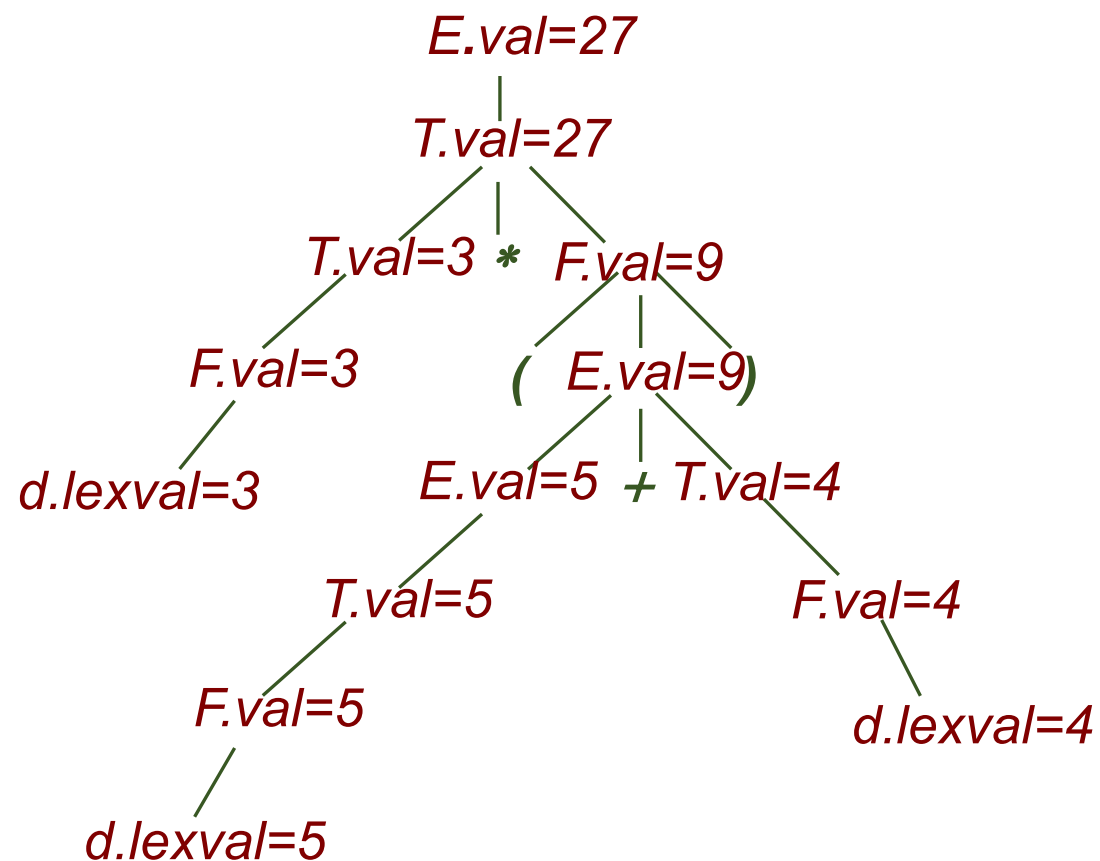
$3 * (5 + 4)$



语法制导定义

► 综合属性

表示出每个符号
属性值的语法树
称为注释语法树



语法制导定义

➤ 继承属性

产生式

$D \rightarrow T L$

$T \rightarrow \text{int}$

$T \rightarrow \text{real}$

$L \rightarrow L_1, v$

$L \rightarrow v$

语义规则

$L.in = T.type$

$T.type = \text{integer}$

$T.type = \text{real}$

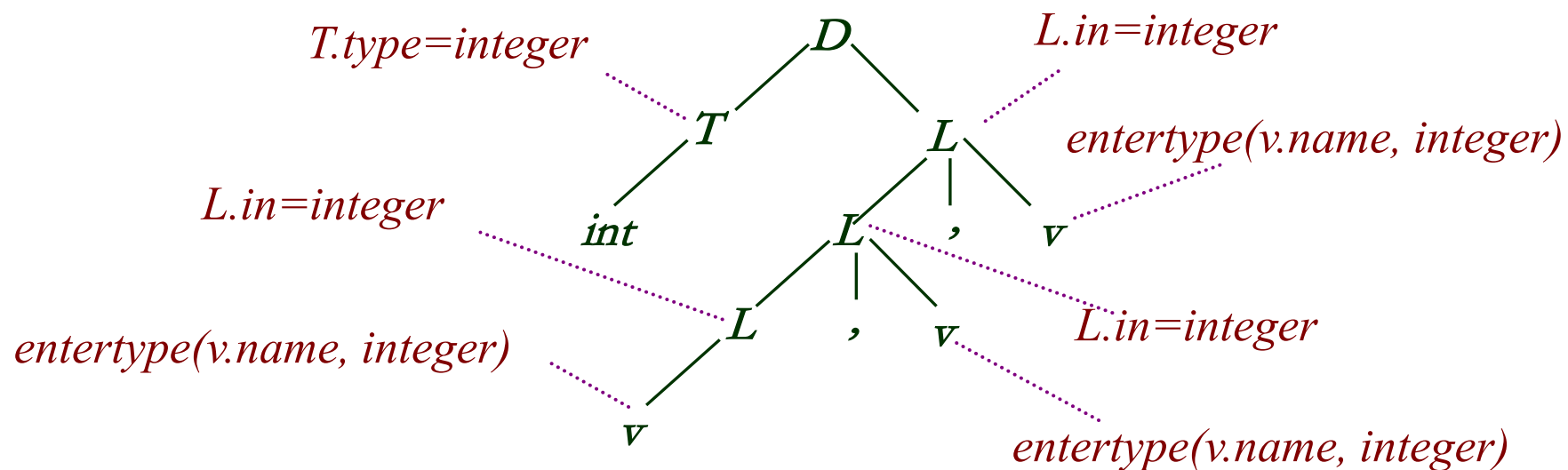
$L_1.in = L.in$

$\text{entertype}(v.name, L.in)$

$\text{entertype}(v.name, L.in)$

语法制导定义

➤ 声明语句 `int x,y,z`



语法制导定义

➤ 综合属性

- ◇ 用于“自下而上”传递信息
- ◇ 产生式 $A \rightarrow \alpha$ 的规则 $b := f(c_1, c_2, \dots, c_k)$ ，如果 b 是 A 的某个属性，则称 b 是 A 的一个综合属性

➤ 继承属性

- ◇ 用于“自上而下”传递信息
- ◇ 产生式 $A \rightarrow \alpha$ 的语义规则 $b := f(c_1, c_2, \dots, c_k)$ ，如果 b 是产生式右部某个文法符号 X 的某个属性，则称 b 是文法符号 X 的一个继承属性

依赖图

➤ 任意的语法制导定义

◇ 任意的句型

怎么知道哪个属性先算，哪个属性后算



依赖图

➤ 用有向图来表示属性之间的依赖关系

for 分析树中每一个结点n do

for 结点n所用产生式的语义规则中涉及的每一个属性a do
为a在依赖图中建立一个结点;

for 结点n所用产生式中每个形如 $f(c_1, c_2, \dots, c_k)$ 的语义规则 do
为该规则在依赖图中也建立一个结点（称为虚结点）;

for 分析树中每一个结点n do

for 结点n所用产生式对应的每个语义规则 $b := f(c_1, c_2, \dots, c_k)$ do
(可以只是 $f(c_1, c_2, \dots, c_k)$ ，此时b结点为一个虚结点)

for $i := 1$ to k do

从 c_i 结点到b结点构造一条有向边

依赖图

➤ Pascal风格声明语句的语法制导定义

产生式

$D \rightarrow L:T$

$T \rightarrow \text{integer}$

$T \rightarrow \text{real}$

$L \rightarrow id, L_1$

$L \rightarrow id$

语义规则

$L.in := T.type$

$T.type := \text{integer}$

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

$\text{entertype}(id.name, L.in);$

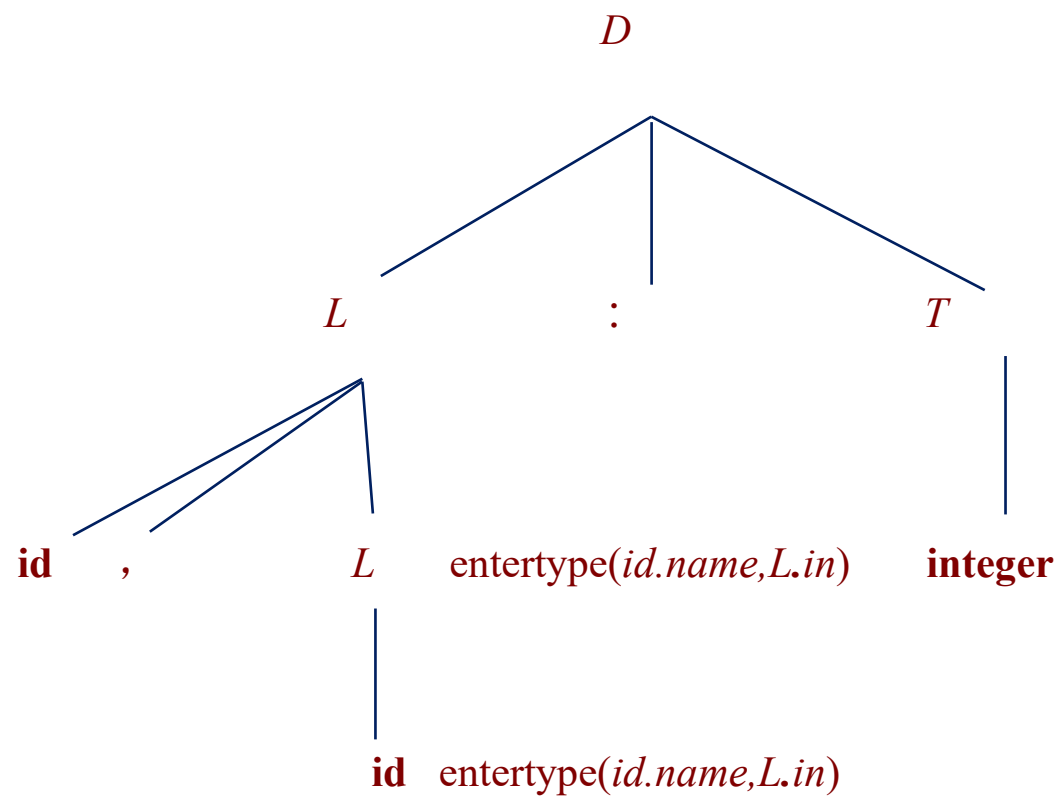
$L_1.in := L.in$

$\text{entertype}(id.name, L.in)$

依赖图

➤ 构造语法树

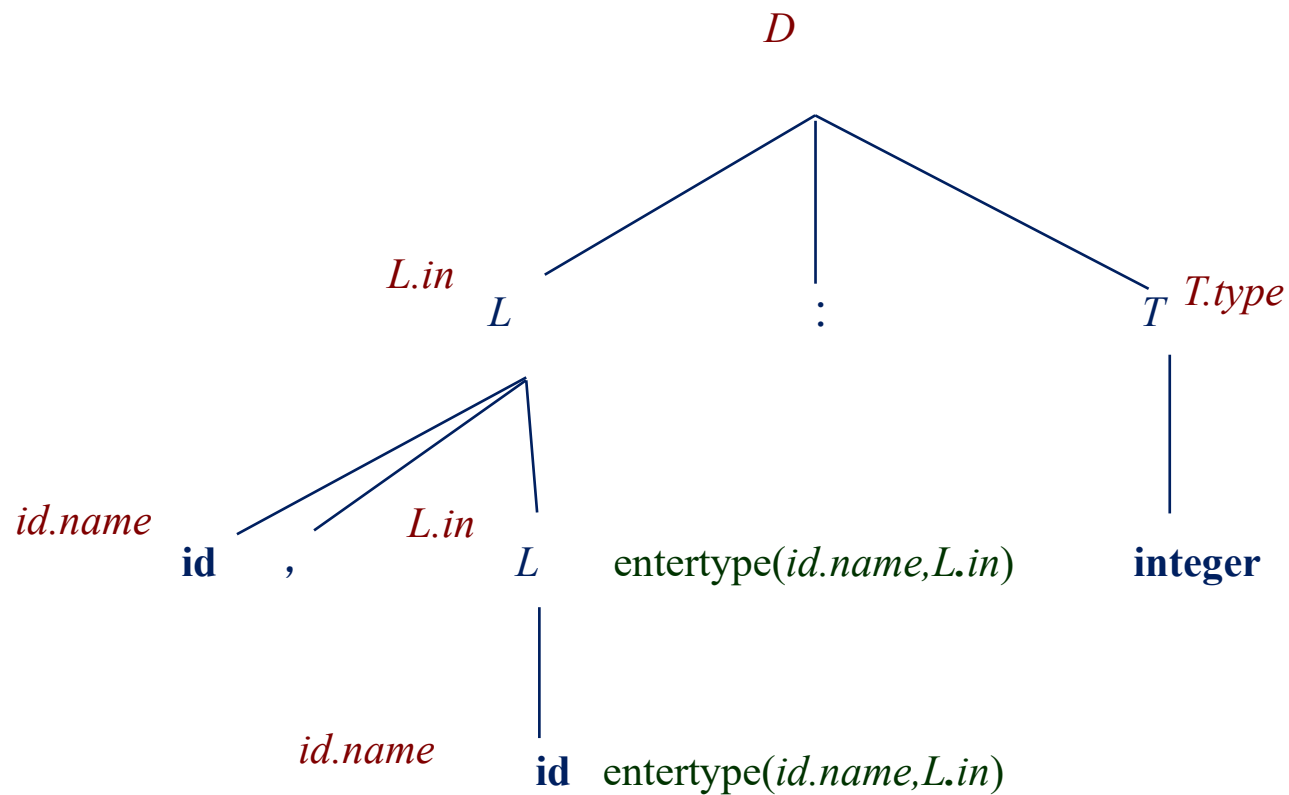
x,y:integer



依赖图

➤ 依赖图

✧ 在语法树
上增加属性



依赖图

➤ 依赖图

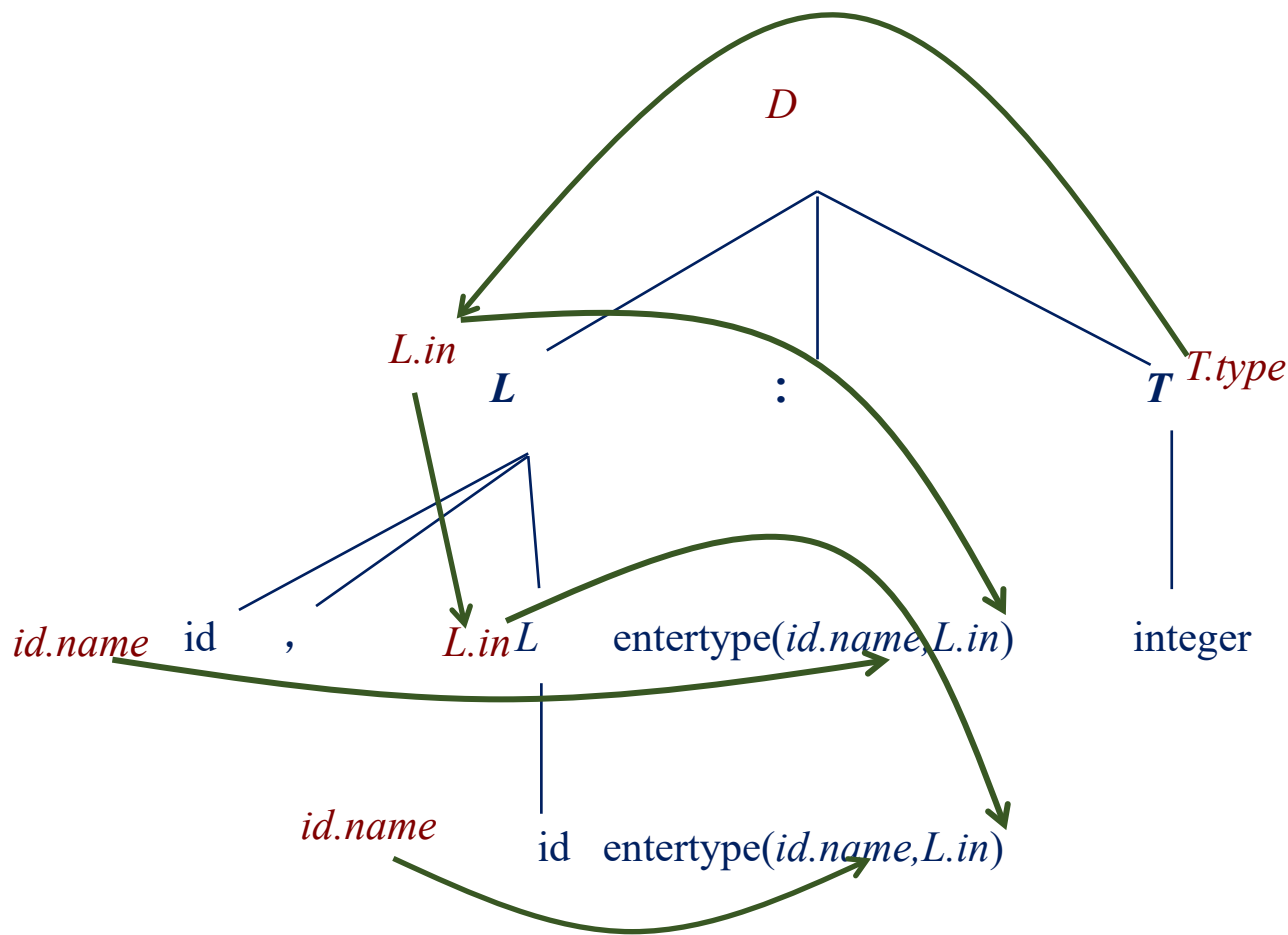
✧ 增加属性间依赖

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

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

$L \rightarrow id, L_1$ $\text{entertype}(id.name, L.in)$
 $L_1.in := L.in$

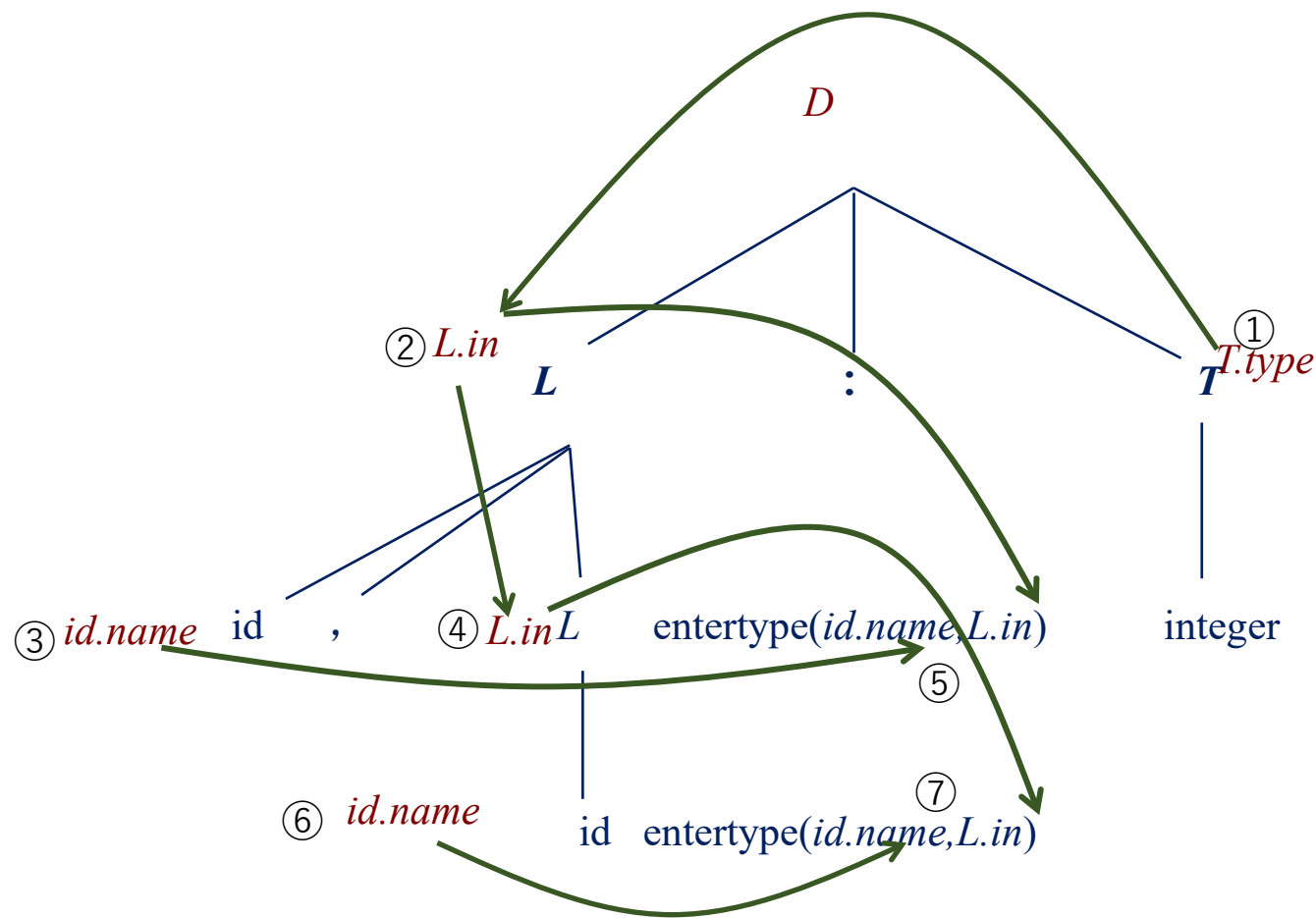
$L \rightarrow id$ $\text{entertype}(id.name, L.in)$



依赖图

➤ 属性计算顺序

✧ 拓扑排序



语义分析的实现

➤ 多遍的组织方式

◇ 语法分析之后进行语义分析

- ✓ 构造输入串的语法分析树
- ✓ 构造依赖图
- ✓ 拓扑排序，若无圈，按顺序对语法树进行遍历并属性

语义分析的实现

► 课堂练习

$S \rightarrow ABC$ { if ($A.num == B.num$) and ($B.num == C.num$)
 then *print*("Accepted!") else *print*("Refused!") }

$A \rightarrow A_1a$ { $A.num = A_1.num + 1$ }

$A \rightarrow a$ { $A.num = 1$ }

$B \rightarrow B_1b$ { $B.num = B_1.num + 1$ }

$B \rightarrow b$ { $B.num = 1$ }

$C \rightarrow C_1c$ { $C.num = C_1.num + 1$ }

$C \rightarrow c$ { $C.num = 1$ }

语义分析的实现

► 课堂练习

$S \rightarrow ABC$ { $B.in_num == A.num$; $C.in_num == A.num$;
if ($B.num == 0$ and ($C.num == 0$))
then $print("Accepted!")$ else $print("Refused!")$ }

$A \rightarrow A_1a$ { $A.num = A_1.num + 1$ }

$A \rightarrow a$ { $A.num = 1$ }

$B \rightarrow B_1b$ { $B_1.in_num = B.in_num$; $B.num = B_1.num - 1$ }

$B \rightarrow b$ { $B.num = B.in_num - 1$ }

$C \rightarrow C_1c$ { $C_1.in_num = C.in_num$; $C.num = C_1.num - 1$ }

$C \rightarrow c$ { $C.num = C.in_num - 1$ }

语义分析的实现

➤ 一遍的组织方式

◇ 语法分析的同时进行属性计算

✓ 自下而上方法

✓ 自上而下方法



关键是什么

语义分析的实现

➤ S-属性文法

- ◇ 只包含综合属性

➤ L-属性文法

- ◇ $\forall A \rightarrow X_1 X_2 \dots X_n \in P$, 其每一个语义规则中的每一个属性都是一个综合属性, 或是 $X_j (1 \leq j \leq n)$ 的一个继承属性, 这个继承属性仅依赖于
 - ✓ 产生式中 X_j 的左边符号 X_1, X_2, \dots, X_{j-1} 的属性;
 - ✓ A 的继承属性。

语义分析的实现

➤ L-属性文法

产生式

$S \rightarrow AB$

$A \rightarrow A_1 a$

$A \rightarrow a$

$B \rightarrow B_1 b$

$B \rightarrow b$

语义规则

$A.in_num := B.num$

$A_1.in_num := A.in_num;$

$A.num := A_1.num - 1$

$A.num := A.in_num - 1$

$B.num := B_1.num + 1$

$B.num := 1$

语法制导翻译

► 翻译模式

- ◇ 翻译模式是语法制导定义的一种补充;
- ◇ 允许由{}括起来的语义动作嵌入到产生式中, 以此来表示每一个语义规则在从左到右的深度优先遍历中的执行时机。
- ◇ 可以显式地表达动作和属性计算的次序

语法制导翻译

➤L-属性文法的翻译模式

- ◇ 产生式右端某个符号的继承属性的计算必须位于该符号之前;
- ◇ 每个计算规则不访问位于它右边符号的综合属性;
- ◇ 产生式左部非终结符的综合属性的计算只能在所用到的属性都已计算出来之后进行, 通常放在相应产生式右端的末尾

语法制导翻译

➤ L-属性文法的翻译模式

$S \rightarrow (L)$ $S.num := L.num + 1$

$S \rightarrow a$ $S.num := 0$

$L \rightarrow SR$ $R.i := S.num ; L.num := R.num$

$R \rightarrow ,S R_1$ $R_1.i := R.i + S.num ; R.num := R_1.num$

$R \rightarrow \varepsilon$ $R.num := R.i$

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

$S \rightarrow a \quad \{ S.num := 0 \}$

$L \rightarrow S \quad \{ R.i := S.num \} R \quad \{ L.num := R.num \}$

$R \rightarrow ,S \quad \{ R_1.i := R.i + S.num \} R_1 \quad \{ R.num := R_1.num \}$

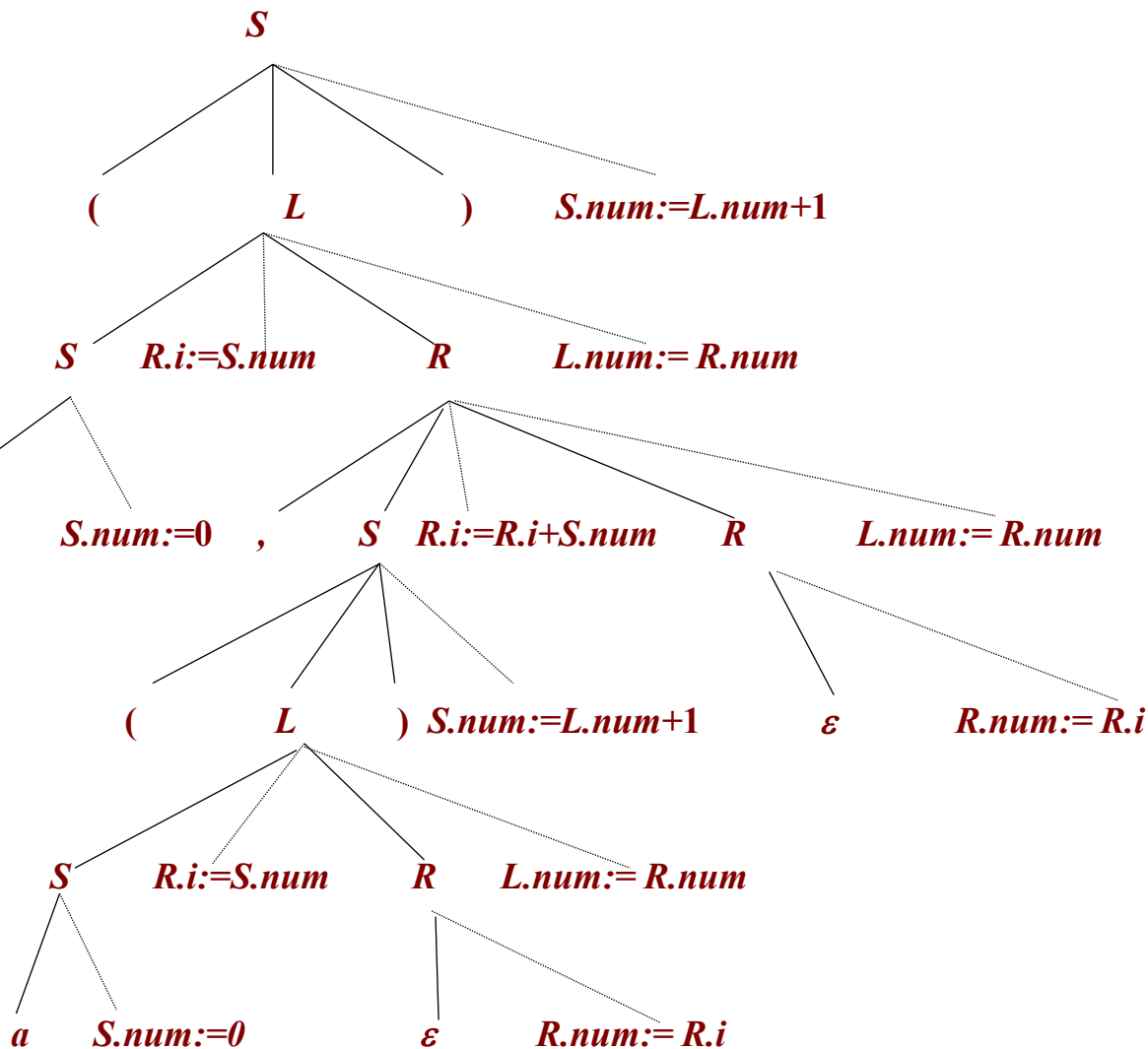
$R \rightarrow \varepsilon \quad \{ R.num := R.i \}$

语法制导翻译

➤ 深度优先后序遍历

✧ (a,(a))

✧ 嵌入语义动作的语法树_a

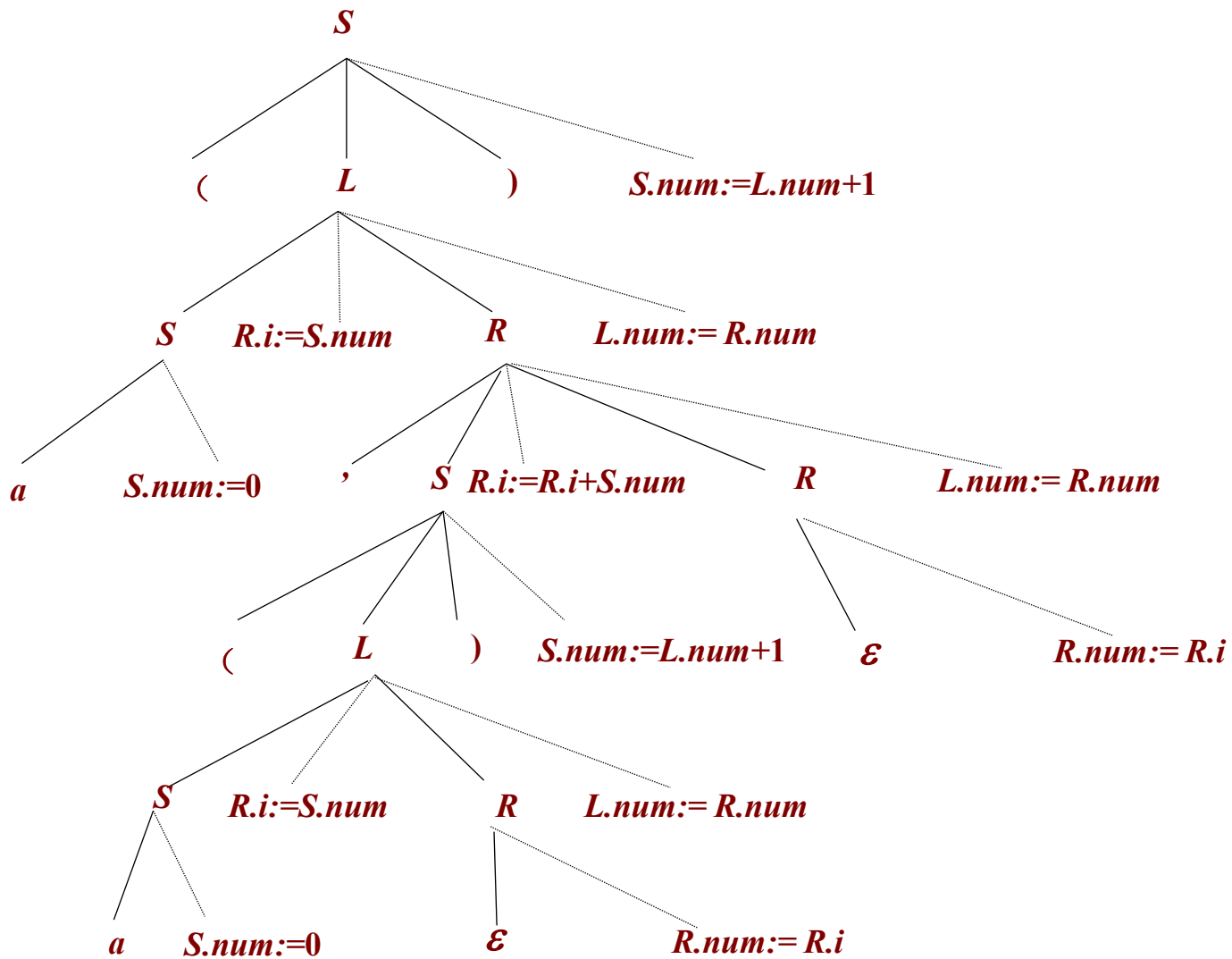


语法制导翻译

深度优先后序遍历

✧ **(a,(a))**

✧ 嵌入语义动作的语法树



语法制导翻译

➤ 在递归下降语法分析过程中进行翻译

- ◇ 对每个非终结符 A ，构造一个函数；
- ◇ 以 A 的每个继承属性为形参，以 A 的综合属性为返回值；
- ◇ 以分支程序实现候选式的选择；
- ◇ 每个分支按照翻译模式的右部依次：
 - ✓ 对终结符 X ，保存其综合属性 x 的值至专为 $X.x$ 声明的变量；调用匹配终结符和取下一符号；
 - ✓ 对非终结符 B ，调用函数 $c := B(b_1, b_2, \dots, b_k)$ ，其中变量 b_1, b_2, \dots, b_k 对应 B 继承属性，变量 c 对应 B 的综合属性；
 - ✓ 对语义规则集，copy 每一语义规则来产生代码，将对属性的访问替换为对相应变量的访问

语法制导翻译

➤ 在递归下降语法分析过程中进行翻译

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

$S \rightarrow a \{S.num := 0\}$

```
int ParseS() {  
    if (token=='(') {  
        gettoken();  
        Lnum=parseL();  
        if (sym==')') gettoken();  
        else error();  
        Snum=Lnum+1  
    }  
    else if (token=='a') {  
        gettoken();  
        Snum =0;  
    }  
    else error();  
    return Snum;  
}
```

语法制导翻译

➤ 在递归下降语法分析过程中进行翻译

```
 $L \rightarrow S \{R.i := S.num\} R \{L.num := R.num\}$   int ParseL() {  
    if (token == '(' || token == 'a') {  
        Snum = parseS();  
        Ri = Snum  
        Rnum = ParseR(Ri);  
        Lnum = Rnum;  
    }  
    else error();  
}
```


语法制导翻译

➤ 在递归下降语法分析过程中进行翻译

$R \rightarrow , S \{ R_1.i := R.i + S.num \} R_1 \{ R.num := R_1.num \}$

$R \rightarrow \varepsilon \{ R.num := R.i \}$

```
int ParseR( int Ri) {  
    if (token==' , ' ) {  
        gettoken();  
        Snum=parseS();  
        R1i = Ri+Snum  
        R1num=ParseR(R1i);  
        Rnum=R1num;  
    }  
    else if (token==' ) ' ) Rnum=Ri  
    else error();  
    return Rnum;  
}
```

符号表

➤ 符号表的作用

◇ 符号表的作用是将信息从声明的地方传递到实际使用的地方，
为静态语义检查和代码生成提供辅助信息。

✓ 静态语义检查（作用域，唯一性，类型检查）

✓ 代码生成提供辅助信息（对符号名进行地址分配的依据）

符号表

➤ 符号的属性

◇ 符号名

◇ 符号的类型

◇ 符号的地址

◇ 其他属性

数组内情向量

记录结构的成员信息

函数及过程的形参

名字		内情向量指针
$\overleftrightarrow{\dots}$			
A			
.....			
B			

A 的内情向量空间

l_1	u_1	d_1
1	C	$@A$

B 的内情向量空间

l_1	u_1	d_1
l_2	u_2	d_2
l_3	u_3	d_3
3	C	$@B$

符号表

➤ 符号表的设计

✧ 符号表的组织

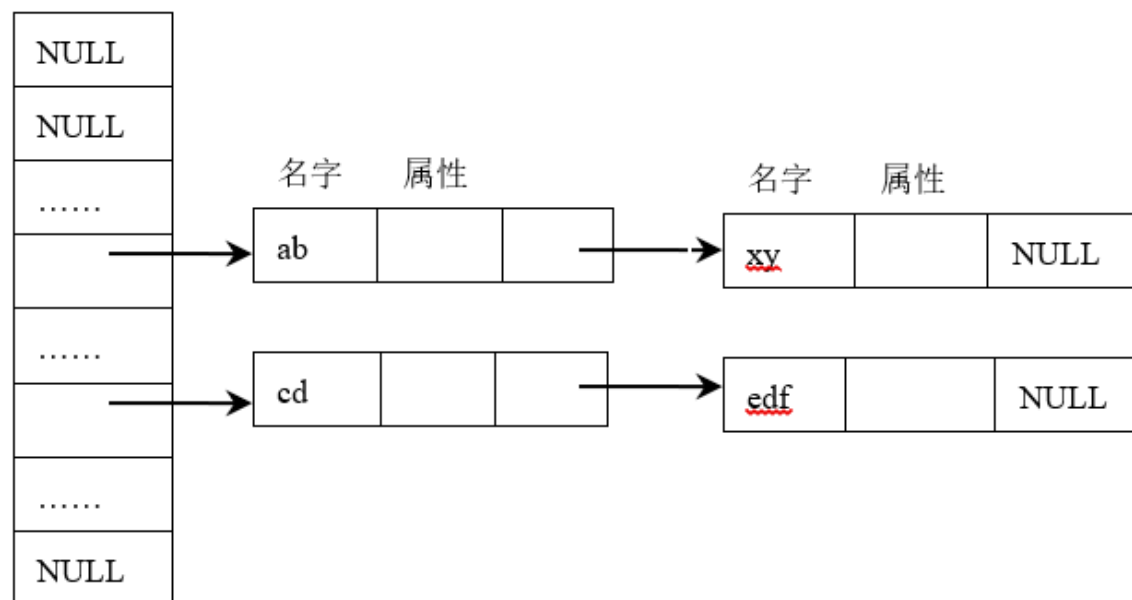
✓ 一个符号表

✓ 多个符号表

✧ 符号表的数据结构

✓ 线性表：数组，链表

✓ 散列表



符号表

➤ 符号表的管理

- ◇ 创建符号表：在编译开始，或进入一个分程序
- ◇ 插入表项：在遇到新的标识符声明时进行
- ◇ 查询表项：在引用标识符时进行
- ◇ 修改表项：在获得新的语义值信息时进行
- ◇ 删除表项：在标识符成为不可见/不再需要它的任何信息时进行
- ◇ 释放符号表空间：在编译结束前或退出一个分程序

符号表

➤ 嵌套作用域的管理

◇ 块结构

- ✓ 块结构(block, 又称为分程序): 是本身含有局部变量声明的语句
- ✓ C语言的程序块的语法为: {声明 语句}

符号表

➤ 嵌套作用域的管理

```
void main()
{int x=0; int y=1;
  {int y=1;
    {int x=2;
      x=x+y;
    }
    {int y=3;
      x=x+y;
    }
  }
  x=x+y;
}
```

```
program A;
  var x, y: integer;
  procedure P;
    var k:array[1..9] of real;
    function f(i:real):integer;
      var j:integer;
      begin ..... end;
    begin ..... end;
  procedure R;
    var x:real;
    begin .....
  end;
begin ..... end.
```

符号表

➤ 嵌套作用域的管理

✧ 作用域

- ✓ 定义一个以上变量的静态程序结构单元

- ✓ 嵌套的作用域

该点所在的作用域为 **当前作用域**

当前作用域与包含它的程序单元所构成的作用域称为 **开作用域**

不属于开作用域的作用域称为 **闭作用域**

符号表

➤ 嵌套作用域的管理

✧ 可见性

- ✓ 在程序的任何一点，只在该点的开作用域中声明的名字才是可访问的
- ✓ 若一个名字在多个开作用域中被声明，则把离该名字的某个引用最近的声明作为该引用的解释

符号表

➤ 嵌套作用域的管理

void main()

```
{int x=0; int y=1;  
  {int y=1;  
    {int x=2;  
      x=x+y;  
    }  
    {int y=3;  
      x=x+y;  
    }  
    x=x+y;  
  }  
  x=x+y;  
}
```

program A;

```
var x, y: integer;  
procedure P;  
  var k:array[1..9] of real;  
  function f(i:real):integer;  
    var j:integer;  
    begin ..... end;  
  begin ..... end;  
  procedure R;  
    var x:real;  
    begin ..... end;  
  begin ..... end.
```

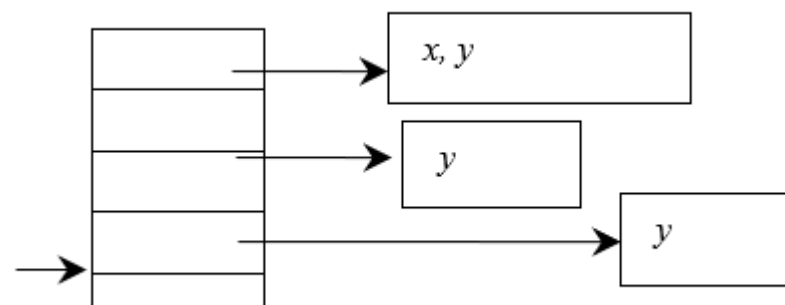
符号表

➤ 块结构中符号表的实现

◇ 独立符号表

✓ 一个作用域存一个表格

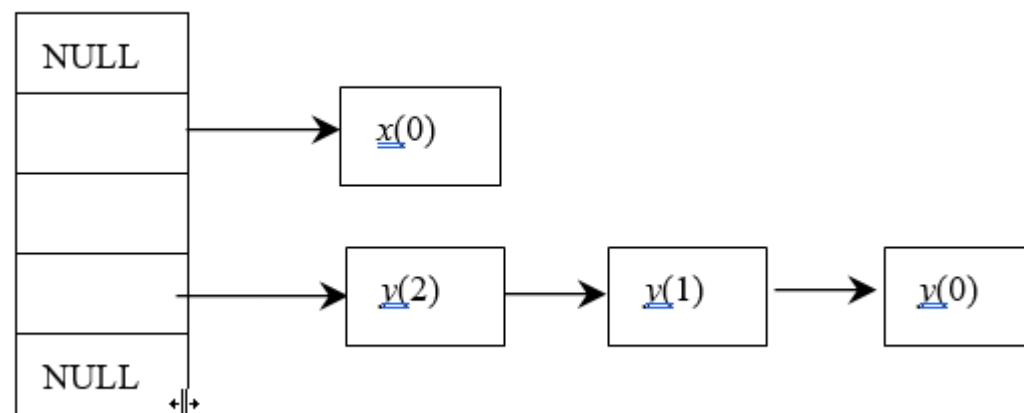
编译到第二个 $x=x+y$



◇ 单一全局符号符号表

✓ 所有符号存一个表格

✓ 作用域号



符号表

► 案例分析: *Pascal*风格语言

<i>name</i>	<i>kind</i>	<i>...</i>	<i>address</i>	<i>type</i>	<i>level</i>
<i>x</i>	<i>variable</i>			<i>int</i>	0
<i>y</i>	<i>variable</i>			<i>int</i>	0
<i>P</i>	<i>procname</i>				0
<i>k</i>	<i>variable</i>			<i>real</i>	1
<i>f</i>	<i>funname</i>			<i>real</i>	1
<i>i</i>	<i>parname</i>			<i>int</i>	2
<i>j</i>	<i>variable</i>			<i>int</i>	2

```
var x, y: integer;
procedure P;
    var k:array[1..9] of real;
    function f(i:real):integer;
        var j:integer;
        begin
            if i>5 then f:=1 else f:=2;
        end;
    begin
        k[1]:=6.5;
        x:=f(k)+x
    end;
procedure R;
    var x:real;
    begin
        x:=1.5;
        P
    end;
begin
    x:=1;
    y:=2;
    R
end.
```

符号表

➤ 案例分析：C风格语言

<i>name</i>	<i>kind</i>	<i>...</i>	<i>address</i>	<i>type</i>	<i>level</i>
<i>x</i>	<i>variable</i>			<i>int</i>	0
<i>y</i>	<i>variable</i>			<i>int</i>	0
<i>y</i>	<i>variable</i>			<i>int</i>	1
<i>x</i>	<i>variable</i>			<i>int</i>	2

```
void main()  
    {int x=0; int y=1;  
        {int y=1;  
            {int x=2;  
                x=x+y;  
            }  
            {int y=3;  
                x=x+y;  
            }  
        }  
        x=x+y;  
    }  
x=x+y;  
}
```

符号表

- 某高级语言只有整型一种数据类型，可以声明常量、变量、过程和数组，过程可以嵌套定义
- 请给出一个全局符号表设计方案

```
(1) const a=10;  
(2)  var x(1:5),b;  
(3)  procedure p;  
(4)      var a,b,z;  
(5)      begin  
(6)          z:=a+b  
(7)      end;  
(8)  begin  
(9)      call p;  
(10)   x(2):=a  
(11) end.
```

类型检查

➤ 类型检查

- ✧ **作用：**根据类型系统和类型推断对程序中类型信息的计算和维护，确保程序的每一部分是有意义的
- ✧ **声明：**符号的属性来源于符号出现的上下文中显式或者隐式的声明，编译器通过符号表将属性和符号的名字关联起来。
- ✧ **类型系统：**类型集合、描述程序行为的类型规则统称类型系统

类型检查

➤ 声明

$P \rightarrow \text{program id} \{ D.offset := 0 \} D; S$

$D \rightarrow \{ D_1.offset := D.offset \} D_1; \{ D_2.offset := D.offset + D_1.width \}$

$D_2 \{ D.width := D_1.width + D_2.width \}$

$D \rightarrow \text{id} : T \{ \text{enter}(\text{id.name}, T.type, D.offset); D.width := T.width \}$

类型检查

➤ 声明

$T \rightarrow \text{boolean} \{ T.type := \text{char}; T.width := 1 \}$

$T \rightarrow \text{integer} \{ T.type := \text{int}; T.width := 4 \}$

$T \rightarrow \text{real} \{ T.type := \text{real}; T.width := 8 \}$

$T \rightarrow ^T_1 \{ T.type := \text{pointer}(T_1.type); T.width := 4 \}$

$T \rightarrow \text{array}[\text{num}] \text{ of } T_1 \{ T.type := \text{array}(\text{num.val}, T_1.width);$

$T.width := \text{num.val} \times T_1.width \}$

类型检查

➤ 句型

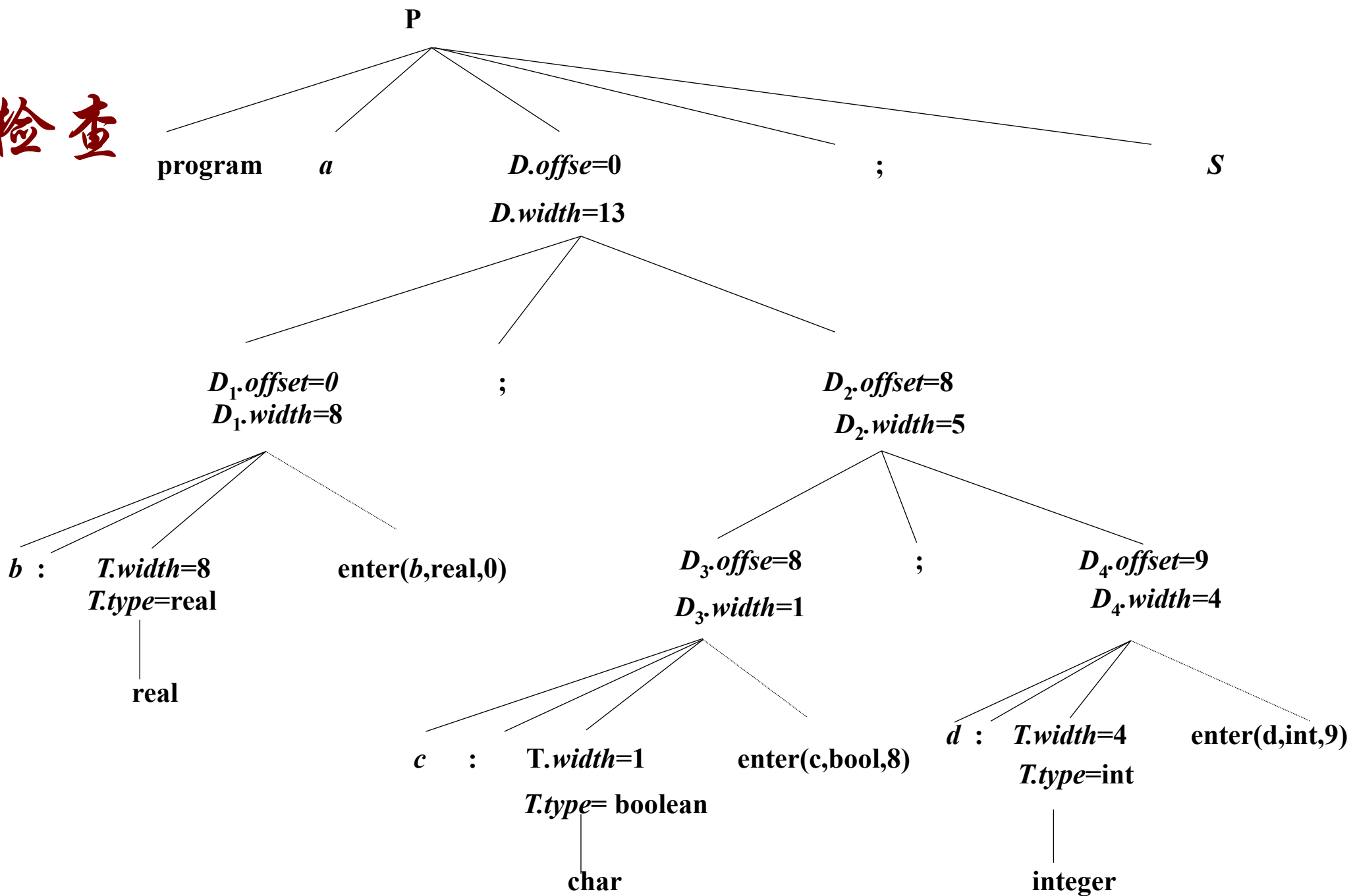
program *a*

b: **real**;

c: **Boolean**;

d:**integer**;

S



类型检查

➤ 类型表达式

◇ 基本类型：基本类型是程序语言预定义的类型。

- ✓ 典型的基本类型包括bool、char、int、real、type_error和void;
- ✓ type_error用于类型检查时出现类型错误;
- ✓ 而void表示无值类型，可用于语句的类型检查。

类型检查

➤ 类型检查

$E \rightarrow \mathbf{true}$ $\{E.type := \mathbf{bool}\}$

$E \rightarrow \mathbf{false}$ $\{E.type := \mathbf{bool}\}$

$E \rightarrow \mathbf{num}$ $\{E.type := \mathbf{int}\}$

$E \rightarrow \mathbf{rnum}$ $\{E.type := \mathbf{real}\}$

$E \rightarrow \mathbf{id}$ $\{E.type := \mathbf{lookup_type(id.name)}\}$

类型检查

➤ 类型检查

$E \rightarrow E \text{ op } E \quad \{ E.type := \text{if } E_1.type = \text{real and } E_2.type = \text{real then real}$
else if $E_1.type = \text{int and } E_2.type = \text{int}$
then int else type_error}

$E \rightarrow E \text{ rop } E \quad \{ E.type := \text{if } E_1.type = \text{real and } E_2.type = \text{real then bool}$
else if $E_1.type = \text{int and } E_2.type = \text{int}$
then bool else type_error}

类型检查

➤ 类型检查

$E \rightarrow E[E] \{ E.type := \text{if } E_2.type = \text{int and } E_1.type = \text{array}(s, t) \text{ then } t \text{ else type_error} \}$

$E \rightarrow E^* \{ E.type := \text{if } E_1.type = \text{pointer}(t) \text{ then } t \text{ else type_error} \}$

$S \rightarrow \text{id} := E \{ S.type := \text{if lookup_type}(\text{id.name}) = E.type \text{ then void else type_error} \}$

$S \rightarrow \text{if } E \text{ then } S \{ S.type := \text{if } E.type = \text{bool} \text{ then } S_1.type \text{ else type_error} \}$

$S \rightarrow \text{while } E \text{ do } S \{ S.type := \text{if } E.type = \text{bool} \text{ then } S_1.type \text{ else type_error} \}$

$S \rightarrow S; S \{ S.type := \text{if } S_1.type = \text{void and } S_2.type = \text{void} \text{ then void else type_error} \}$

类型检查

➤ 类型转换

◇ 不同数据类型的数据之间进行运算的结果类型

$$E \rightarrow E_1 \text{ op } E_2 \{ \begin{array}{l} E.type := \text{if } E_1.type = \text{real and } E_2.type = \text{real then real} \\ \text{else if } E_1.type = \text{int and } E_2.type = \text{int then int} \\ \text{else if } E_1.type = \text{real and } E_2.type = \text{int then real} \\ \text{else if } E_1.type = \text{int and } E_2.type = \text{real then real} \\ \text{else type_error} \end{array} \}$$

语法制导定义的设计

- 将每个变量名及其类型信息填入符号表
- 输出共说明多少个变量

$$S \rightarrow D$$

$$D \rightarrow \text{integer id} \mid \text{real id} \mid D_1, \text{id}$$