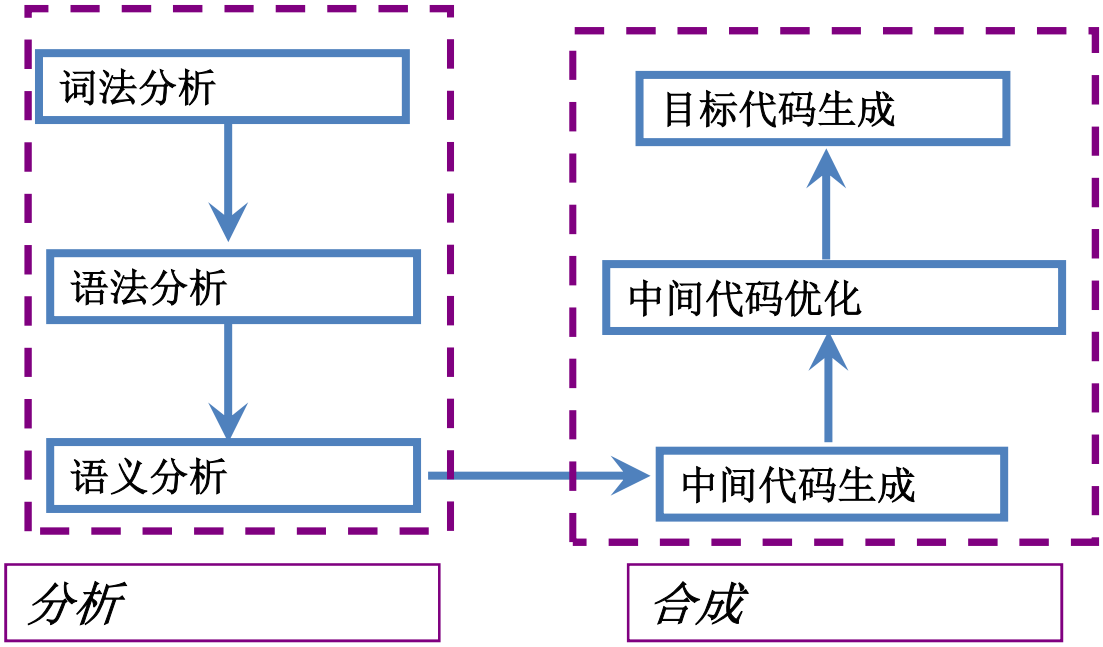
# Lecture 9: 语法制导的翻译-I

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

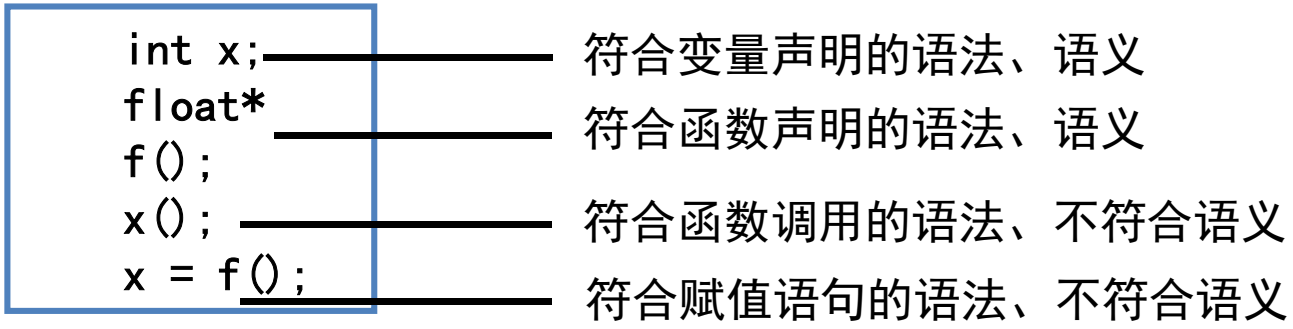

# 9.1 概述

# 语义分析在编译程序中的作用



# 语法和语义的区别

- **语法：**
  - **是描述一个合法定义的程序结构的规则**
  - **例如：<函数调用语句> → id( <实参表达式>)**
- **语义：**
  - **说明一个合法定义的程序的含义**



## 语义分析的必要性

- 一个语法正确的程序不能保证它是有意义的！
- 程序中容易出现各种语义错误：
  - 标识符未声明　y = x+3;
  - 操作数的类型与操作符的类型不匹配 y = x*3;
  - 数组下标变量的类型出错　A[x];
  - ……

语义分析不能检查程序在计算逻辑上的错误！！！！

# 程序设计语言语义的分类

- **静态语义(static semantics)**
  - 编译时(compile-time)可以检查的语义
  - 例如：标识符未声明
    标识符重复声明
- **动态语义(dynamics semantics)**
  - 目标程序运行时（run-time）才能检查的语义
  - 例如：除零溢出错误 x/(y-i); 当 y = i时
    数组下标越界 int a[5]; a[5] =0;
    无效指针 int*p; p =NULL; *p = 3;

## 如何描述程序设计语言的语义？

- 程序设计语言的形式语义
  - 属性文法 (用于描述静态语义)
  - 操作语义(Operational Semantics)
  - 指称语义(Denotational Semantics)
  - 代数语义(Algebra Semantics)
  - 公理语义(Axiomatic Semantics)
- 形式语义技术没有形式语法技术成熟
- 硕士研究生的课程--《形式语义学》


# 9.2 基本概念

# 什么是语法制导翻译

- **语法制导翻译：在语法分析基础上边分析边翻译**
  - 翻译的依据：语义规则或语义子程序
  - 翻译的结果：相应的中间代码
  - 翻译的做法：为每个产生式配置相应的语义子程序，每当使用某个产生式进行规约或推导时，就调用语义子程序，完成一部分翻译工作
  - 语法分析完成时，翻译工作也告结束

# 语义规则

- **产生式与语义规则**
  - 每个文法产生式$A \rightarrow \alpha$有一组形式为$b=f(c_1, c_2, \ldots, c_k)$的语义规则，其中$b$和$c_1, c_2, \ldots, c_k$是该产生式文法符号的属性，$f$是函数
  - 描述一个产生式所对应的翻译工作，如：
    - 改变某些变量的值；填/查各种符号表；发现并报告源程序错误；产生中间代码
    - 决定于要产生什么样的中间代码

# 文法属性

- **文法属性：每个文法符号有一组属性**
  - X是一个文法符号，a是X的一个属性，用X.a表示a在某个标号为X的分析树结点上的属性值
  - 属性可以有多种类型，如num, type, table_entry, text等。
- **两种类型**
  - 综合属性：如果*b*是*A*的属性，$c_1$，$c_2$，…，$c_k$ 是产生式右部文法符号的属性或*A*的其它属性
  - 继承属性：如果*b*是右部某文法符号*X*的属性

## 基本概念(skip, 将在后面讨论)

- **综合属性**
  - 在分析树结点N上的非终结符号A的属性值由N对应的产生式所关联的语义规则来定义，通过N的子结点或N本身的属性值来定义
- **继承属性**
  - 结点N的属性值由N的父结点所关联的语义规则来定义，依赖于N的父结点、N本身和N的兄弟结点上的属性值

不允许N的继承属性通过N的子结点上的属性来定义, 但是允许N的综合属性依赖于N本身的继承属性, 终结符号有综合属性(由词法分析获得)，但是没有继承属性

# 语法制导定义

- **语法制导(的文法)定义是一个CFG和属性及规则的结合**
  - Syntax-Directed Definition, SDD
  - 属性和文法符号相关联，规则与产生式相关联

只含有综合属性的例子

| | 产生式 | 语义规则 |
|---|---|---|
| 1) | $L \to E$ **n** | $L.val = E.val$ |
| 2) | $E \to E_1 + T$ | $E.val = E_1.val + T.val$ |
| 3) | $E \to T$ | $E.val = T.val$ |
| 4) | $T \to T_1 * F$ | $T.val = T_1.val \times F.val$ |
| 5) | $T \to F$ | $T.val = F.val$ |
| 6) | $F \to ( E )$ | $F.val = E.val$ |
| 7) | $F \to$ **digit** | $F.val =$ **digit**.lexval |

# 语法分析树上的SDD求值

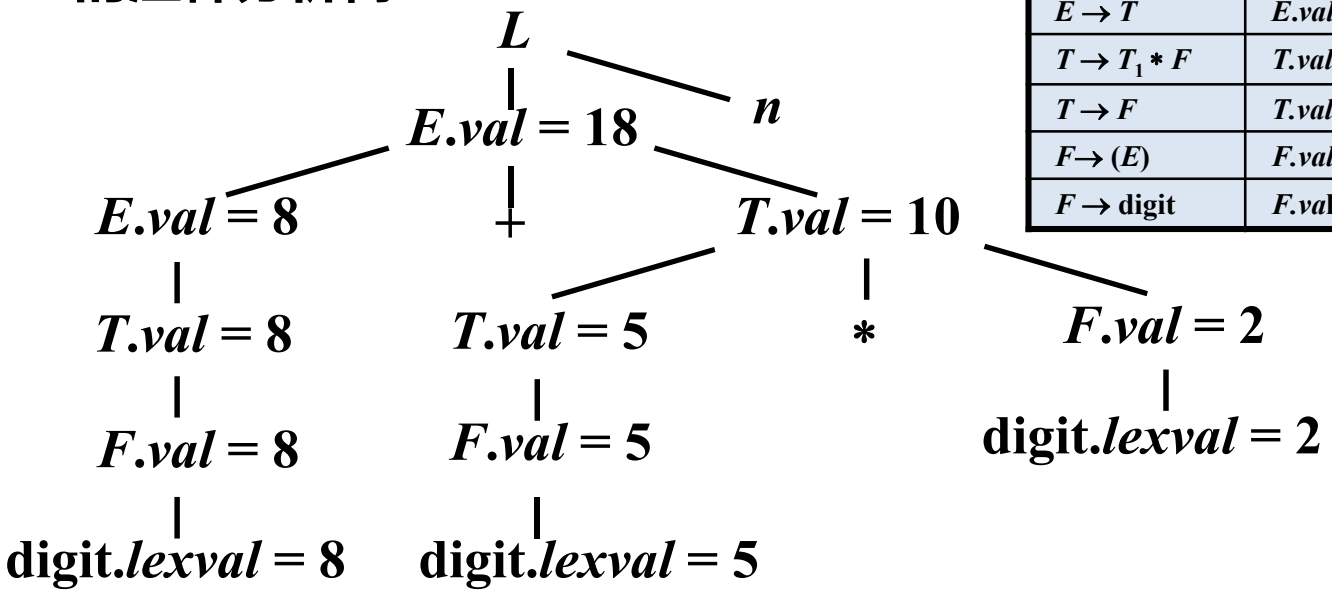
- **实践中很少先构造语法分析树再进行SDD求值**
- **但在分析树上求值有助于翻译方案的可视化，便于理解**
- **注释语法分析树**
  - 包含了各个结点的各属性值的语法分析树
- **步骤：**
  - 对于任意的输入串，首先构造出相应的分析树。
  - 给各个结点（根据其文法符号）加上相应的属性值
  - 按照语义规则计算这些属性值即可

# 语法分析树上的SDD求值

## 注释分析树：结点的属性值都标注出来的分析

| 产 生 式 | 语 义 规 则 |
|---|---|
| $L \rightarrow E$ n | *print* (*E.val*) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | *E.val* = *T.val* |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | *T.val* = *F.val* |
| $F \rightarrow (E)$ | *F.val* = *E.val* |
| $F \rightarrow$ digit | *F.val* = digit.*lexval* |

**8+5*2 n的注释分析树**

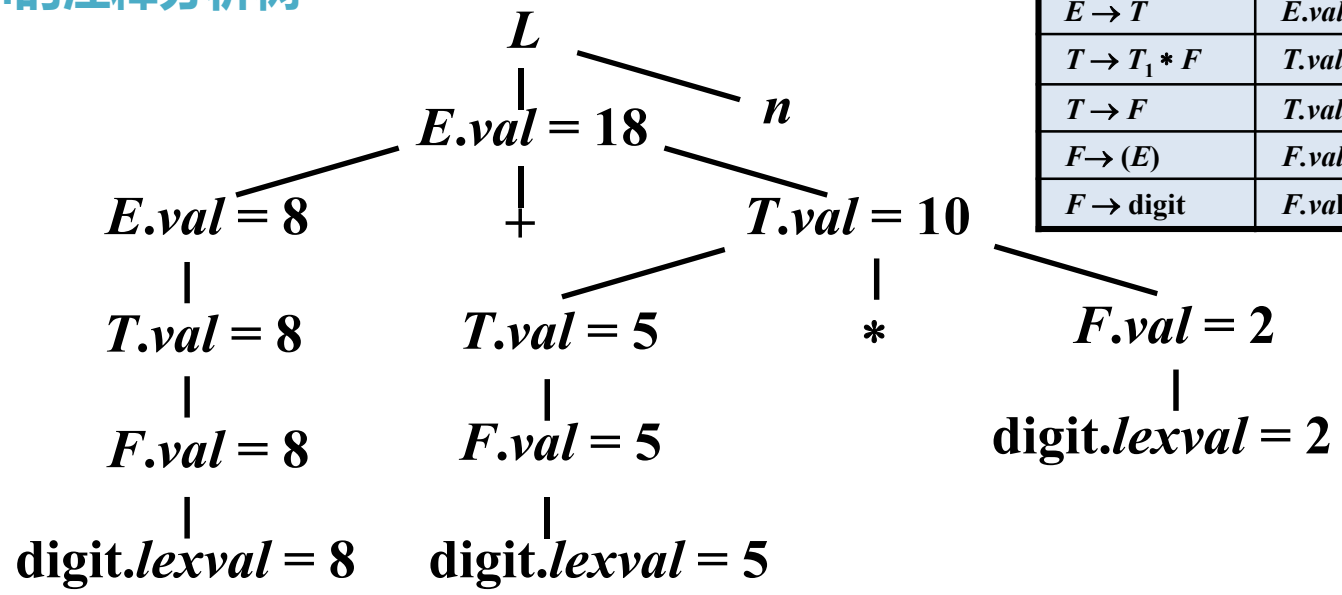


# 语法分析树上的SDD求值

S属性SDD的分析树各结点属性的计算可以自下而上地

| 产 生 式 | 语 义 规 则 |
|---|---|
| $L \rightarrow E$ n | *print* (*E.val*) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | *E.val* = *T.val* |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | *T.val* = *F.val* |
| $F \rightarrow (E)$ | *F.val* = *E.val* |
| $F \rightarrow$ digit | *F.val* = digit.*lexval* |

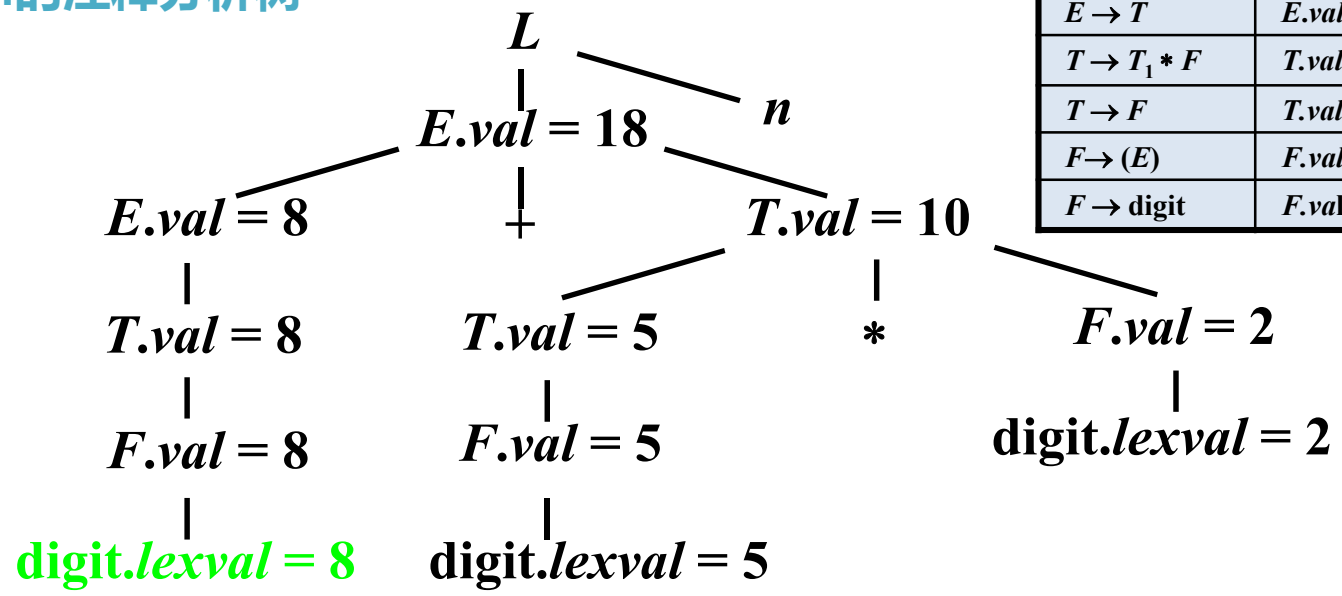
8+5*2 n的注释分析树



# 语法分析树上的SDD求值

## 分析树各结点属性的计算可以自下而上地完成

8+5*2 n的注释分析树

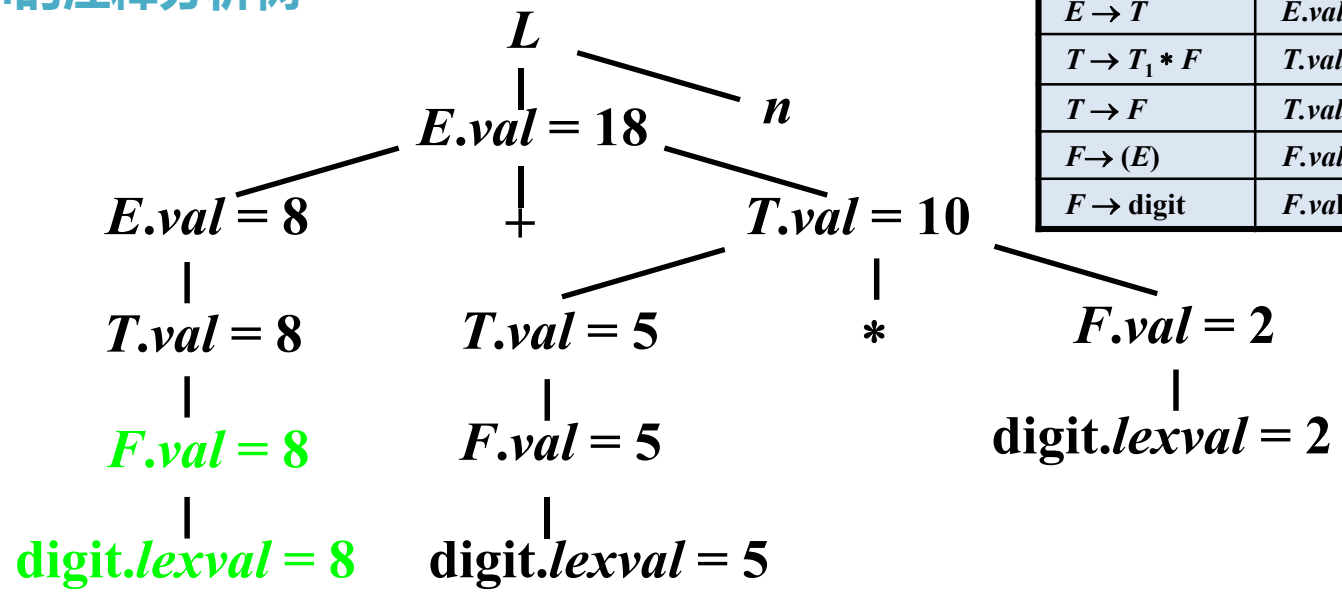



| 产 生 式 | 语 义 规 则 |
|---|---|
| $L \rightarrow E$ n | *print* (*E.val*) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | *E.val* = *T.val* |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | *T.val* = *F.val* |
| $F \rightarrow (E)$ | *F.val* = *E.val* |
| $F \rightarrow$ digit | *F.val* = digit.*lexval* |

# 语法分析树上的SDD求值

## 分析树各结点属性的计算可以自下而上地完成

8+5*2 n的注释分析树

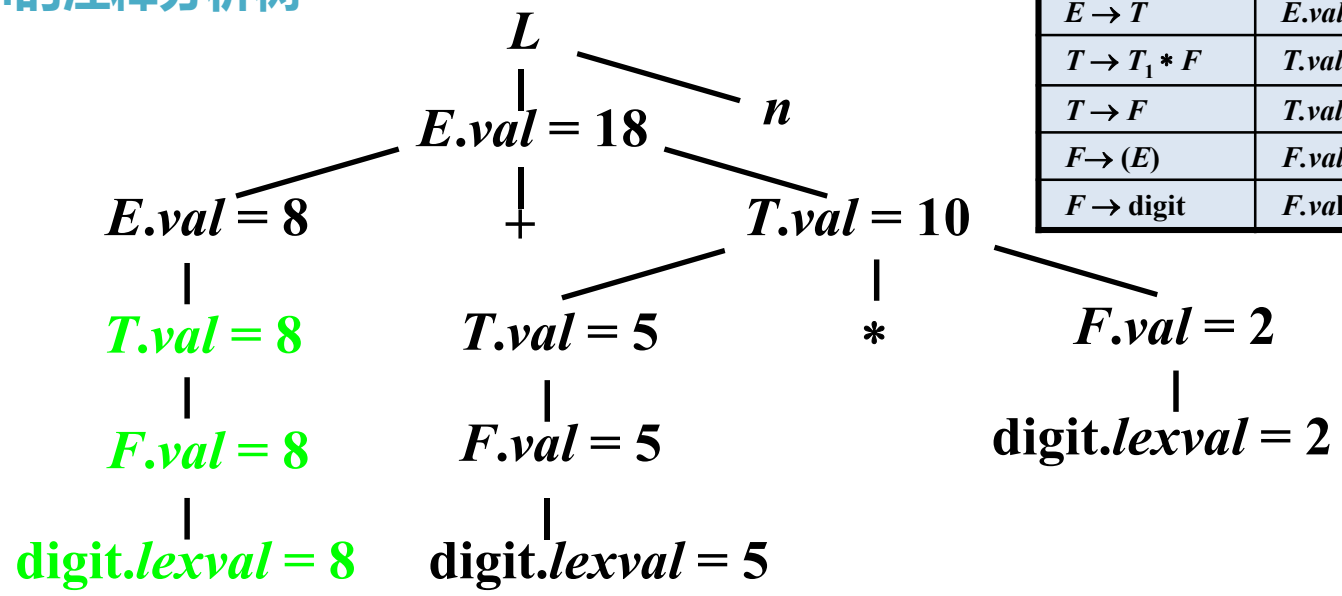



| 产 生 式 | 语 义 规 则 |
|---|---|
| $L \rightarrow E$ n | *print* (*E.val*) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | *E.val* = *T.val* |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | *T.val* = *F.val* |
| $F \rightarrow (E)$ | *F.val* = *E.val* |
| $F \rightarrow$ digit | *F.val* = digit.*lexval* |

# 语法分析树上的SDD求值

## 分析树各结点属性的计算可以自下而上地完成

8+5*2 n的注释分析树

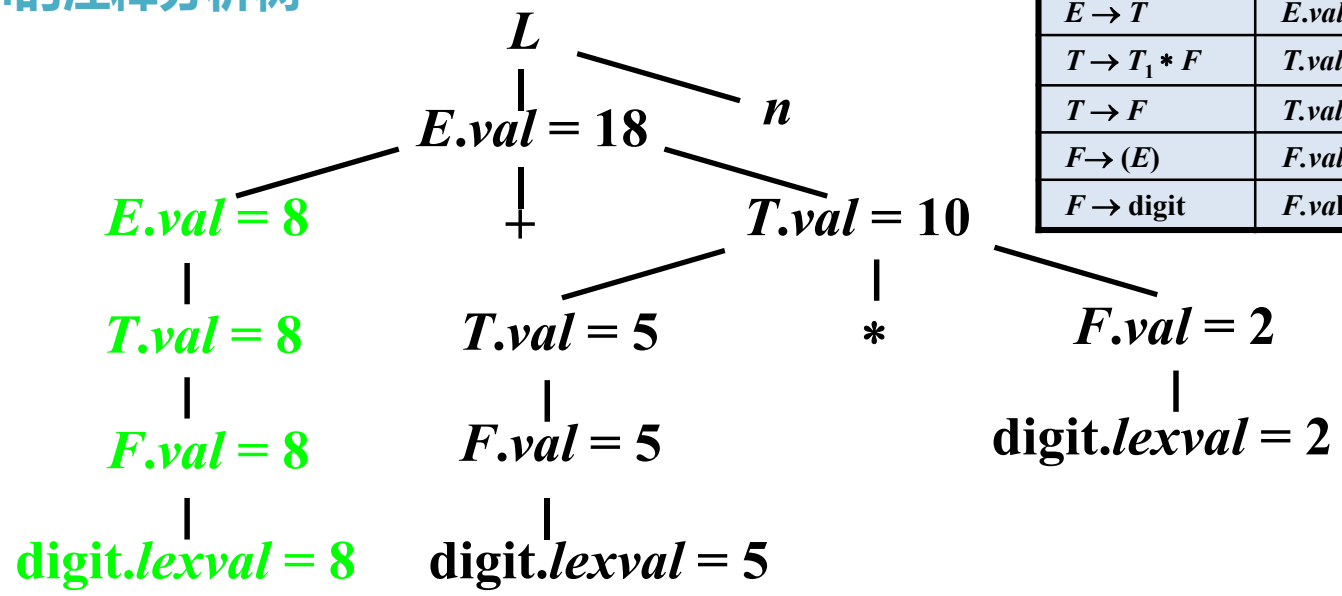



| 产 生 式 | 语 义 规 则 |
|---|---|
| $L \rightarrow E$ n | *print* (*E.val*) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | *E.val* = *T.val* |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | *T.val* = *F.val* |
| $F \rightarrow (E)$ | *F.val* = *E.val* |
| $F \rightarrow$ digit | *F.val* = digit.*lexval* |

# 语法分析树上的SDD求值

## 分析树各结点属性的计算可以自下而上地完成

8+5*2 n的注释分析树

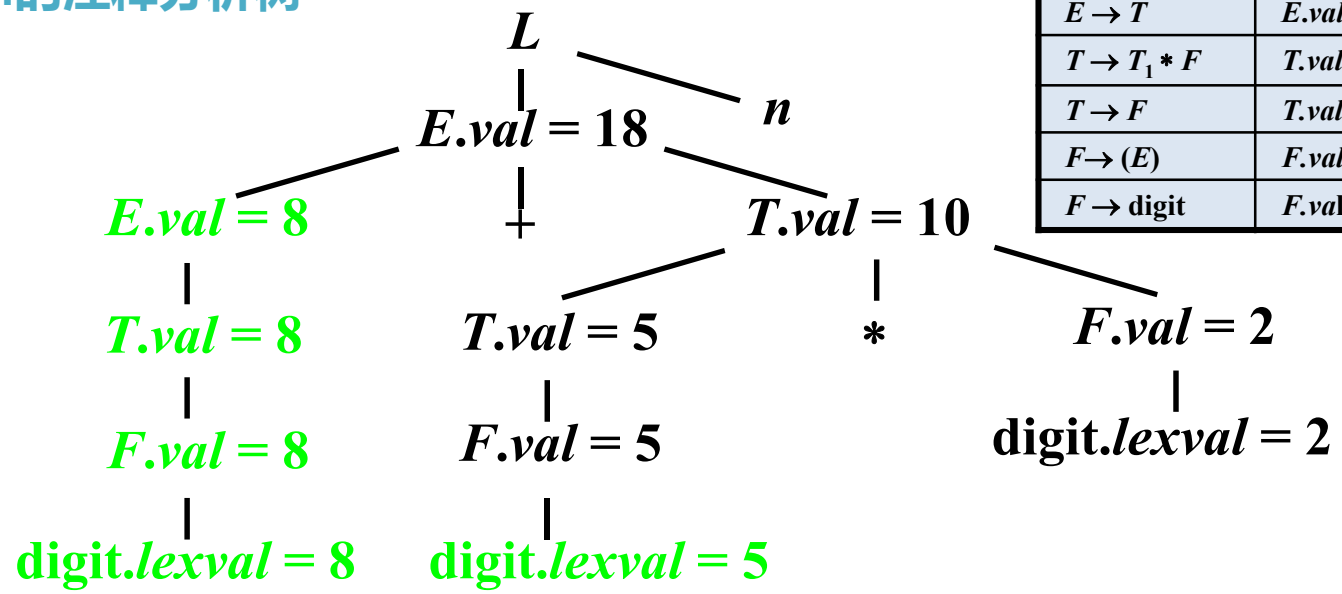



| 产 生 式 | 语 义 规 则 |
|---|---|
| $L \rightarrow E$ n | *print* (*E.val*) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | *E.val* = *T.val* |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | *T.val* = *F.val* |
| $F \rightarrow (E)$ | *F.val* = *E.val* |
| $F \rightarrow$ digit | *F.val* = digit.*lexval* |

# 语法分析树上的SDD求值

## 分析树各结点属性的计算可以自下而上地完成

8+5*2 n的注释分析树

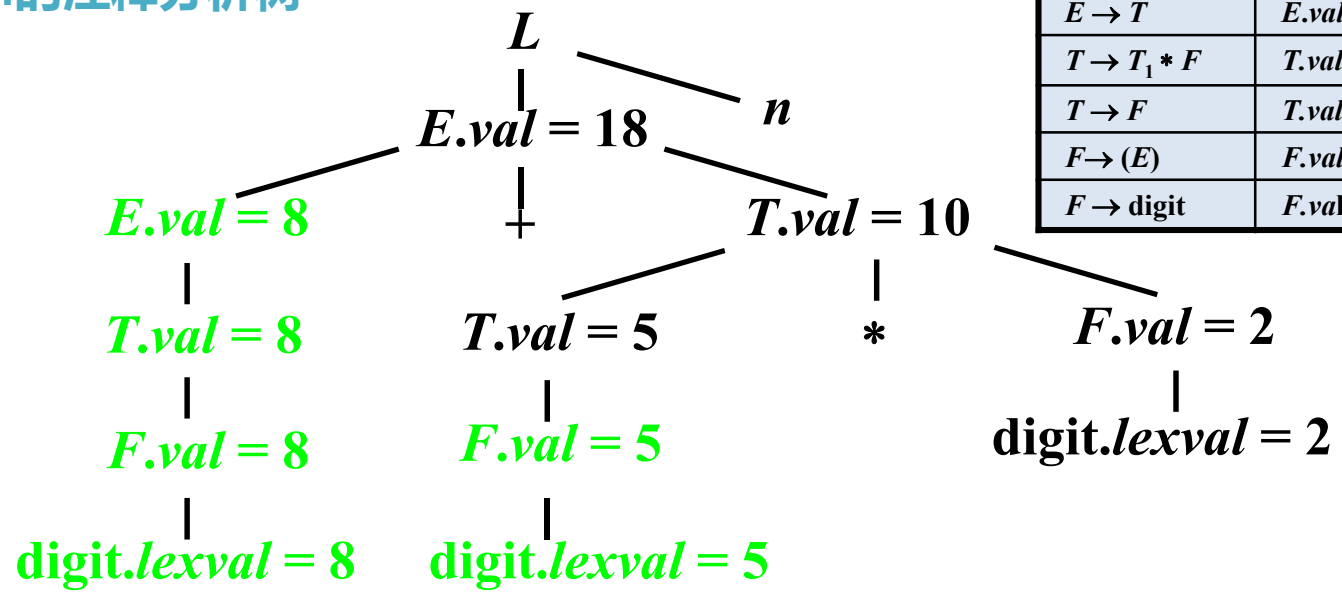



| 产 生 式 | 语 义 规 则 |
|---|---|
| $L \rightarrow E$ n | *print* (*E.val*) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | *E.val* = *T.val* |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | *T.val* = *F.val* |
| $F \rightarrow (E)$ | *F.val* = *E.val* |
| $F \rightarrow$ digit | *F.val* = digit.*lexval* |

# 语法分析树上的SDD求值

## 分析树各结点属性的计算可以自下而上地完成

8+5*2 n的注释分析树

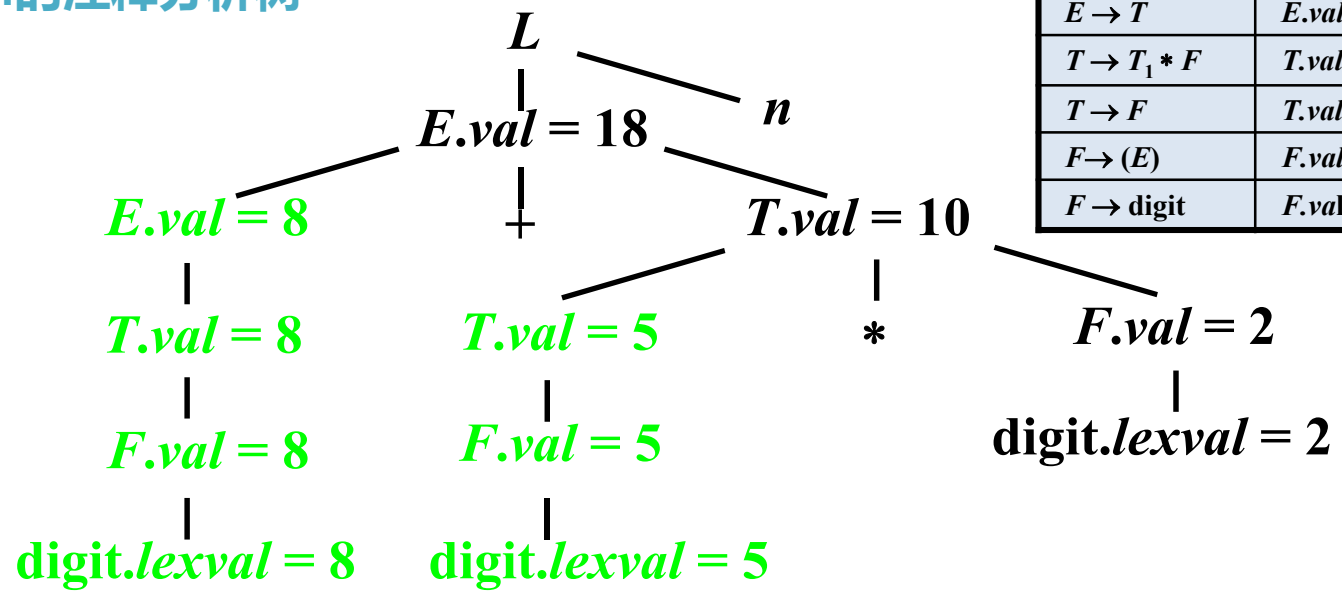



| 产 生 式 | 语 义 规 则 |
|---|---|
| $L \rightarrow E$ n | *print* (*E.val*) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | *E.val* = *T.val* |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | *T.val* = *F.val* |
| $F \rightarrow (E)$ | *F.val* = *E.val* |
| $F \rightarrow$ digit | *F.val* = digit.*lexval* |

# 语法分析树上的SDD求值

## 分析树各结点属性的计算可以自下而上地完成

8+5*2 n的注释分析树

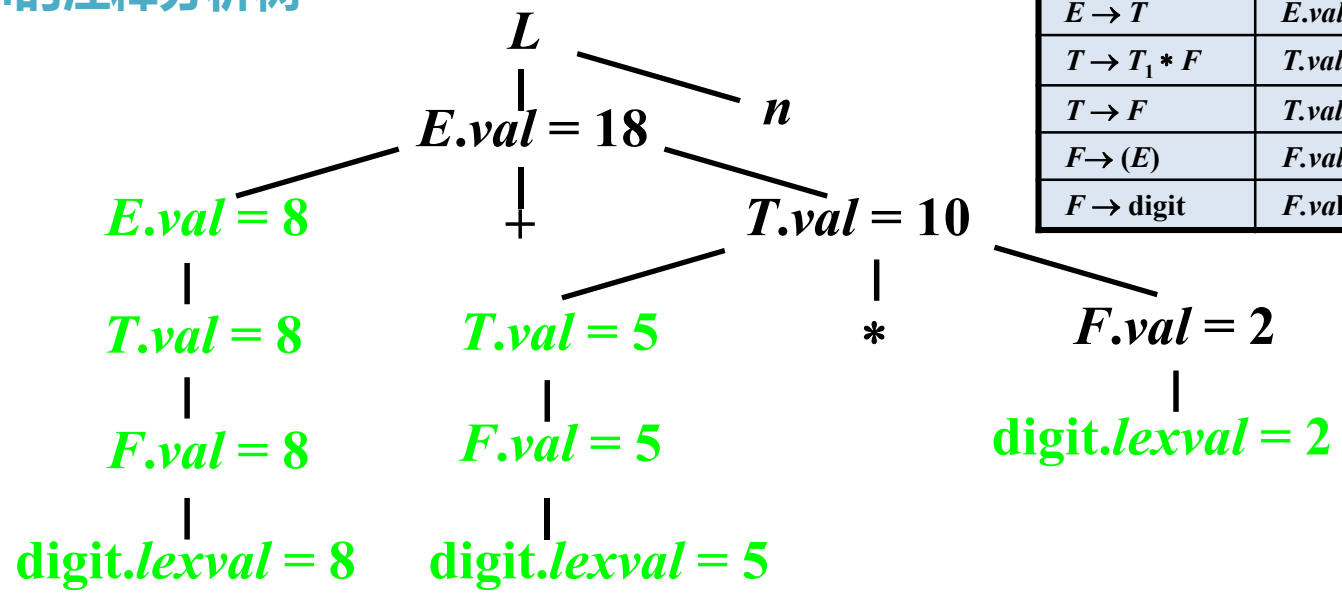



| 产 生 式 | 语 义 规 则 |
|---|---|
| $L \rightarrow E$ n | *print* (*E.val*) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | *E.val* = *T.val* |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | *T.val* = *F.val* |
| $F \rightarrow (E)$ | *F.val* = *E.val* |
| $F \rightarrow$ digit | *F.val* = digit.*lexval* |

# 语法分析树上的SDD求值

## 分析树各结点属性的计算可以自下而上地完成

8+5*2 n的注释分析树

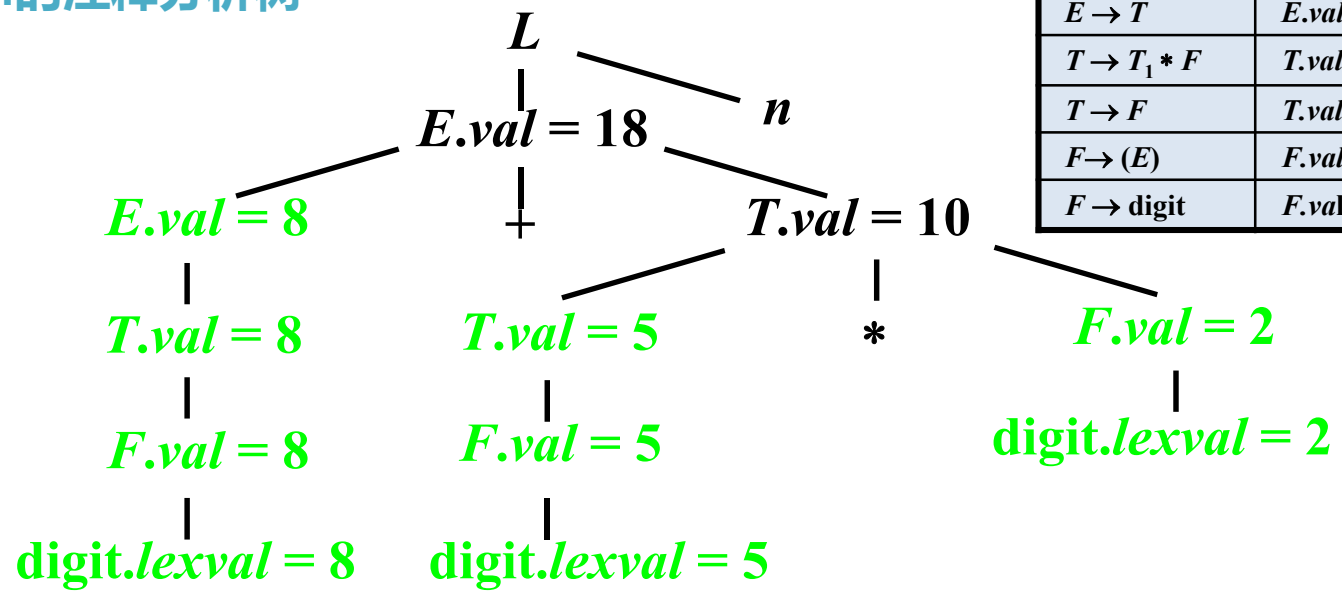



| 产 生 式 | 语 义 规 则 |
|---|---|
| $L \rightarrow E$ n | *print* (*E.val*) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | *E.val* = *T.val* |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | *T.val* = *F.val* |
| $F \rightarrow (E)$ | *F.val* = *E.val* |
| $F \rightarrow$ digit | *F.val* = digit.*lexval* |

# 语法分析树上的SDD求值

## 分析树各结点属性的计算可以自下而上地完成

8+5*2 n的注释分析树

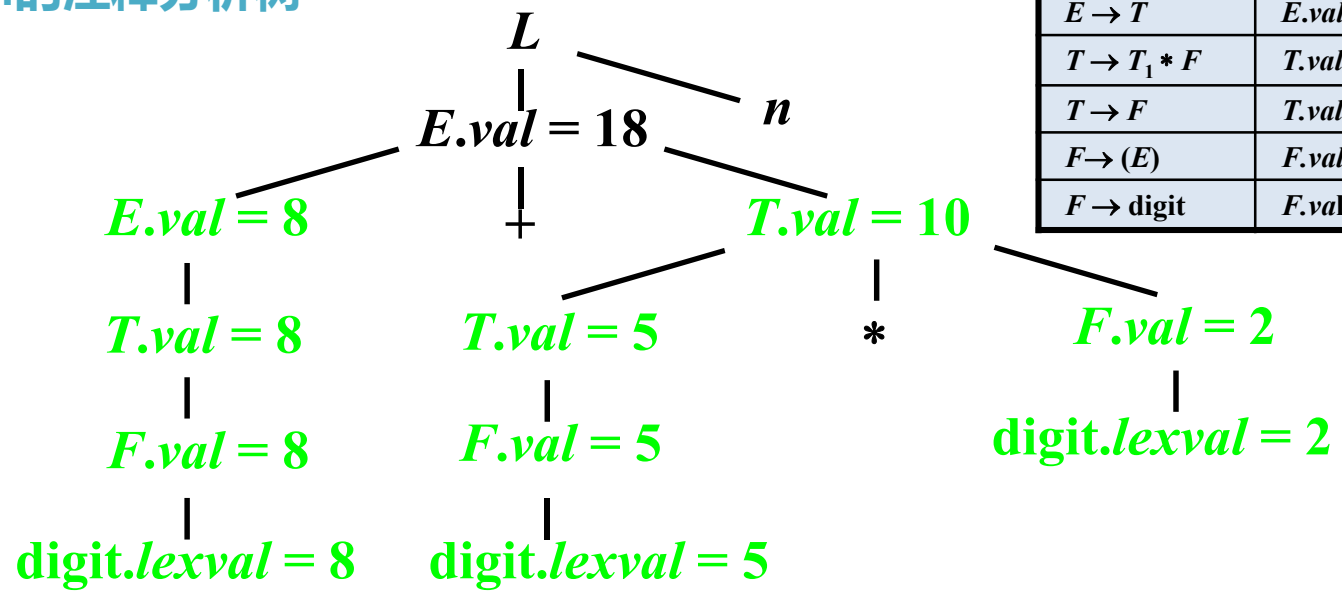



| 产 生 式 | 语 义 规 则 |
|---|---|
| $L \rightarrow E$ n | *print* (*E.val*) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | *E.val* = *T.val* |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | *T.val* = *F.val* |
| $F \rightarrow (E)$ | *F.val* = *E.val* |
| $F \rightarrow$ digit | *F.val* = digit.*lexval* |

# 语法分析树上的SDD求值

## 分析树各结点属性的计算可以自下而上地完成

8+5*2 n的注释分析树

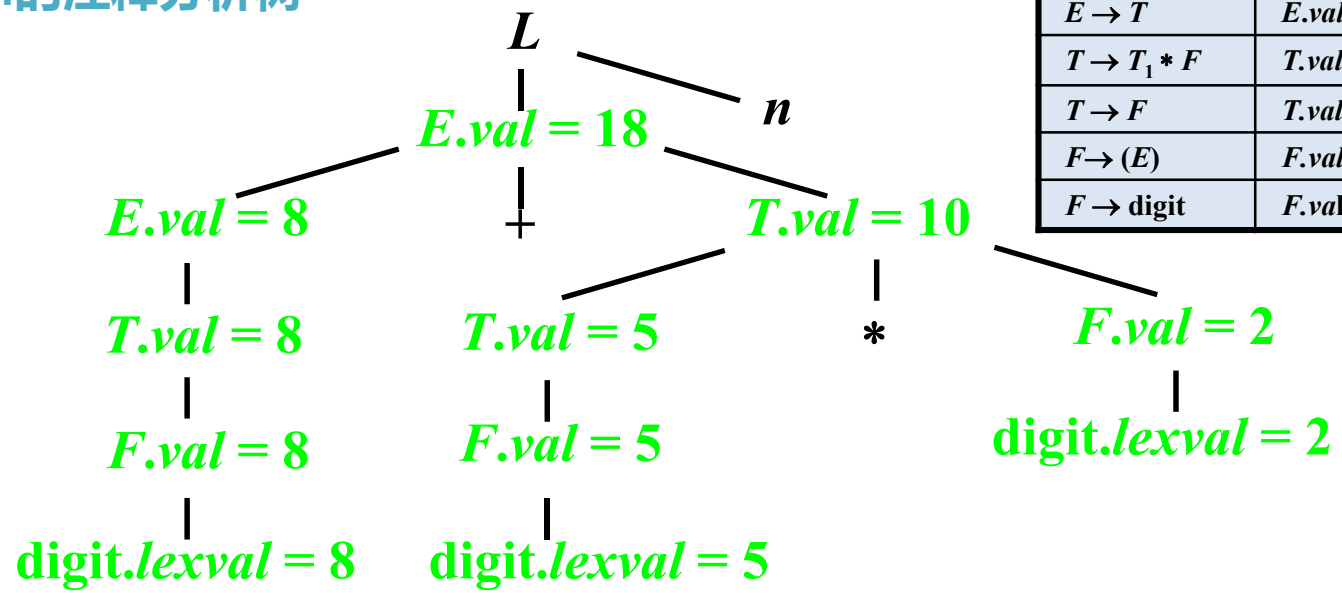



| 产 生 式 | 语 义 规 则 |
|---|---|
| $L \rightarrow E$ n | *print* (*E.val*) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | *E.val* = *T.val* |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | *T.val* = *F.val* |
| $F \rightarrow (E)$ | *F.val* = *E.val* |
| $F \rightarrow$ digit | *F.val* = digit.*lexval* |

# 语法分析树上的SDD求值

## 分析树各结点属性的计算可以自下而上地完成

8+5*2 n的注释分析树




| 产 生 式 | 语 义 规 则 |
|---|---|
| $L \rightarrow E$ n | *print* (*E.val*) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | *E.val* = *T.val* |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | *T.val* = *F.val* |
| $F \rightarrow (E)$ | *F.val* = *E.val* |
| $F \rightarrow$ digit | *F.val* = digit.*lexval* |

# 语法分析树上的SDD求值

- **按照分析树中的分支对应的文法产生式，应用相应的语义规则计算属性值**
- **计算顺序问题：**
  - 如果某个结点N的属性a为f($N_1.b_1$,$N_2.b_2$,...,$N_k.b_k$)，那么我们需要先算出$N_1.b_1$,$N_2.b_2$,...,$N_k.b_k$的值。
- **如果我们可以给各个属性值排出计算顺序，那么这个注释分析树就可以计算得到**
  - S属性的SDD一定可以按照自底向上的方式求值
- **下面的SDD不能计算**
  - $A \rightarrow B$　　A.s=B.i；　　B.i=A.s+1；

# 回顾一下两种属性的定义

- **综合属性**
  - 在分析树结点N上的非终结符号A的属性值由N对应的产生式所关联的语义规则来定义，通过N的子结点或N本身的属性值来定义
- **继承属性**
  - 结点N的属性值由N的父结点所关联的语义规则来定义，依赖于N的父结点、N本身和N的兄弟结点上的属性值

不允许N的继承属性通过N的子结点上的属性来定义，但是允许N的综合属性依赖于N本身的继承属性，终结符号有综合属性(由词法分析获得)，但是没有继承属性

# SDD (含有继承属性的例子1)

- 左递归文法使用自顶向下分析时，要消除左递归

| 产生式 | |
|---|---|
| 1) | $L \to E\ \mathbf{n}$ |
| 2) | $E \to E_1 + T$ |
| 3) | $E \to T$ |
| 4) | $T \to T_1 * F$ |
| 5) | $T \to F$ |
| 6) | $F \to (\ E\ )$ |
| 7) | $F \to \mathbf{digit}$ |

| PRODUCTION | |
|---|---|
| 1) | $T \to F\ T'$ |
| 2) | $T' \to * F\ T_1'$ |
| 3) | $T' \to \epsilon$ |
| 4) | $F \to \mathbf{digit}$ |



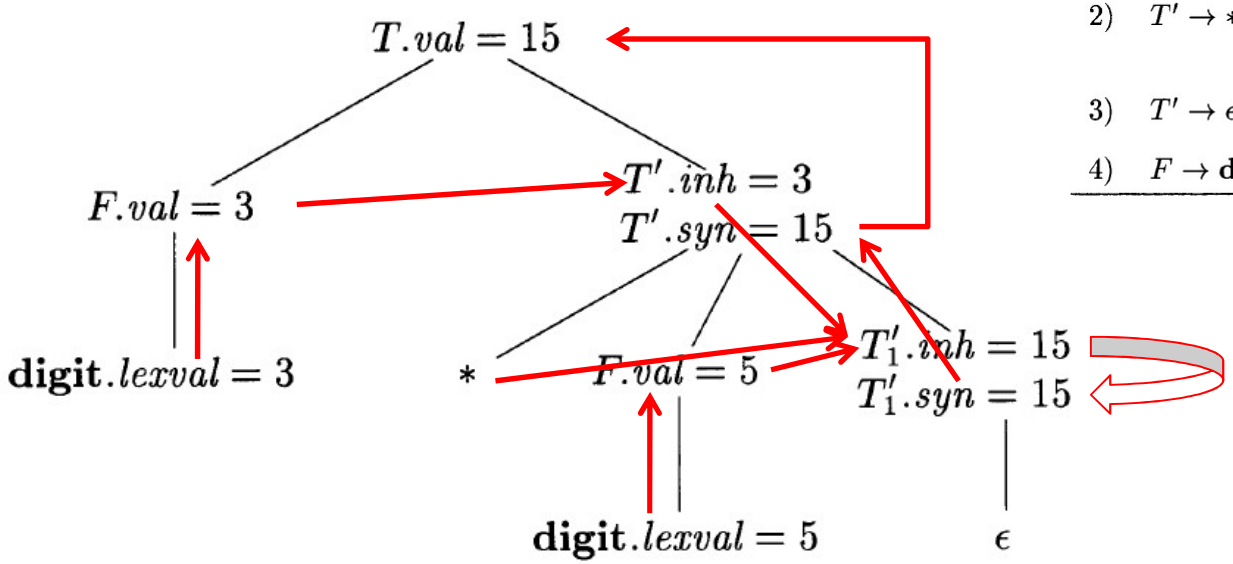
| 产生式 | | 语义规则 |
|---|---|---|
| 1) | $T \to F\ T'$ | $T'.inh = F.val$<br>$T.val = T'.syn$ |
| 2) | $T' \to * F\ T_1'$ | $T_1'.inh = T'.inh \times F.val$<br>$T'.syn = T_1'.syn$ |
| 3) | $T' \to \epsilon$ | $T'.syn = T'.inh$ |
| 4) | $F \to \mathbf{digit}$ | $F.val = \mathbf{digit}.lexval$ |

出现了继承属性！

# SDD (含有继承属性的例子1)

- **3*5注释分析树及属性依赖图**

  注意观察inh属性是如何传递的




| | 产生式 | 语义规则 |
|---|---|---|
| 1) | $T \to F\,T'$ | $T'.inh = F.val$<br>$T.val = T'.syn$ |
| 2) | $T' \to *\,F\,T'_1$ | $T'_1.inh = T'.inh \times F.val$<br>$T'.syn = T'_1.syn$ |
| 3) | $T' \to \epsilon$ | $T'.syn = T'.inh$ |
| 4) | $F \to \mathbf{digit}$ | $F.val = \mathbf{digit}.lexval$ |

# SDD (含有继承属性的例子2)

## SDD

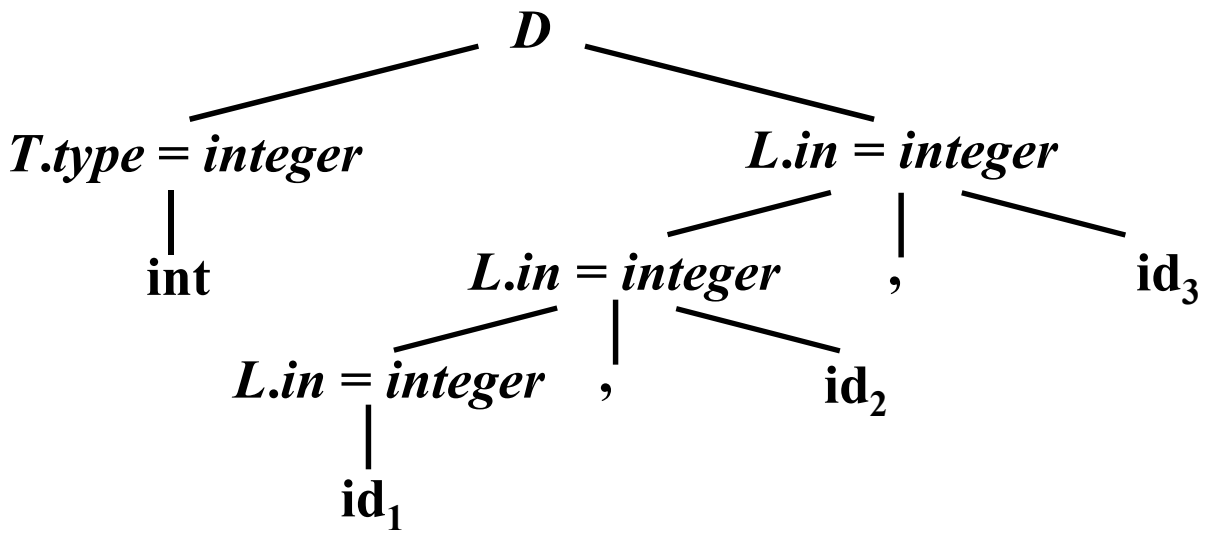
int $id_1$, $id_2$, $id_3$

| 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 | $L_1.in = L.in$;<br>$addType(\text{id}.entry, L_1.in)$ |
| $L \rightarrow$ id | $addType(\text{id}.entry, L.in)$ |

对于一个属性，不允许出现即是综合又是继承的现象

# SDD (含有继承属性的例子2)

- int $id_1$, $id_2$, $id_3$ 注释分析树
  - 不可能像像综合属性那样自下而上标注属性

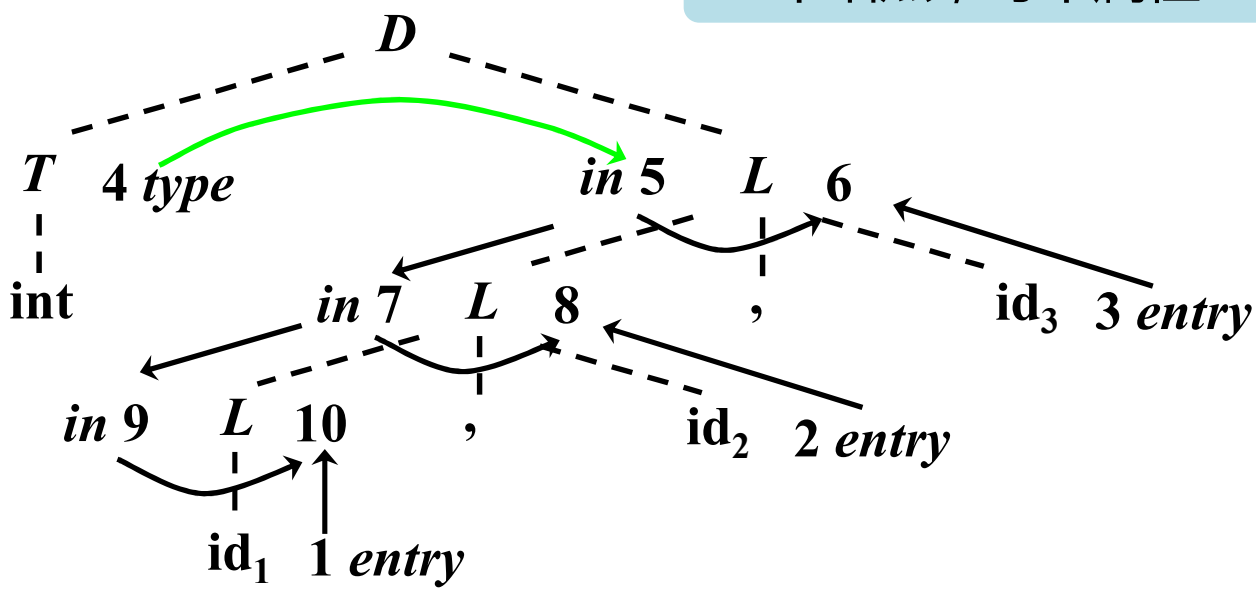


# SDD (含有继承属性的例子2)

- int $id_1$, $id_2$, $id_3$的分析树（虚线）的依赖图（实线）

  $D \rightarrow TL$　　$L.in = T.type$

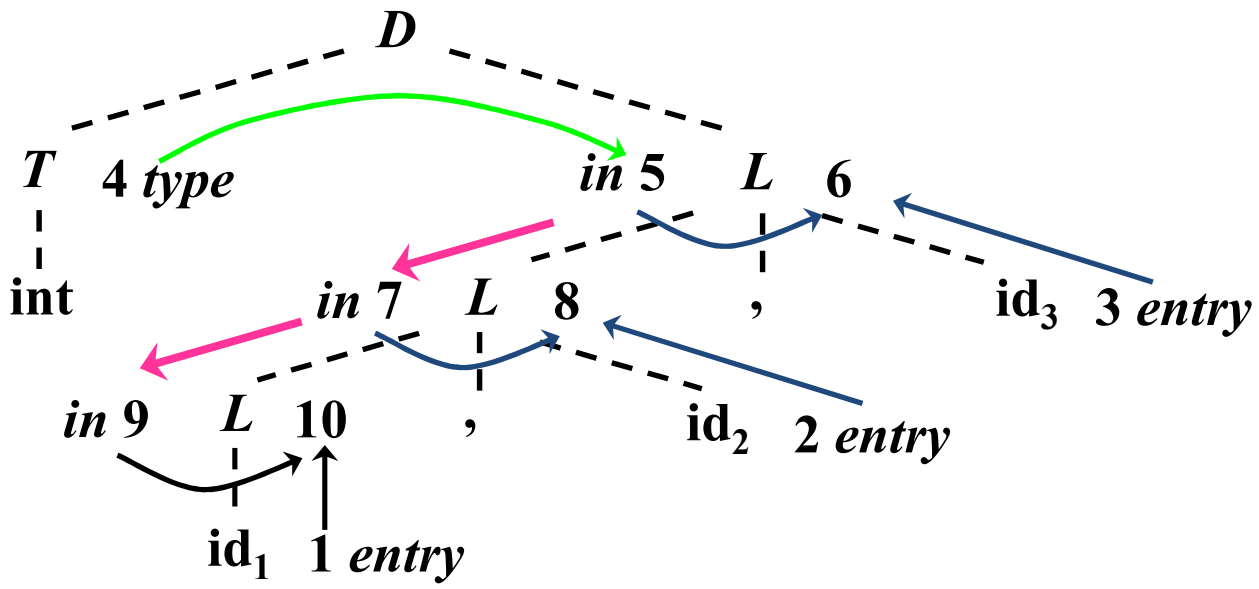
10个结点，每个属性一个结点



## SDD (含有继承属性的例子2)

- int $id_1$, $id_2$, $id_3$的分析树（虚线）的依赖图（实线）

  $L \rightarrow L_1$, id  $L_1.in = L.in$; *addType* (id.*entry*, *L.in*)



# 拓扑顺序

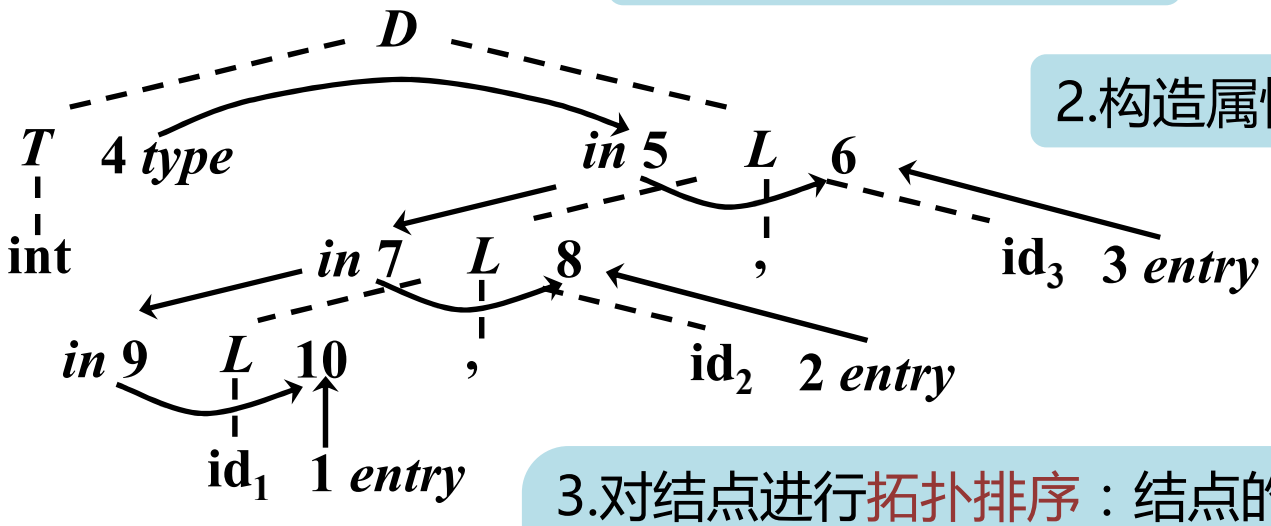
- **使用依赖图来表示计算顺序 ---拓扑顺序**
  - 显然，这些值的计算顺序应该形成一个偏序关系。如果依赖图中出现了环，表示属性值无法计算
- **给定一个SDD，很难判定是否存在一棵分析树，其对应的依赖图包含环**

# SDD (含有继承属性的例子2)

## Int id1,id2, id3计算顺序

1.先构造输入的分析树

2.构造属性依赖图

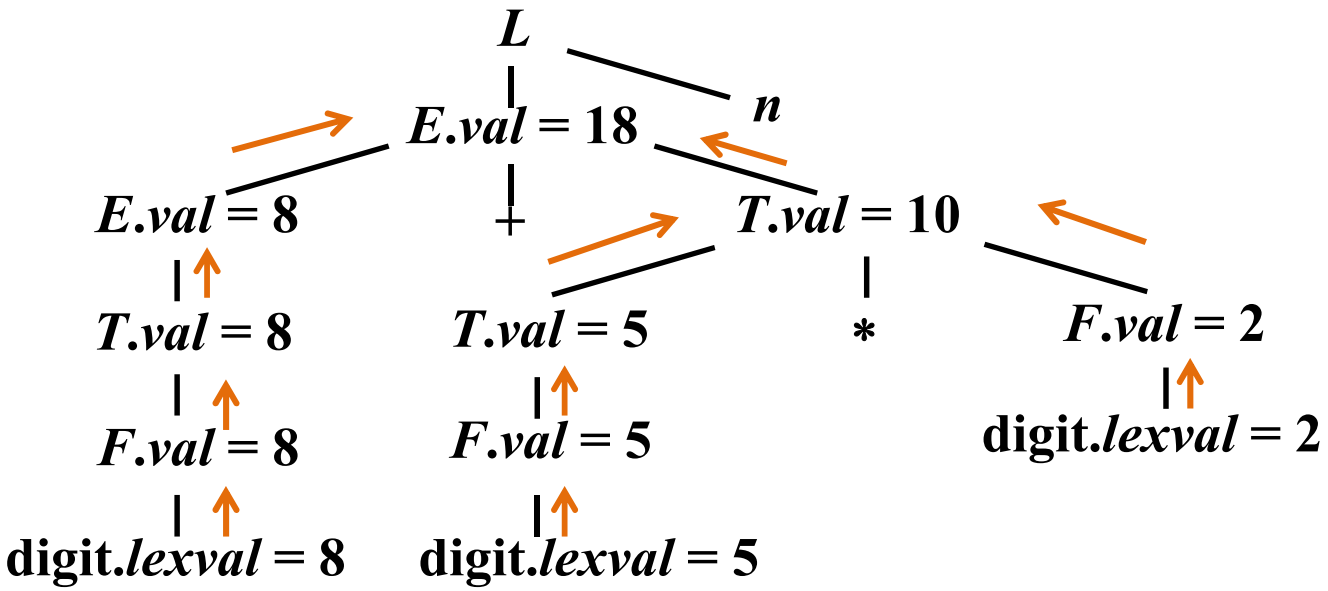



3.对结点进行拓扑排序：结点的一种排序，使得边只会从该次序中先出现的结点到后出现的结点，例：1，2，3，4，5，6，7，8，9，10 --- 不一定唯一

4.按拓扑排序的次序计算属性

# SDD (只含综合属性的例子)

## 8+5*2 n的计算顺序



# 语义规则的计算方法---理论做法

输入：任意的语句 s；输出：附注语法树 AT(s).

1. 对 s 进行语法分析并构建语法树 T(s);
2. 对 T(s) 构造依赖关系图 DG(s);
3. 判断 DG(s) 是否有回路，如果有，报错；否则下一步.
4. 对 DG(s) 进行拓扑排序.
5. 按照拓扑排序得到的线序依次对每个属性求值.
6. 输出附注语法树.

- 优点：可以求解所有的属性. 缺点：代价大，效率低.
- 最高效的方法是：属性求值的次序与语法分析对语法树遍历的次序相容，使得属性的求值能够与语法分析同步完成.
- 这需要在 SDD 时设计好属性的依赖关系，保证在任意的语法树上的求值次序与语法分析的遍历相容.

## 语义规则的计算方法---实际操作tips

- 基于规则的方法：（编译器实现者）静态确定（编译器设计者提供的）语义规则的计算次序---用于手工构造的方法
- 忽略规则的方法：（编译器实现者）事先确定属性的计算策略（如边分析边计算），（编译器设计者提供的）语义规则必须符合所选分析方法的限制---适用于自动生成的方法

## 实现边分析边翻译

- **怎样的SDD可以和以前所介绍的语法分析结合在一起？**
  - 属性的计算次序一定受分析方法所限定的分析树结点建立次序的限制
    - 在对SDD的求值过程中，如果结点N的属性a依赖于结点$M_1$的属性$a_1$，$M_2$的属性$a_2$，…。那么我们必须先计算出$M_i$的属性，才能计算N的属性a
  - 分析树的结点是自左向右生成
  - 如果属性信息是自左向右流动，那么就有可能在分析的同时完成属性计算

# 实现边分析边翻译

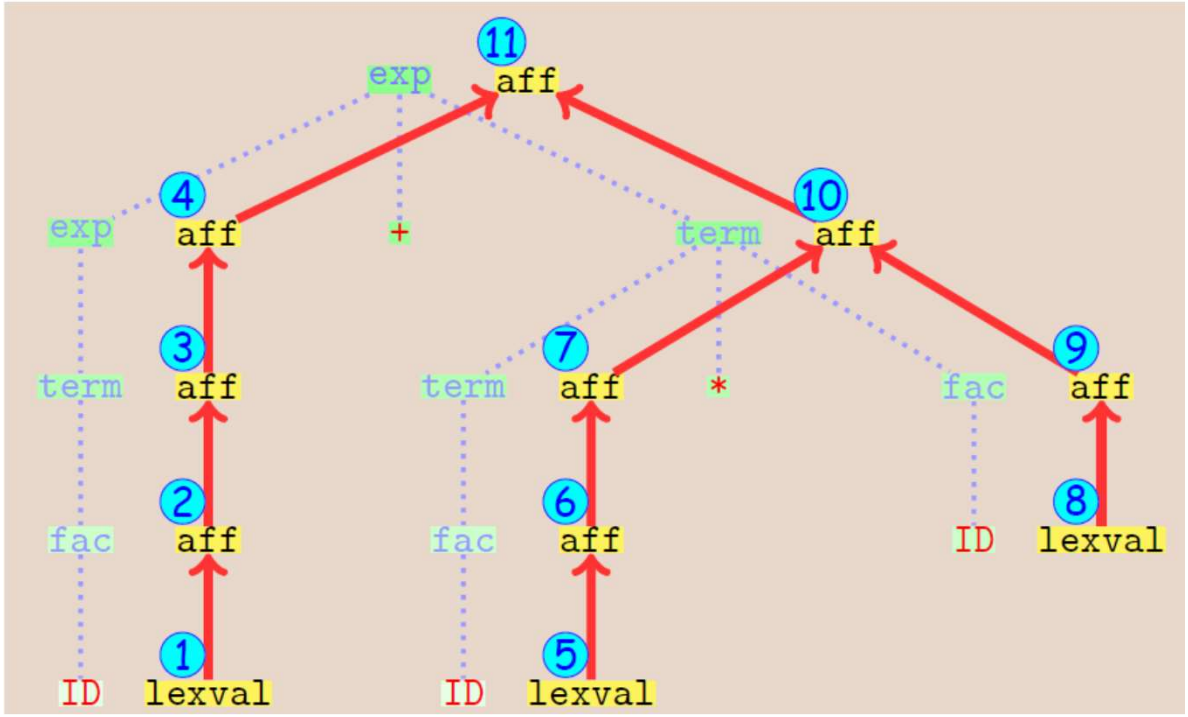
- **特定类型的SDD一定不包含环，且有固定排序模式**
  - S属性的SDD
  - L属性的SDD
- **对于这些类型的SDD，我们可以确定属性的计算顺序，且可以把不需要的属性（及分析树结点）抛弃以提高效率**
- **这两类SDD可以很好地和我们已经研究过的语法分析相结合**

# S属性的SDD及其相容分析方法

- **只包含综合属性的SDD称为S属性的SDD**
  - 每个语义规则都根据产生式体中的属性值来计算头部非终结符号的属性值
  - 在依赖图中，总是通过子结点的属性值来计算父结点的属性值。可以和自顶向下、自底向上的语法分析过程一起计算
- **自底向上（和LR语法分析器一起实现）**
  - 在构造分析树的结点的同时计算相关的属性（此时其子结点的属性必然已经计算完毕）；栈中的状态可以附加相应的属性值；在进行归约时，按照语义规则计算归约得到的符号的属性值
- **自顶向下**
  - 递归下降分析中，可以在过程A()的最后计算A的属性---此时A调用的其他过程（对应于子结构）已经调用完毕

# S属性的SDD及其相容分析方法

- **例如，和LR分析相容**



# 在分析树上计算SDD

**按照后序遍历的顺序计算属性值即可**

```
postorder(N)
{
    for(从左边开始，对N的每个子结点C)
        postorder(C);
        //递归调用返回时，各子结点的属性计算完毕
        对N的各个属性求值；
}
```

一个自底向上的语法分析过程对应于一次后序遍历：后续遍历精确地对应于一个LR分析器将一个产生式规约为它的开始符号的过程。

在LR分析过程中，我们实际上不需要构造分析树的结点

# L属性的SDD及其相容分析方法

- **每个属性**
  - 要么是综合属性(S属性定义属于L属性定义)
  - 要么是继承属性，且产生式$A \rightarrow X_1X_2...X_n$中计算$X_i.a$的规则只能使用：
    - A的继承属性
    - $X_i$左边的文法符号$X_j$的继承属性或综合属性
    - $X_i$自身的继承或综合属性，且这些属性之间的依赖关系不形成环
- **依赖图的边**
  - 继承属性从左到右，从上到下
  - 综合属性从下到上
- **在扫描过程中计算一个属性值时，和它相关的依赖属性都已经计算完毕**

# L属性的SDD及其相容分析方法

- **例如，教材例5.8所示SDD是L属性的**

| 产生式 | 语义规则 |
| --- | --- |
| $T \rightarrow F\,T'$ | $T'.inh = F.val$ |
| $T' \rightarrow *\,F\,T_1'$ | $T_1'.inh = T'.inh \times F.val$ |

- **再如，教材例5.9，任何包含下列产生式和规则的SDD都不是L属性的**

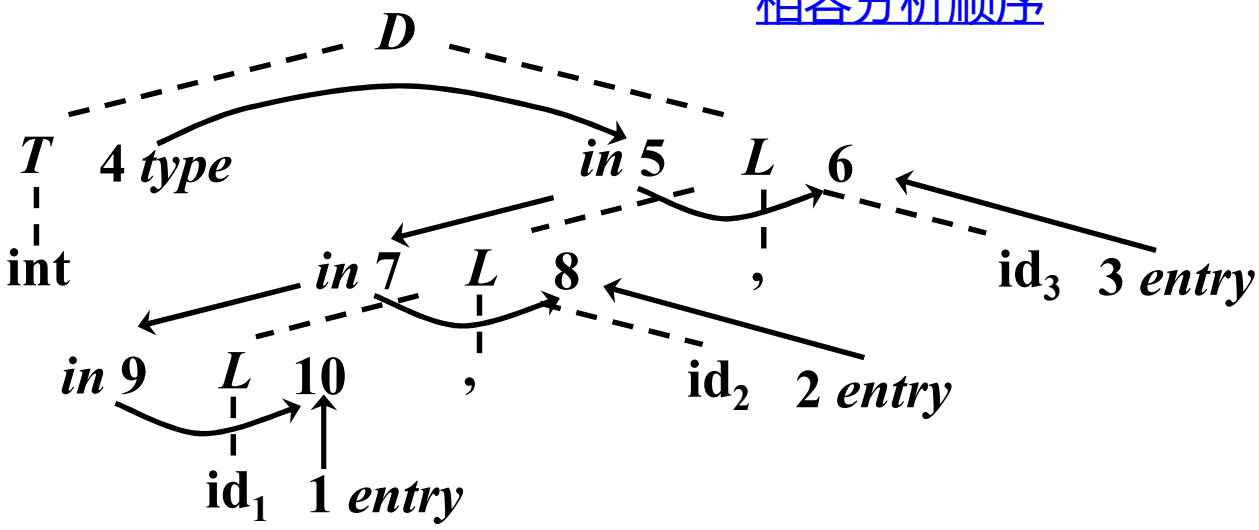
| 产生式 | 语义规则 |
| --- | --- |
| $A \rightarrow B\,C$ | $A.s = B.b;$<br>$B.i = f(C.c, A.s)$ |

# L属性的SDD及其相容分析方法

## 例如：变量类型声明的语法制导定义是一个*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 | $L_1.in = L.in$; $addType(\text{id}.entry, L.in)$ |
| $L \rightarrow$ id | $addType(\text{id}.entry, L.in)$ |

相容分析顺序



# 具有受控副作用的语义规则

- **一个没有副作用的SDD称为属性文法，一个属性文法的规则仅通过其他属性值和常量值来顶一个另一个属性值**
  - 增加了描述的复杂度
    - 比如语法分析时如果没有副作用，标识符表就必须作为属性传递
    - 可以把标识符表作为全局变量，然后通过副作用函数来添加新标识符
- **受控的副作用**
  - 不会对属性求值产生约束，即可以按照任何拓扑顺序求值，不会影响最终结果
  - 或者对求值过程添加简单的约束

# 受控副作用的例子

- **L → E n　　print(E.val)**
  - 通过副作用打印出E的值
  - 总是在最后执行，而且不会影响其它属性的求值
- **变量声明的SDD中的副作用**
  - addType将标识符的类型信息加入到标识符表中
  - 只要标识符不被重复声明，标识符的类型信息总是正确的

| | 产生式 | 语义规则 |
|---|---|---|
| 1) | $D \rightarrow T\ L$ | $L.inh = T.type$ |
| 2) | $T \rightarrow \mathbf{int}$ | $T.type = \text{integer}$ |
| 3) | $T \rightarrow \mathbf{float}$ | $T.type = \text{float}$ |
| 4) | $L \rightarrow L_1\ ,\ \mathbf{id}$ | $L_1.inh = L.inh$<br>$addType(\mathbf{id}.entry, L.inh)$ |
| 5) | $L \rightarrow \mathbf{id}$ | $addType(\mathbf{id}.entry, L.inh)$ |

# 9.3 语法制导翻译的应用

## 语法制导翻译的应用

- **上面例子中已经介绍了一些语法制导翻译技术 (SDD) 的应用**
  - 计算值
  - 变量类型声明 (在符号表里填写内容)
  - 还能做什么？

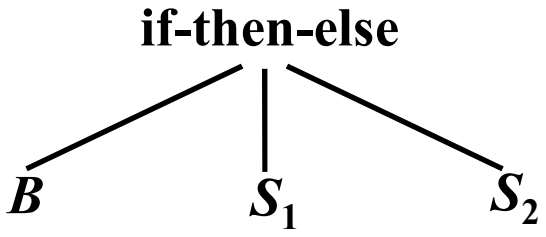
## 语法制导翻译的应用

- **语法制导翻译技术主要应用**
  - 中间代码生成
    - 抽象语法树---一种中间代码表示形式
    - 三地址代码
    - 后缀、前缀表达式
  - 类型检查
    - 处理基本类型和数组类型的L属性定义

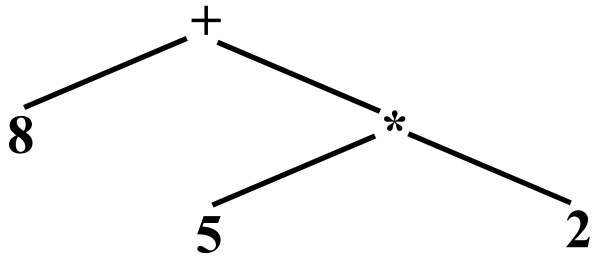
## 构造AST

- **抽象语法树AST**
  - 语法树是分析树的浓缩表示：算符和关键字是作为内部结点
  - 语法制导翻译可以基于分析树，也可以基于语法树
  - 语法树的例子：

if $B$ then $S_1$ else $S_2$




8 + 5 * 2



# 构造AST

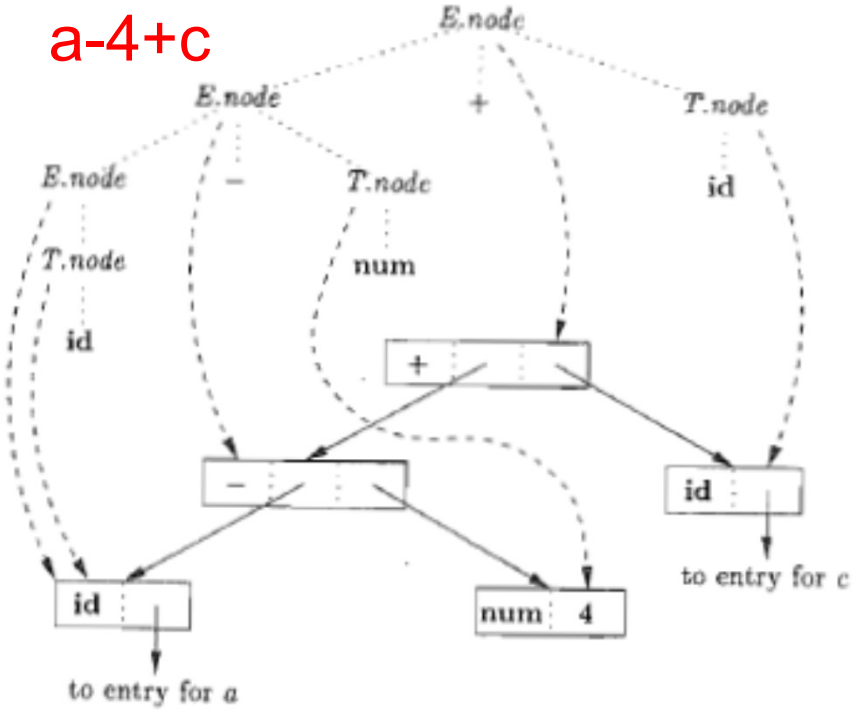
- **抽象语法树**
  - 每个结点代表一个语法结构；对应于一个运算符
  - 结点的每个子结点代表其子结构；对应于运算分量
  - 表示这些子结构按照特定方式组成了较大的结构
  - 可以忽略掉一些标点符号等非本质的东西
- **语法树的表示方法**
  - 每个结点用一个对象表示，对象有多个域
    - Leaf(op, val)创造一个叶子对象，返回一个指向叶子结点对应新记录的指针
    - Node(op, c1, c2, …, ck)，其中c1-ck为子结点

# 构造AST (S属性SDD)

## 例1：S属性SDD，构造语法树的语法制导定义（p203, 5.11）

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

a-4+c

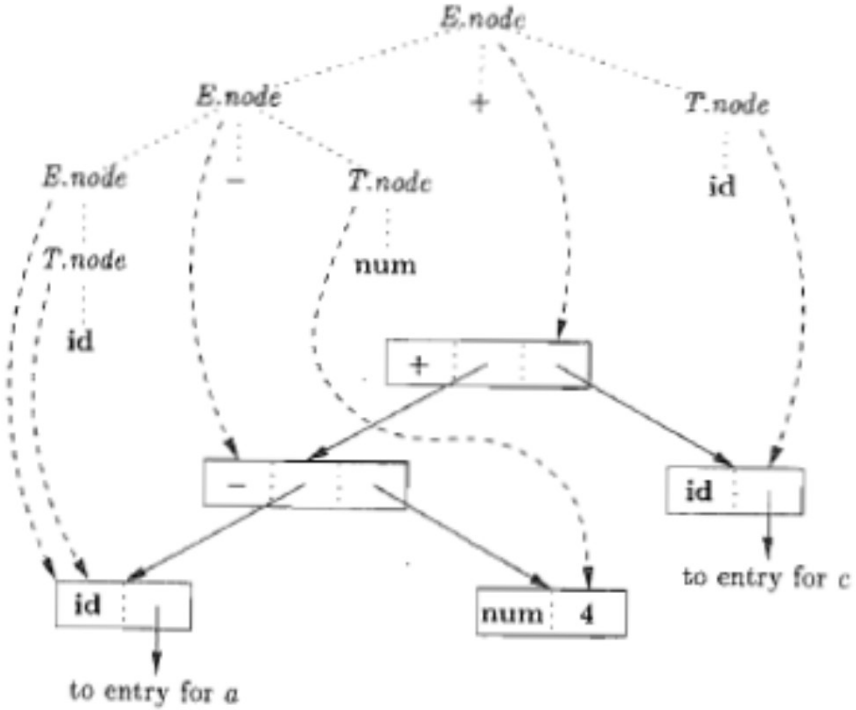


# 构造AST (S属性SDD)

## 例1：S属性SDD，构造语法树的语法制导定义（p203, 5.11）

a-4+c, 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);



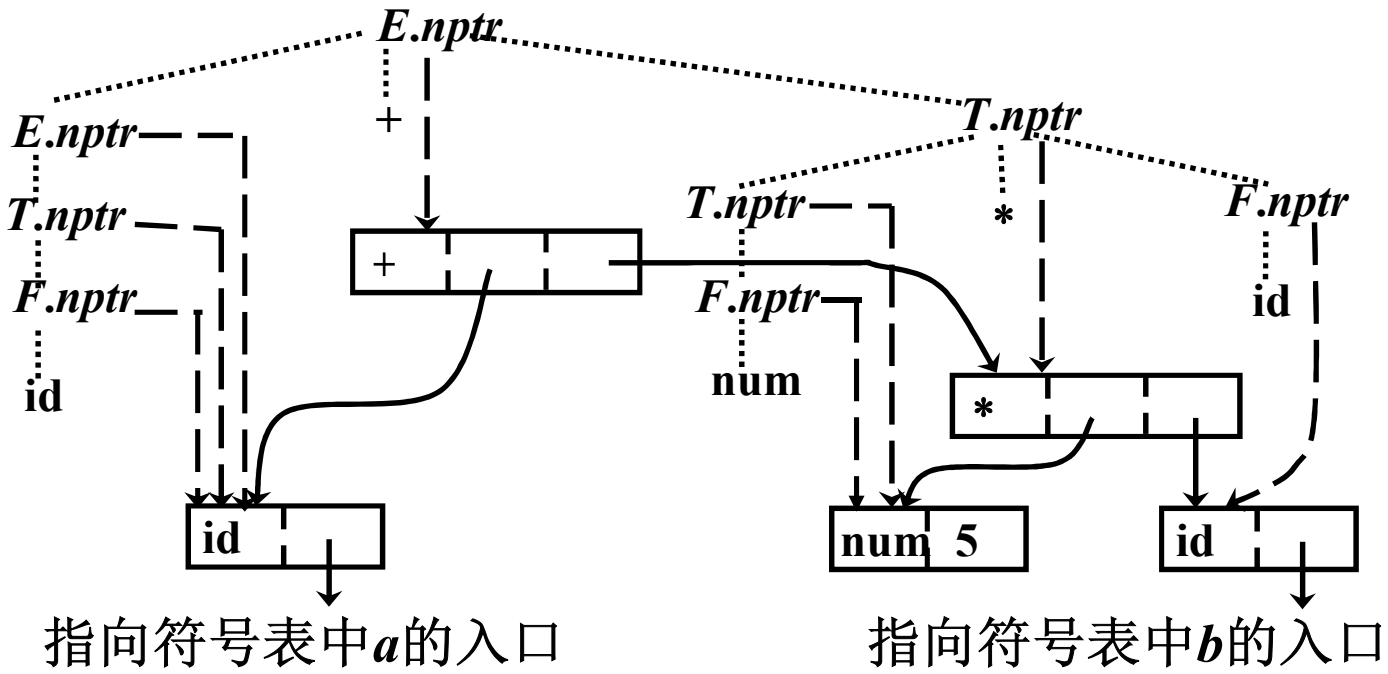
# 构造AST (S属性SDD)

## 例2：S属性SDD，构造语法树的语法制导定义

| 产　生　式 | 语 义 规 则 |
|---|---|
| $E \rightarrow E_1 + T$ | $E.nptr = mkNode(\text{‘+’}, E_1.nptr, T.nptr)$ |
| $E \rightarrow T$ | $E.nptr = T.nptr$ |
| $T \rightarrow T_1 * F$ | $T.nptr = mkNode(\text{‘*’}, T_1.nptr, F.nptr)$ |
| $T \rightarrow F$ | $T.nptr = F.nptr$ |
| $F \rightarrow (E)$ | $F.nptr = E.nptr$ |
| $F \rightarrow$ id | $F.nptr = mkLeaf$ (id, id.*entry*) |
| $F \rightarrow$ num | $F.nptr = mkLeaf$ (num, num.*val*) |

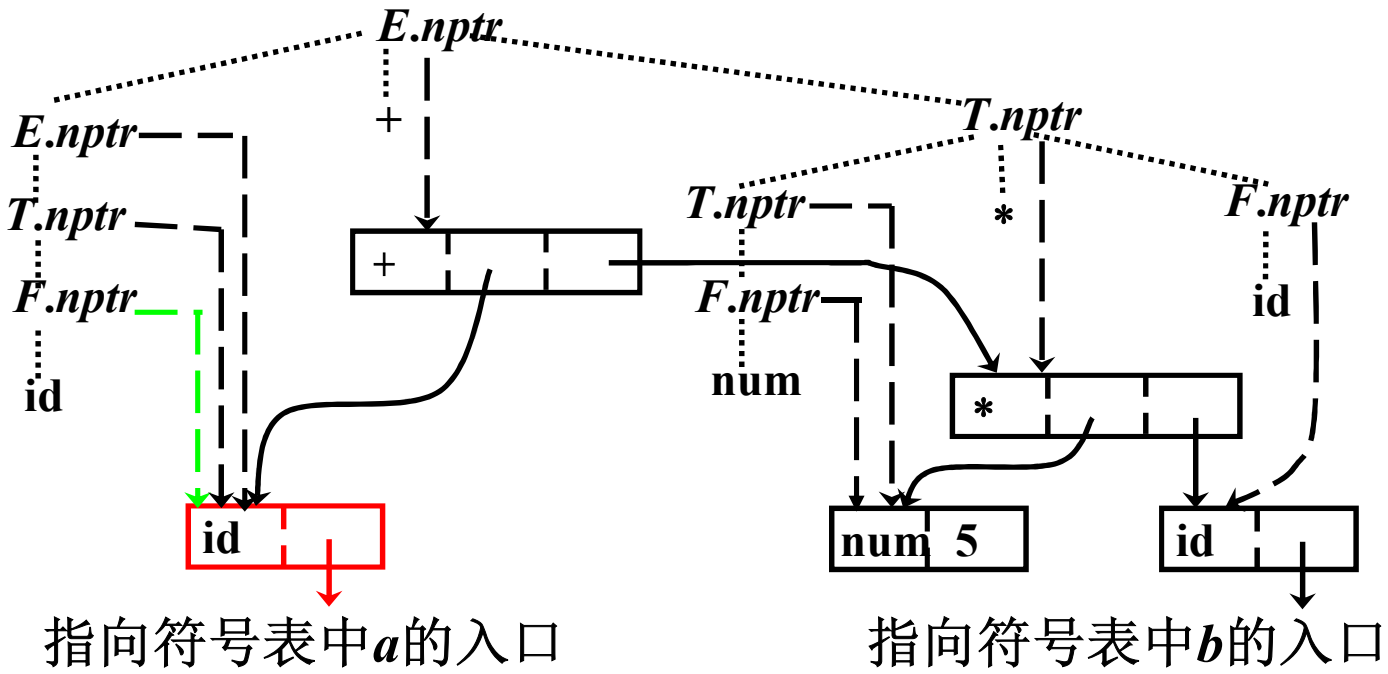
# 构造AST (S属性SDD)

## *a*+5∗*b*的语法树的构造



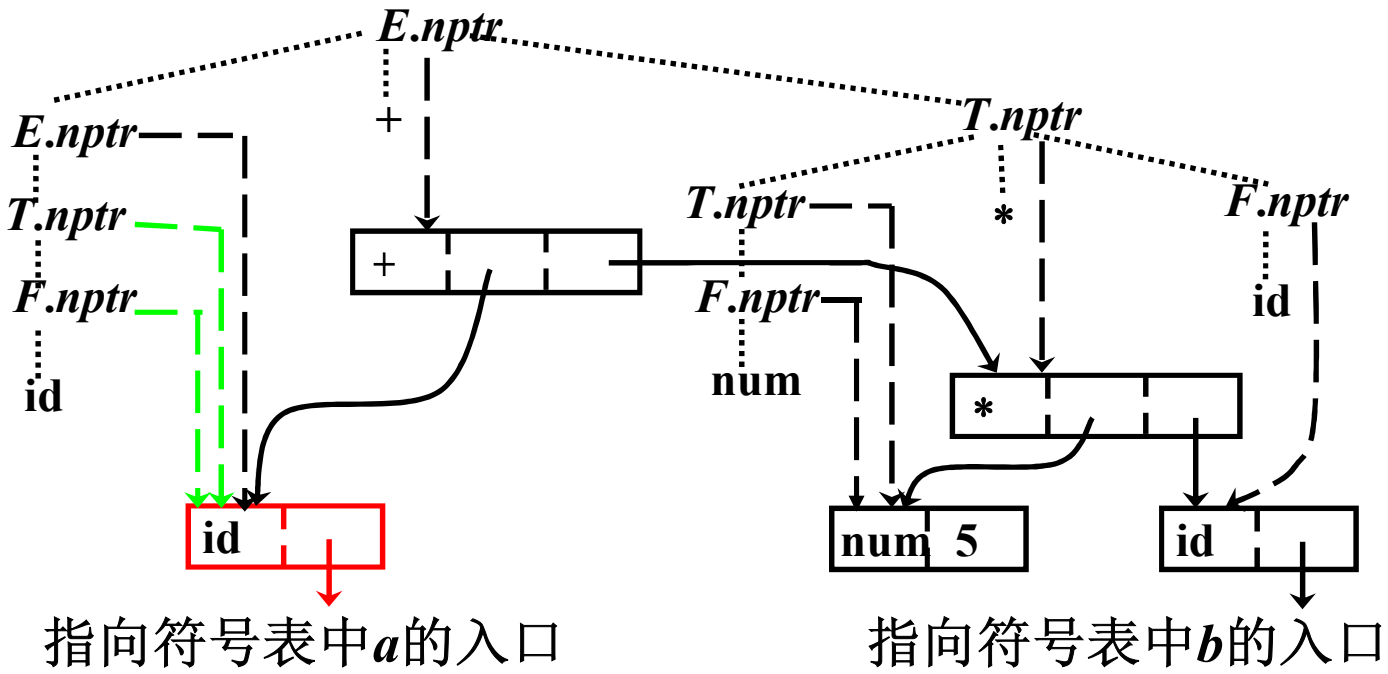
# 构造AST (S属性SDD)

## *a*+5∗*b*的语法树的构造



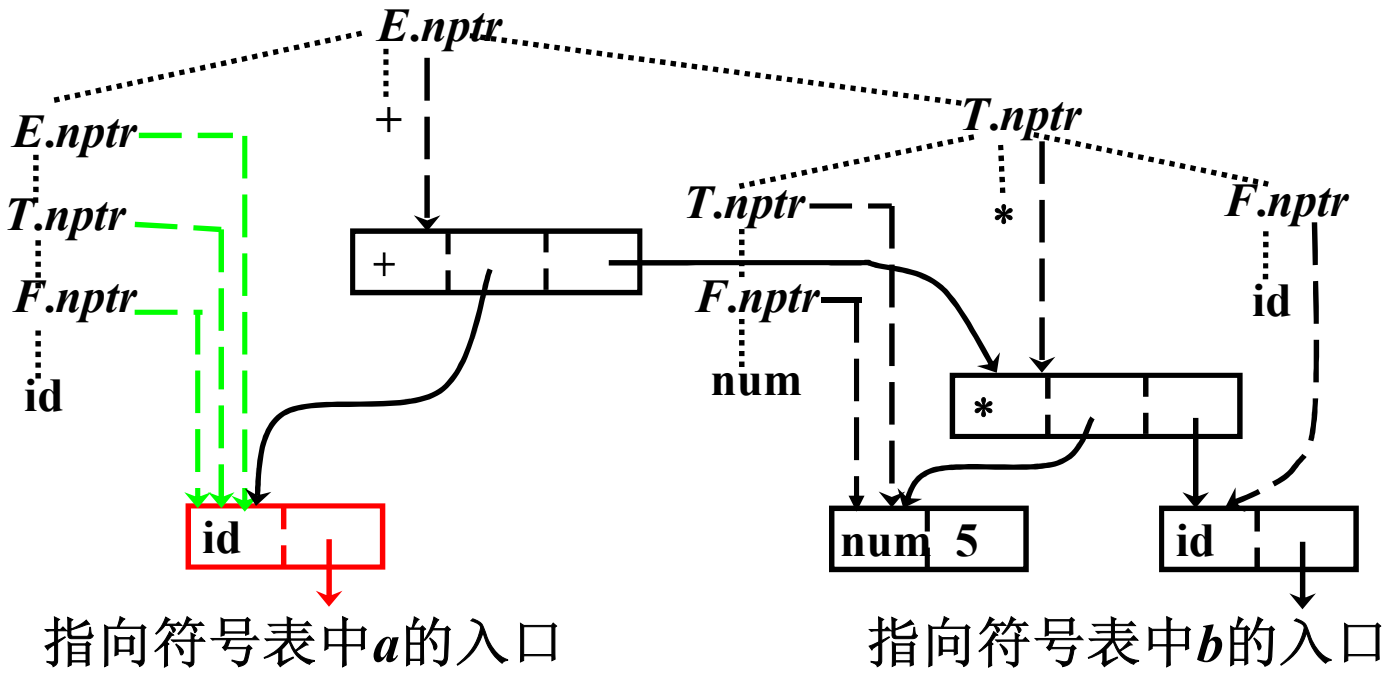
# 构造AST (S属性SDD)

## *a*+5∗*b*的语法树的构造



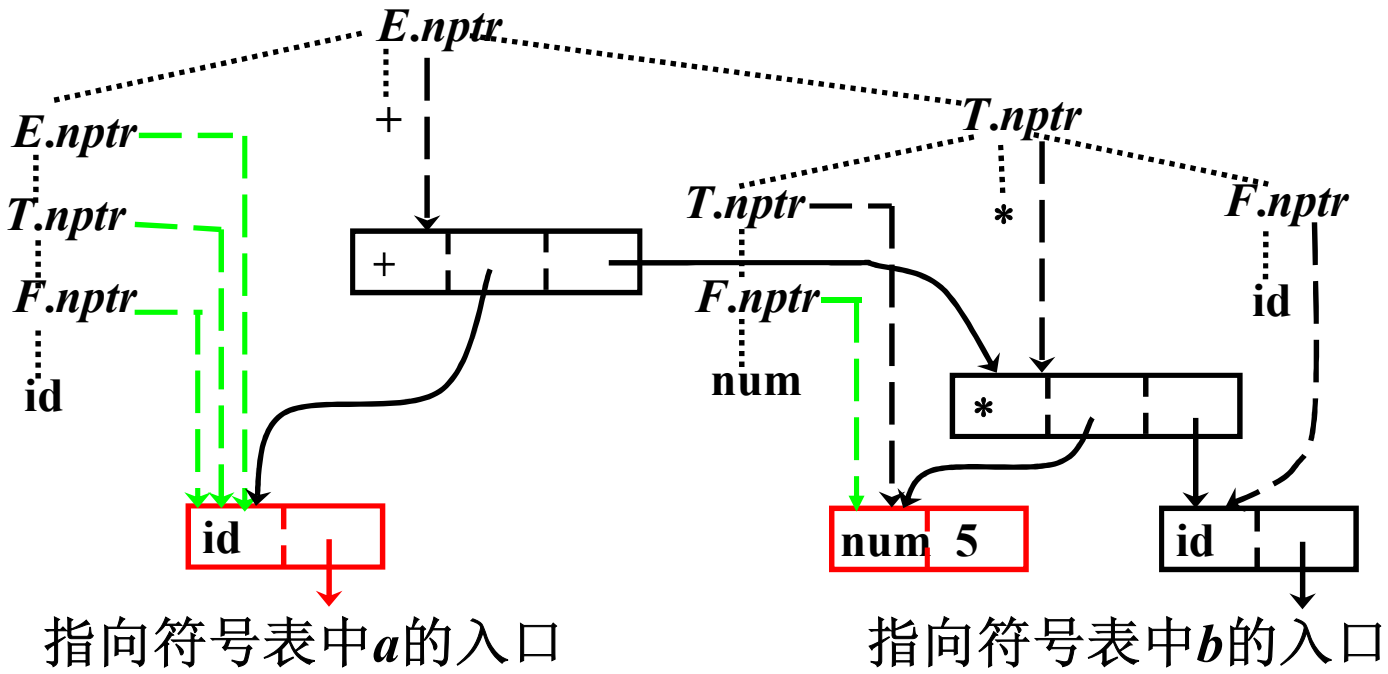
# 构造AST (S属性SDD)

## *a*+5∗*b*的语法树的构造



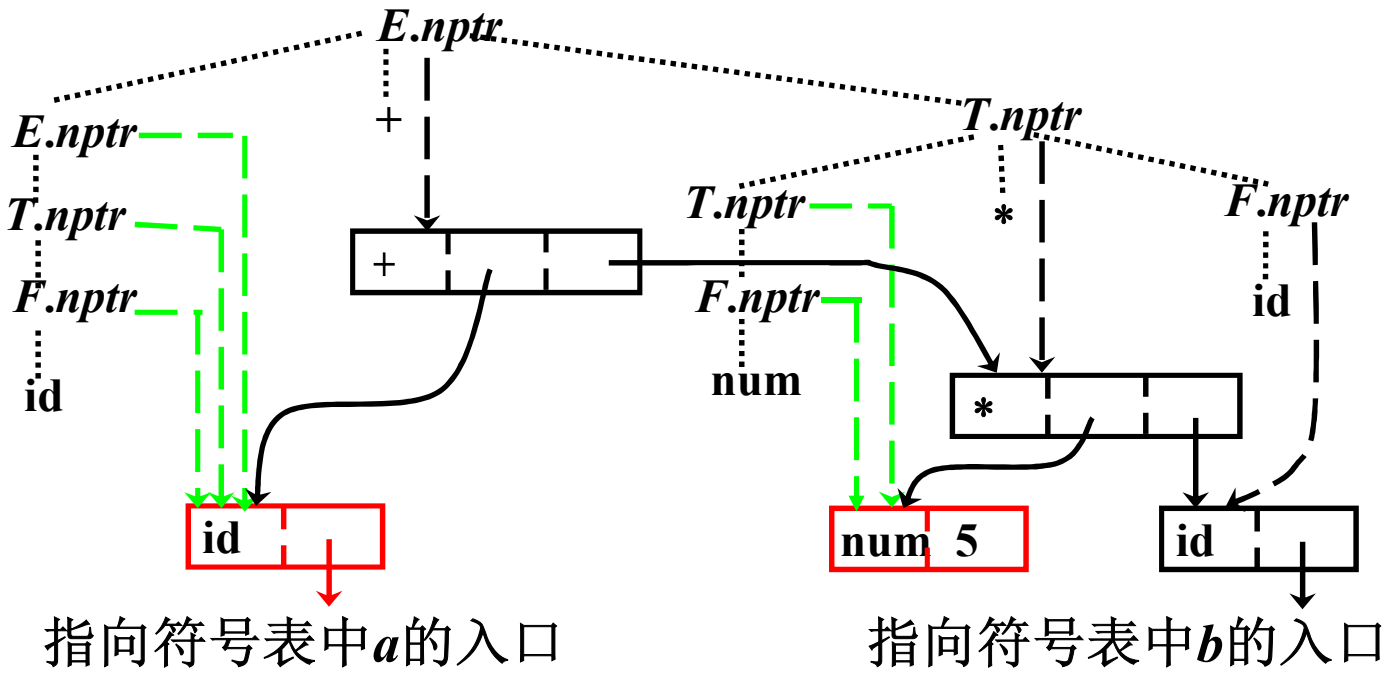
# 构造AST (S属性SDD)

## *a*+5∗*b*的语法树的构造



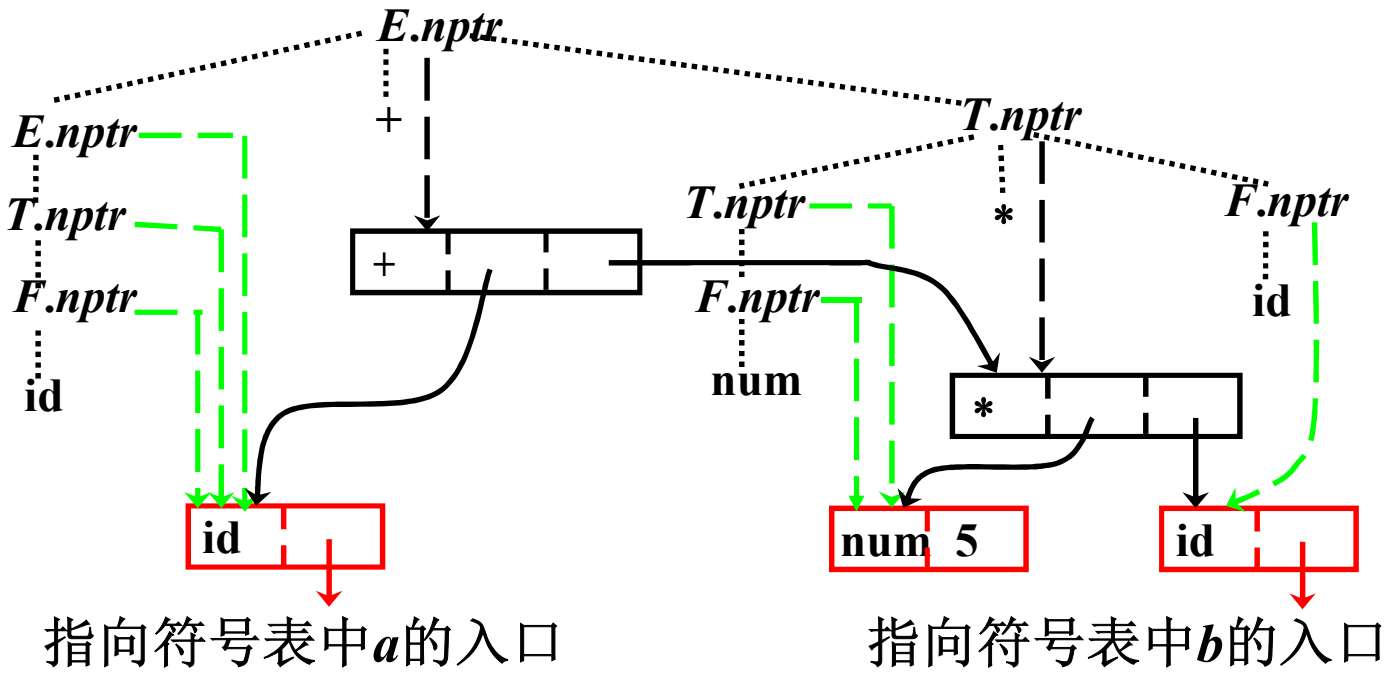
# 构造AST (S属性SDD)

## *a*+5∗*b*的语法树的构造



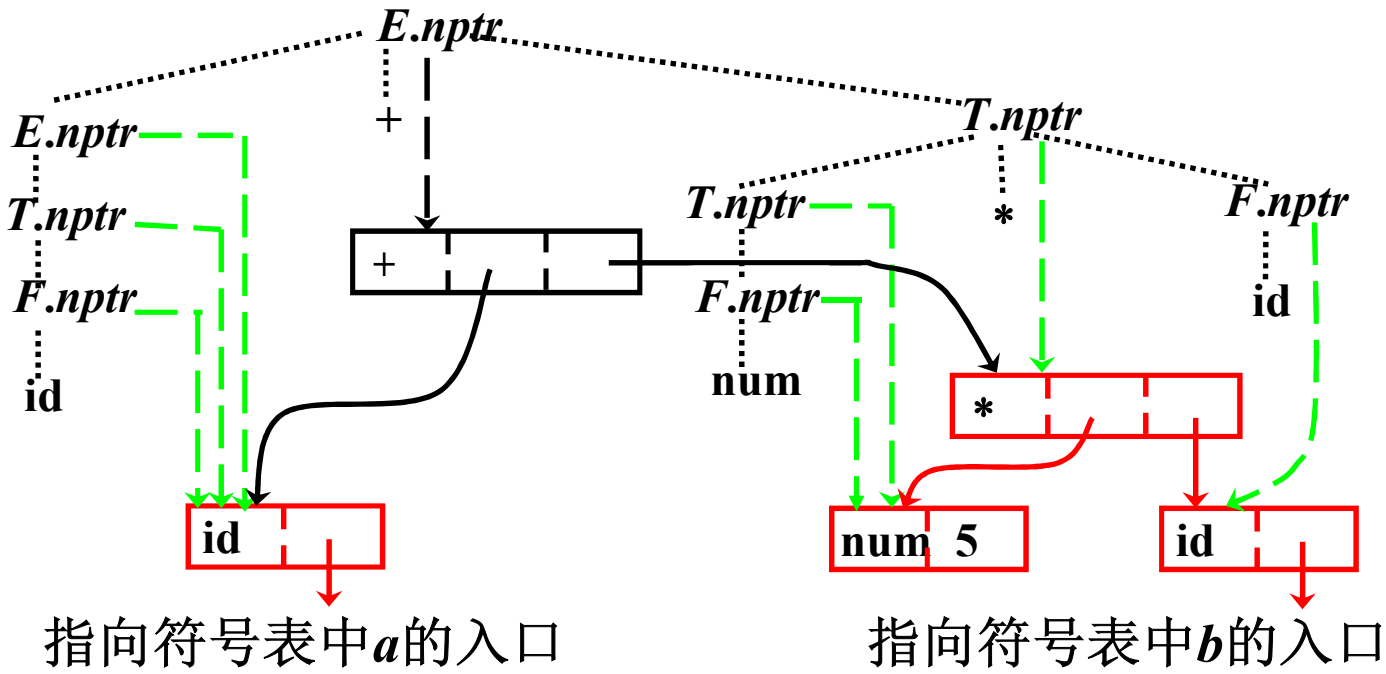
# 构造AST (S属性SDD)

## *a*+5∗*b*的语法树的构造



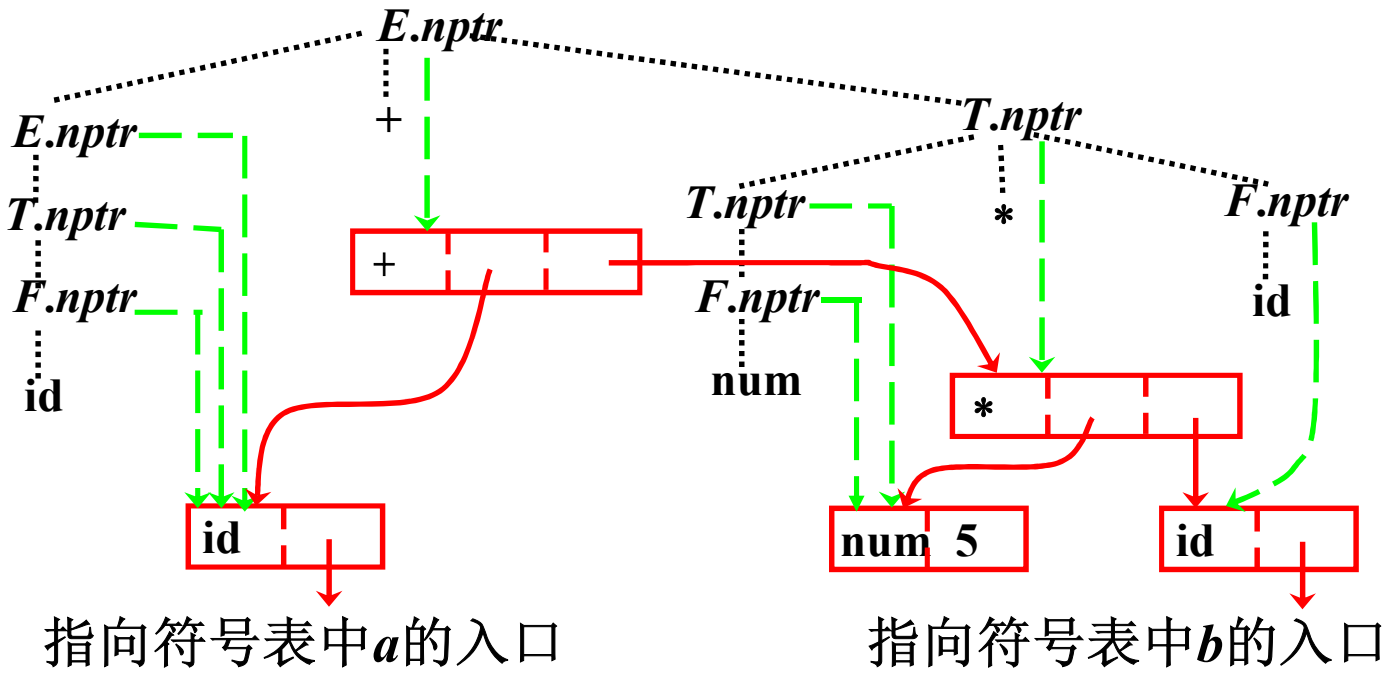
# 构造AST (S属性SDD)

## *a*+5∗*b*的语法树的构造



# 构造AST (S属性SDD)

## *a*+5∗*b*的语法树的构造



# 构造AST (L属性SDD)

## 例3：L属性SDD，构造语法树的语法制导定义（p205, 5.12）

| 产生式 | 语义规则 |
|---|---|
| $E \to E_1 + T$ | $E.node =$ **new** $Node('+', E_1.node, T.node)$ |
| $E \to E_1 - T$ | $E.node =$ **new** $Node('-', E_1.node, T.node)$ |
| $E \to T$ | $E.node = T.node$ |
| $T \to (\ E\ )$ | $T.node = E.node$ |
| $T \to$ **id** | $T.node =$ **new** $Leaf(\textbf{id}, \textbf{id}.entry)$ |
| $T \to$ **num** | $T.node =$ **new** $Leaf(\textbf{num}, \textbf{num}.val)$ |

消除左递归

| | 产生式 | 语义规则 |
|---|---|---|
| 1) | $E \to T\ E'$ | $E.node = E'.syn$<br>$E'.inh = T.node$ |
| 2) | $E' \to +\ T\ E'_1$ | $E'_1.inh =$ **new** $Node('+', E'.inh, T.node)$<br>$E'.syn = E'_1.syn$ |
| 3) | $E' \to -\ T\ E'_1$ | $E'_1.inh =$ **new** $Node('-', E'.inh, T.node)$<br>$E'.syn = E'_1.syn$ |
| 4) | $E' \to \epsilon$ | $E'.syn = E'.inh$ |
| 5) | $T \to (\ E\ )$ | $T.node = E.node$ |
| 6) | $T \to$ **id** | $T.node =$ **new** $Leaf(\textbf{id}, \textbf{id}.entry)$ |
| 7) | $T \to$ **num** | $T.node =$ **new** $Leaf(\textbf{num}, \textbf{num}.val)$ |

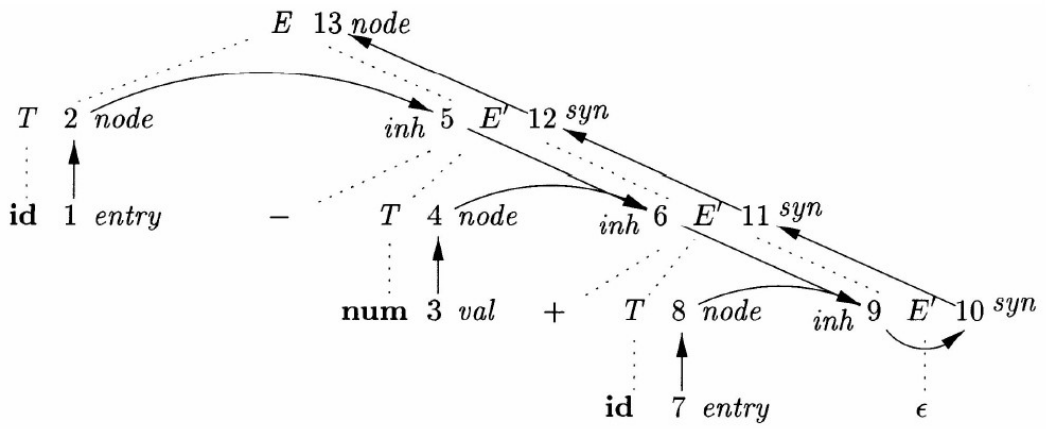
# 构造AST (L属性SDD)

## 例3：L属性SDD，构造语法树的语法制导定义（p205, 5.12）

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

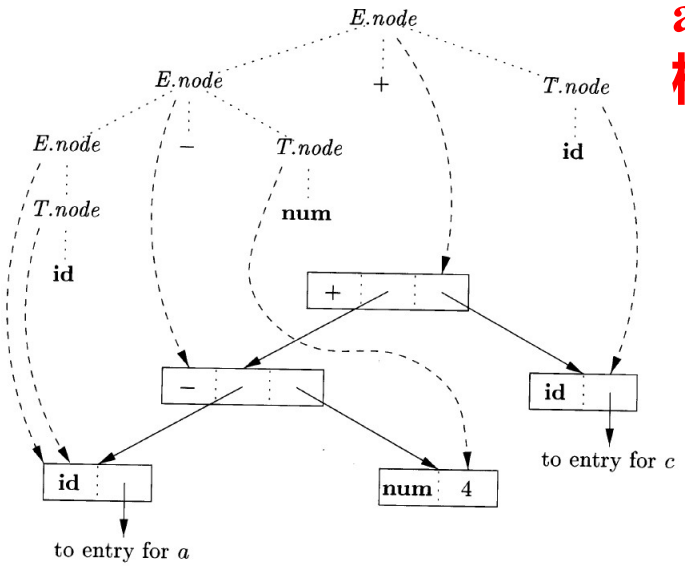
构造依赖图

a-4+c

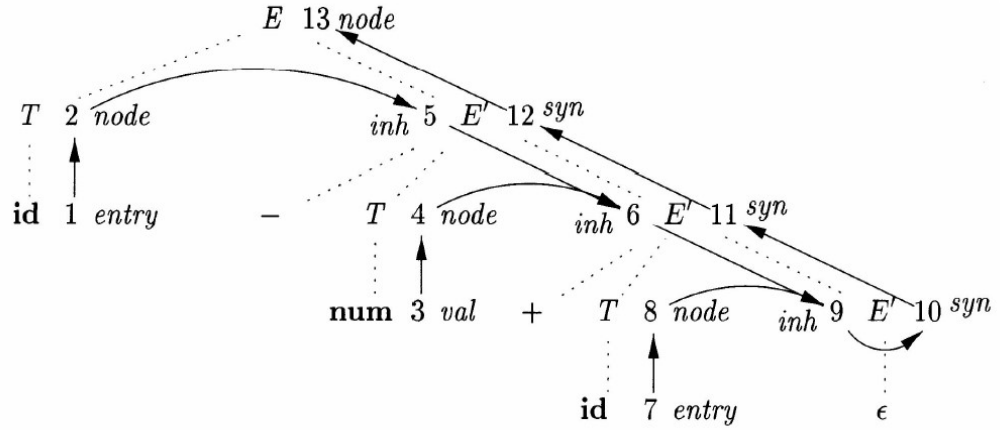


# 构造AST (L属性SDD)

## 例3：L属性SDD，构造语法树的语法制导定义（p205, 5.12）

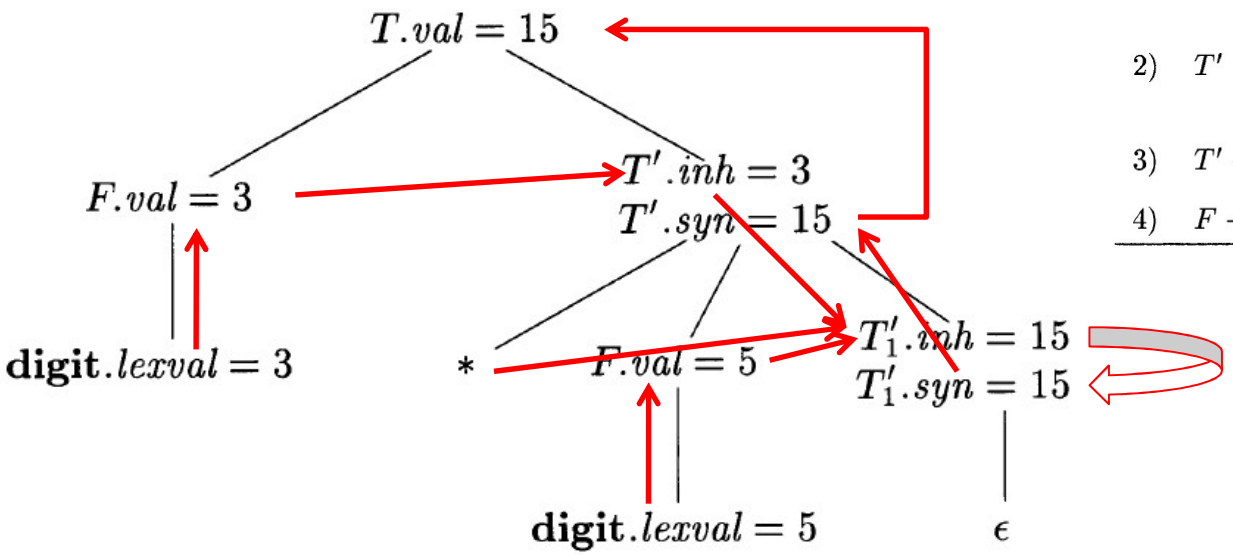



**a-4+c：不同的语法分析树与抽象语法树**



# 构造AST (L属性SDD)

- 和“3*5”例子一致




| | 产生式 | 语义规则 |
|---|---|---|
| 1) | $T \to F\,T'$ | $T'.inh = F.val$<br>$T.val = T'.syn$ |
| 2) | $T' \to *\,F\,T'_1$ | $T'_1.inh = T'.inh \times F.val$<br>$T'.syn = T'_1.syn$ |
| 3) | $T' \to \epsilon$ | $T'.syn = T'.inh$ |
| 4) | $F \to \mathbf{digit}$ | $F.val = \mathbf{digit}.lexval$ |

# 基本类型和数组类型的L属性定义

- **例4：简化的类型表达式的语法**

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

类型包括两个部分：T → B C
基本类型B；分量C

分量形如[3][4]：表示3X4的二维数组，e.g. int [3][4]

数组构造算符array：array(3,array(4,int))
表示抽象的3X4的二维数组

# 基本类型和数组类型的L属性定义

- **例4：简化的类型表达式的语法，int [2][3]的语法注释树**

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

array(2, array(3, integer))：可以理解为array返回当前数组类型。如果使用树来表示类型，array的作用就是构建了一个标号为array的结点，具有子结点：数字，类型

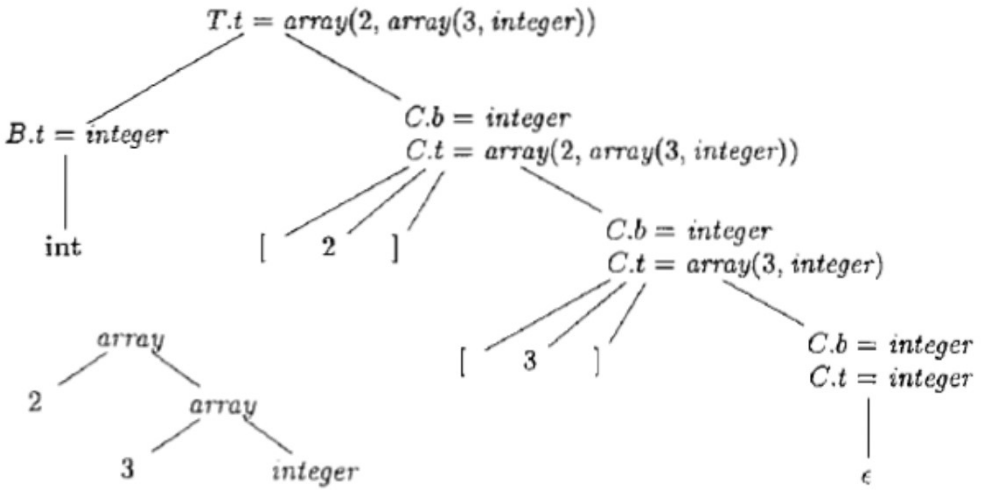



图 5-15 **int**[2][3]的类型表达式


# 9.4 语法制导的翻译方案

# 语法制导的翻译方案

- **语法制导的翻译方案(syntax-directed translation scheme, SDT)**
  - SDD的一种补充
  - 在产生式体内嵌入了程序片段的一个CFG
  - 程序片段成为语义动作，可以出现在产生式体中任何地方
  - 用 {语义动作}表示

# 可在语法分析过程中实现的SDT

- **实现SDT时，实际上并不会真的构造语法分析树，而是在分析过程中执行语义动作**
- **我们主要关注用SDT实现以下两类重要的SDD**
  - 基本文法是LR的，且SDD是S属性的 (最简单的情况)
  - 基本文法是LL的，且SDD是L属性的 (generalized)

# 可在语法分析过程中实现的SDT

- **判断是否可在分析过程中实现**
  - 将每个语义动作替换为一个独有的标记非终结符号；每个标记非终结符号M的产生式为$M \rightarrow \varepsilon$
  - 如果新的文法可以由某种方法进行分析，那么这个SDT就可以在这个分析过程中实现
  - 注意：这个方法没有考虑变量值的传递等要求

即使基础文法可以应用某种分析技术，仍可能因为动作的缘故导致此技术不可应用

# 判断SDT可否用特定分析技术实现例子

- $L \rightarrow E\ n\ M_1$ $\quad\quad$ $M_1 \rightarrow \varepsilon$
- $E \rightarrow E+T\ M_2$ $\quad\quad$ $M_2 \rightarrow \varepsilon$
- $E \rightarrow T\ M_3$ $\quad\quad$ $M_3 \rightarrow \varepsilon$
- …… ……

| | | | |
|---|---|---|---|
| $L$ | $\rightarrow$ | $E\ \mathbf{n}$ | $\{\ \text{print}(E.val);\ \}$ A1 |
| $E$ | $\rightarrow$ | $E_1 + T$ | $\{\ E.val = E_1.val + T.val;\ \}$ A2 |
| $E$ | $\rightarrow$ | $T$ | $\{\ E.val = T.val;\ \}$ A3 |
| $T$ | $\rightarrow$ | $T_1 * F$ | $\{\ T.val = T_1.val \times F.val;\ \}$ A4 |
| $T$ | $\rightarrow$ | $F$ | $\{\ T.val = F.val;\ \}$ A5 |
| $F$ | $\rightarrow$ | $(\ E\ )$ | $\{\ F.val = E.val;\ \}$ A6 |
| $F$ | $\rightarrow$ | $\mathbf{digit}$ | $\{\ F.val = \mathbf{digit}.lexval;\ \}$ A7 |

# 面向S属性SDD的后缀翻译方案

- **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**
- **构造方法**
  - 将每个语义规则看作是一个语义动作
  - 将所有的语义动作放在规则的最右端
- **按照产生式规约为左部非终结符时执行该动作**

# 面向S属性SDD的后缀翻译方案

- **实现桌上计算器的后缀SDT**

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

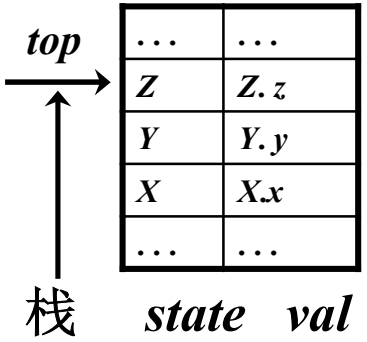
该SDD基本文法是LR的，并且SDD是S属性的，所以所构造的后缀SDT可以和LR分析的规约步骤一起正确执行

注意动作中对属性值的引用

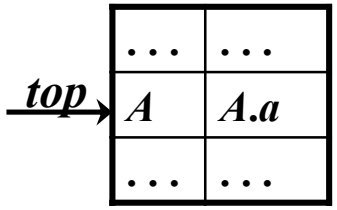
- 我们允许语句引用全局变量，局部变量，文法符号的属性
- 文法符号的属性只能被赋值一次

# 面向S属性SDD的后缀翻译方案

- **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 → 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




若产生式$A \rightarrow XYZ$的语义规则是
$A.a = f(X.x, Y.y, Z.z)$，
那么归约后：



# 面向S属性SDD的后缀翻译方案

- **分析栈实现的例子**
  - 假设语法分析栈存放在一个被称为stack的记录数组中，下标top指向栈顶
    - stack[top]是这个栈的栈顶
    - stack[top-1]指向栈顶下一个位置
    - 如果不同的文法符号有不同的属性集合，我们可以使用union来保存这些属性值
      - 归约时能够知道栈顶向下的各个符号分别是什么,因此我们也能够确定各个union中究竟存放了什么样的值

# 面向S属性SDD的后缀翻译方案

- **分析栈实现的例子**

**注意：stack[top-i]和文法符号的对应**

| 产生式 | 语义动作 |
| --- | --- |
| $L \to E$ **n** | { print($stack[top-1].val$); $top = top - 1$; } |
| $E \to E_1 + T$ | { $stack[top-2].val = stack[top-2].val + stack[top].val$; $top = top - 2$; } |
| $E \to T$ | |
| $T \to T_1 * F$ | { $stack[top-2].val = stack[top-2].val \times stack[top].val$; $top = top - 2$; } |
| $T \to F$ | |
| $F \to (\ E\ )$ | { $stack[top-2].val = stack[top-1].val$; $top = top - 2$; } |
| $F \to$ **digit** | |

# 产生式内部带有语义动作的SDT

- **例：把有加和减的中缀表达式翻译成后缀表达式**
  - 例如如果输入是8+5 –2，则输出是8 5 + 2 –

$E \rightarrow T\ R$

$R \rightarrow$ addop $T$ {*print* (addop.*lexeme*)} $R_1 \mid \varepsilon$

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

$E \Rightarrow T\ R \Rightarrow$ num {*print* (8)} $R$

$\Rightarrow$ num{*print* (8)}addop $T${*print* (+)}$R$

$\Rightarrow$ num{*print*(8)}addop num{*print*(5)}{*print* (+)}$R$

… {*print*(8)}{*print*(5)}{*print*(+)}addop $T${*print*(–)}$R$

… {*print*(8)}{*print*(5)}{*print*(+)}{*print*(2)}{*print*(–)}

# 产生式内部带有语义动作的SDT

- **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

# 产生式内部带有语义动作的SDT

- **不是所有的SDT都可以在分析过程中实现，例如：打印前缀表达式**
  - 不能和LL和LR分析同步进行

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

**移入/规约 冲突：**
**$M_2 \rightarrow \varepsilon$**
**$M_4 \rightarrow \varepsilon$**
**移入数字**

如何解决

# SDT的基本实现方法

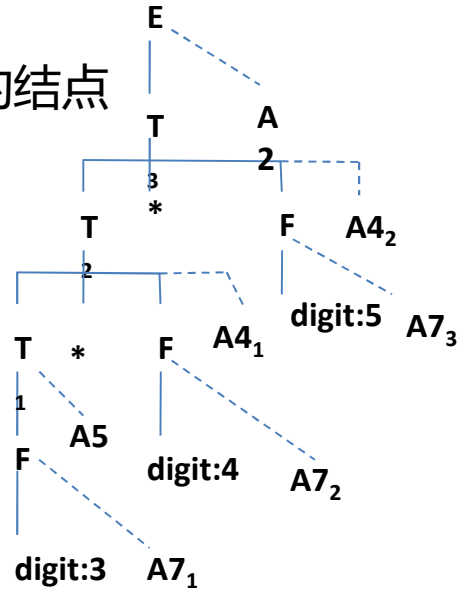
- **SDT的基本实现方法(不和语法分析同步进行)**
  - 忽略语义动作，对输入进行语法分析，产生一个语法分析树
  - 检查每个内部结点N，如果产生式为A → α，将α中各个动作当做N的附加子结点加入，使得N的子结点从左到右和α中符号及动作完全一致
  - 对分析树进行前序遍历，在访问虚拟结点时执行相应动作

# SDT的基本实现方法

**例如：语句3*4*5的分析树如右**

- **DFS可知动作执行顺序**
  - $A7_1$,A5, $A7_2$, $A4_1$, $A7_3$, $A4_2$, A2
  - 注意，一个动作的不同实例所访问的属性值属于不同的结点

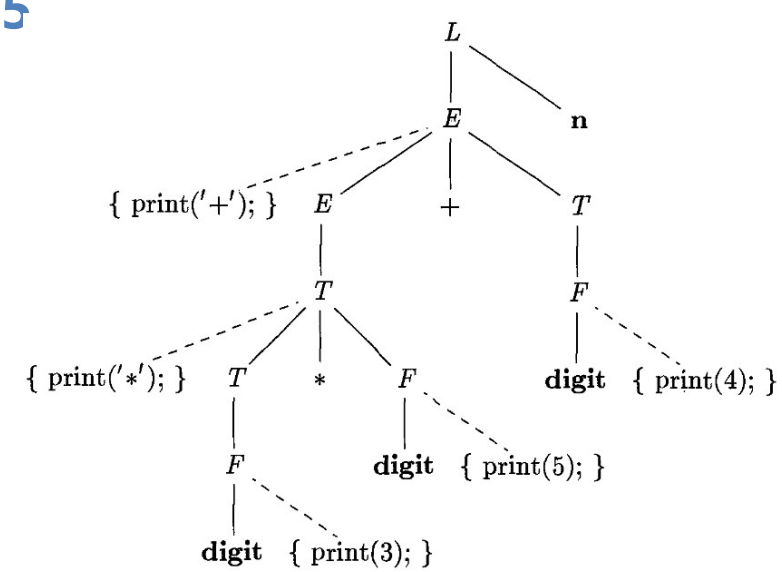
| | | | |
|---|---|---|---|
| $L$ | $\rightarrow$ | $E$ **n** | { print($E.val$); } A1 |
| $E$ | $\rightarrow$ | $E_1 + T$ | { $E.val = E_1.val + T.val$; } A2 |
| $E$ | $\rightarrow$ | $T$ | { $E.val = T.val$; } A3 |
| $T$ | $\rightarrow$ | $T_1 * F$ | { $T.val = T_1.val \times F.val$; } A4 |
| $T$ | $\rightarrow$ | $F$ | { $T.val = F.val$; } A5 |
| $F$ | $\rightarrow$ | $(\ E\ )$ | { $F.val = E.val$; } A6 |
| $F$ | $\rightarrow$ | **digit** | { $F.val =$ **digit**.$lexval$; } A7 |



# SDT的基本实现方法

**再如：语句3*5+4的分析树如右**

**分析后得到+*345**



# 从SDT中消除左递归

- **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**
  - $E \rightarrow E_1 + T$ {print( '+' );}
  - $E \rightarrow T$
- **转换后得到**
  - $E \rightarrow T\ R$
  - $R \rightarrow +\ T$ {print ( '+' );} $R$
  - $R \rightarrow \varepsilon$

$A \rightarrow A\alpha \mid \beta$　　$\beta(\alpha)^*$

转换为：$A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \varepsilon$

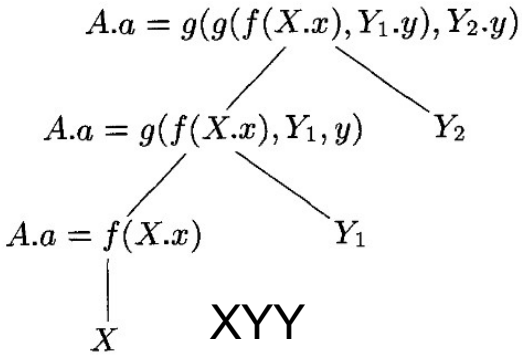
当一个SDD的动作有计算属性的值(而非仅仅打印输出)，我们必须小心处理。

# 消除左递归时SDT的一般转换形式

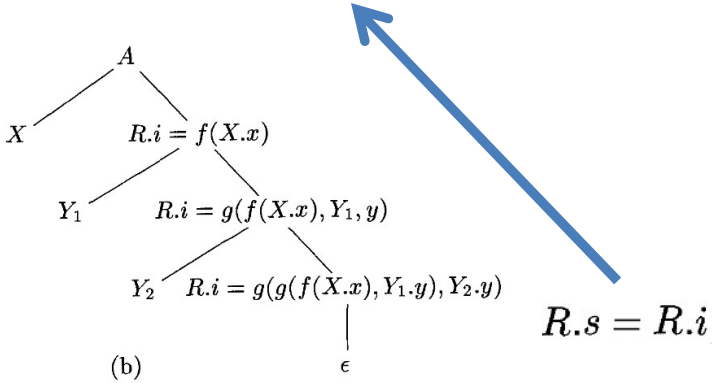
- **本课程仅针对只有单个递归产生式、单个非递归产生式且左递归非终结符只有单个属性的情况，给出通用解决方案。**

$$
\begin{array}{lcl} A & \rightarrow & A_1\ Y\ \{A.a = g(A_1.a, Y.y)\} \\ A & \rightarrow & X\ \{A.a = f(X.x)\} \end{array}
\longrightarrow
\begin{array}{lcl} A & \rightarrow & X\ R \\ R & \rightarrow & Y\ R \mid \epsilon \end{array}
$$

- **引入继承属性R.i，用来累计从A.a的值开始，不断应用g所得到的结果**




XYY



# 消除左递归时SDT的一般转换形式

$$
\begin{aligned}
A &\rightarrow A_1\ Y\ \{A.a = g(A_1.a, Y.y)\} \\
A &\rightarrow X\ \{A.a = f(X.x)\}
\end{aligned}
$$

$$
\begin{aligned}
A &\rightarrow X\ \{R.i = f(X.x)\}\ R\ \{A.a = R.s\} \\
R &\rightarrow Y\ \{R_1.i = g(R.i, Y.y\}\ R_1\ \{R.s = R_1.s\} \\
R &\rightarrow \epsilon\ \{R.s = R.i\}
\end{aligned}
$$

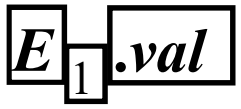
# 面向L属性SDD的翻译方案

- **产生式内部带有语义动作的SDT**
  - 将一个L属性的SDD转换为一个SDT的规则如下
    - 将每个语义规则看作是一个语义动作
    - 将语义动作放到相应产生式的适当位置
      - 计算A的继承属性的动作插入到产生式体中对应的A的左边，如果A的继承属性之间具有依赖关系，则需要对计算动作进行排序
      - 计算产生式头的综合属性的动作在产生式的最右边

# 面向L属性SDD的翻译方案

- **例　数学排版语言EQN**

*E* **sub 1 .val**




$S \rightarrow B$

$B \rightarrow B_1\ B_2$

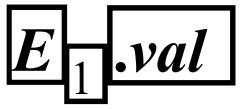
$B \rightarrow B_1$ sub $B_2$

$B \rightarrow$ text

# 面向L属性SDD的翻译方案

- **例　数学排版语言EQN（语法制导定义）**

*E* sub 1 .val




| 产 生 式 | 语 义 规 则 |
|---|---|
| $S \rightarrow B$ | $B.ps = 10;\ S.ht = B.ht$ |
| $B \rightarrow B_1\ B_2$ | $B_1.ps = B.ps;\ B_2.ps = B.ps;$<br>$B.ht = max(B_1.ht, B_2.ht)$ |
| $B \rightarrow B_1$ sub $B_2$ | $B_1.ps = B.ps;\ B_2.ps = shrink(B.ps);$<br>$B.ht = disp\ (B_1.ht, B_2.ht)$ |
| $B \rightarrow$ text | $B.ht = \text{text}.h \times B.ps$ |

# 面向L属性SDD的翻译方案

- 例　数学排版语言EQN（翻译方案）

$S \rightarrow$ {*B.ps* = 10 }　　*B*继承属性的计算

*B*　　{*S.ht* = *B.ht* }　　位于*B*的左边

# 面向L属性SDD的翻译方案

- 例　数学排版语言EQN（翻译方案）

$S \rightarrow$ {*B.ps* = 10 }　　*B*综合属性的计算

　　*B*　　{*S.ht* = *B.ht* }　　放在右部末端

# 面向L属性SDD的翻译方案

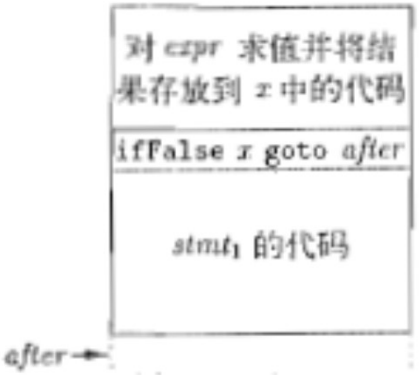
- **例　数学排版语言EQN（翻译方案）**

$S \rightarrow$ {$B.ps = 10$ }
$B$ {$S.ht = B.ht$ }
$B \rightarrow$ {$B_1.ps = B.ps$ }
$B_1$ {$B_2.ps = B.ps$ }
$B_2$ {$B.ht = max(B_1.ht, B_2.ht)$ }
$B \rightarrow$ { $B_1.ps = B.ps$ }
$B_1$
sub { $B_2.ps = shrink(B.ps)$ }
$B_2$ {$B.ht = disp\ (B_1.ht, B_2.ht)$ }
$B \rightarrow$ text {$B.ht = \text{text}.h \times B.ps$ }

# 面向L属性SDD的翻译方案

- **例** S → **while** (*C*) $S_1$

```
ifFalse x goto L    如果 x 为假，下一步执行标号为 L 的指令
ifTrue x goto L     如果 x 为真，下一步执行标号为 L 的指令
goto L              下一步执行标号为 L 的指令
```




- **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**

# 面向L属性SDD的翻译方案

- **语义动作**
  a) L1=new( )；L2=new( )：计算临时值
  b) C.false = S.next; C.true = L2：计算C的继承属性
  c) $S_1$.next = L1：计算$S_1$的继承属性
  d) S.code= …：计算S的综合属型

$A \rightarrow \{B.i = f(A.i);\}\ B\ C$

$A \rightarrow M\ B\ C$
$M \rightarrow \{M.i = A.i;\ M.s = f(M.i);\}$

| | |
|---|---|
| $S \rightarrow \textbf{while}\ (\ C\ )\ S_1$ | $L1 = new();$<br>$L2 = new();$<br>$S_1.next = L1;$<br>$C.false = S.next;$<br>$C.true = L2;$<br>$S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code$ |

# 面向L属性SDD的翻译方案

- **根据放置语义动作的规则得到如下SDT**

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

# 作业

- 图5-4的SDD扩充如右图所示，为教材P198练习5.1.1的第二个表达式画出注释语法树和依赖图
- 教材P203：5.2.4
- 教材P207：5.3.1, 5.3.2

| P | 语法规则 | |
|---|---|---|
| 1) | L -> En | L.val = E.val |
| 2) | E -> TE' | E'.inh = T.val<br>E.val = E'.syn |
| 3) | E' -> +$TE_1'$ | E_1'.inh = E'.inh + T.val<br>E'.syn = E_1'.syn |
| 4) | E' -> ε | E'.syn = E'.inh |
| 5) | T -> FT' | T'.inh = F.val<br>T.val = T'.syn |
| 6) | T' -> *$FT_1'$ | T_1'.inh = T'.inh * F.val<br>T'.syn = T_1'.syn |
| 7) | T' -> ε | T'.syn = T'.inh |
| 8) | F -> (E) | F.val = E.val |
| 9) | F -> digit | F.val = digit.lexval |

# Thank you!