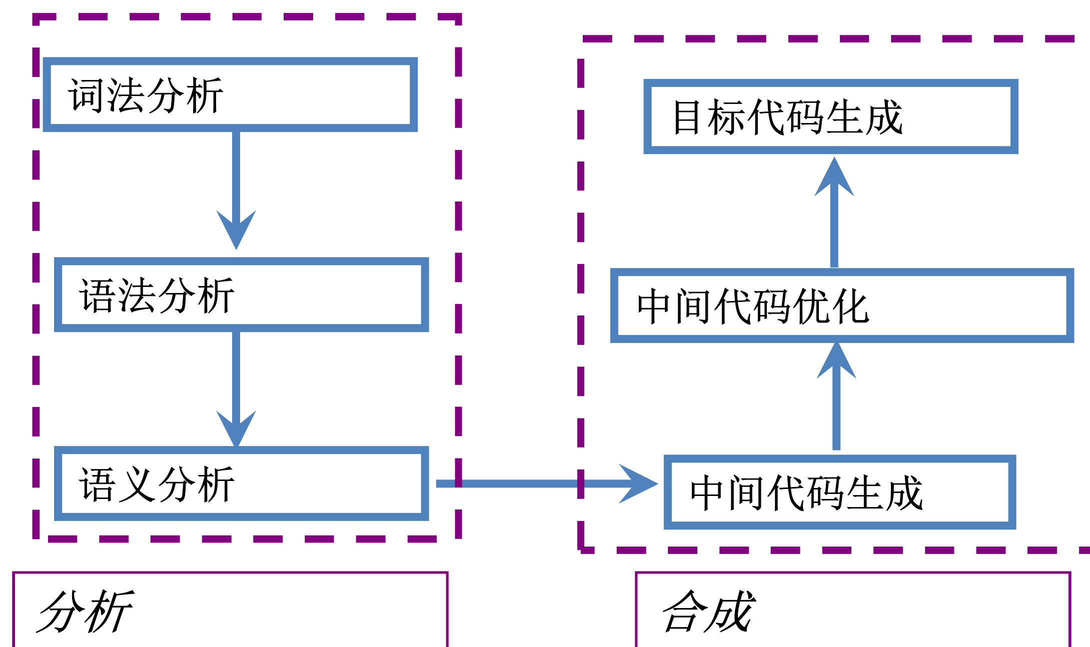
Lecture 6: 语法制导的翻译

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

6.1 概述

语义分析在编译程序中的作用



语义分析在编译程序中的作用

- 用A→ α 进行归约表达的是什么意思
- $E \rightarrow E_1 + T$
- E_1的值+T的值的结果作为E的值——即：取来E_1的值和T的值做加法运算，结果作为E的值
  - $E.val = E_1.val + T.val$

语义分析在编译程序中的作用

- 翻译的任务
  - 首先是语义分析和正确性检查，若正确，则翻译成中间代码或目标代码。
- 语义分析基本思想
  - 根据翻译的需要设置文法符号的属性，以描述语法结构的语义。
  - 例如，一个变量的属性有类型，层次，存储地址等。表达式的属性有类型，值等。
  - 属性值的计算和产生式相联系。随着语法分析的进行，执行属性值的计算，完成语义分析和翻译的任务。

语义分析在编译程序中的作用

- 语义检查
  - 例：类型、运算、维数、越界
- 语义处理
  - 例：变量的存储分配
  - 例：表达式的求值
  - 例：语句的翻译（中间代码的生成）
- 总目标：生成等价的中间代码

语义分析在编译程序中的作用

- 语法制导翻译的概念描述
  - 在进行语法分析的同时，完成相应的语义处理
  - $E \rightarrow E_1 + E_2$ $E.val := E_1.val + E_2.val$

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**

```
int x;
float* f();
x();
x = f();
```

- int x; —— 符合变量声明的语法、语义
- float* f(); —— 符合函数声明的语法、语义
- x(); —— 符合函数调用的语法、不符合语义
- x = f(); —— 符合赋值语句的语法、不符合语义

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！！！！

语法制导翻译的基本思想

- 在进行语法分析的同时，完成相应的语义处理。也就是说，一旦语法分析器识别出一个语法结构就要立即对其进行翻译，翻译是根据语言的语义进行的，并通过调用事先为该语法结构编写的语义子程序来实现。
- 对文法中的每个产生式附加一个/多个语义动作(或语义子程序)，在语法分析的过程中，每当需要使用一个产生式进行推导或归约时，语法分析程序除执行相应的语法分析动作外，还要执行相应的语义动作(或调用相应的语义子程序)

语法制导翻译的基本思想

- 词法分析
- 语法分析
- 语义分析
- 中间代码生成
- 代码优化
- 目标代码生成

语义分析、中间代码生成 } 语义翻译

语法分析、语义翻译 } 语法制导翻译 (Syntax-Directed Translation)

语法制导翻译使用*CFG*来引导对语言的翻译，是一种面向文法的翻译技术

语法制导翻译的基本思想

- **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

语法制导翻译的基本思想

- 如何表示语义信息？
  - 为*CFG*中的文法符号设置语义属性，用来表示语法成分对应的语义信息
- 如何计算语义属性？
  - 文法符号的语义属性值是用与文法符号所在产生式（语法规则）相关联的语义规则来计算的
  - 对于给定的输入串x，构建x的语法分析树，并利用与产生式（语法规则）相关联的语义规则来计算分析树中各结点对应的语义属性值

两种经典处理方法

- **语法制导定义 Syntax-Directed Definitions（SDD）**
  - 将文法符号和某些属性相关联
  - 并通过语义规则来描述如何计算属性的值
  - $E \rightarrow E_1 + T \qquad E.val = E_1.val + T.val$
- **语法制导的翻译方案 Syntax-Directed Translation Scheme（SDT）**
  - 在产生式体中加入语义动作，并在适当的时候执行这些语义动作
  - $E \rightarrow E_1 + T \quad \{E.val = E_1.val + T.val;\}$

上述两种方法都是为了将语义规则同语法规则（产生式）联系起来

6.2 语法制导定义SDD

语法制导定义SDD

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

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
  - 决定于翻译的目的，例如：产生什么样的中间代码

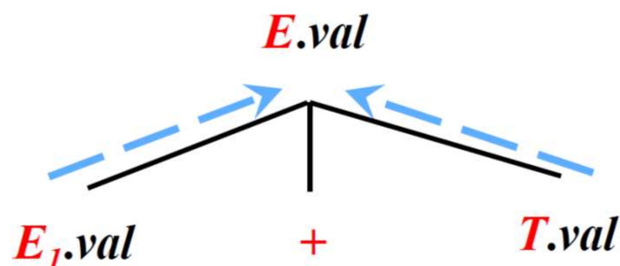
文法属性

- **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的语法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
  - 继承属性

文法属性 – 综合属性

- 在分析树结点 N 上的非终结符 A 的**综合属性**只能通过 N 的子结点或 N 本身的属性值来定义
  - 例

产生式	语义规则
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

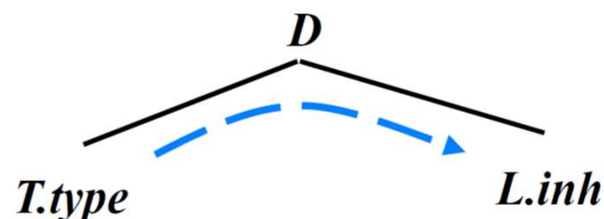


终结符可以具有综合属性。终结符的综合属性值是由词法分析器提供的词法值，因此在SDD中没有计算终结符属性值的语义规则

文法属性 – 继承属性

- 在分析树结点N上的非终结符A的**继承属性**只能通过N的父结点、N的兄弟结点或N本身的属性值来定义
  - 例

产生式	语义规则
$D \rightarrow T\ L$	$L.inh = T.type$



终结符没有继承属性。终结符从词法分析器处获得的属性值被归为综合属性值

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

发现了什么问题?

依赖关系!

语法分析树上的SDD求值

- **例子：目标是计算表达式行L的值（属性val）**
  - 计算L的val值需要E的val值
  - E的val值又依赖于E和T的val值
  - …
  - 终结符号digit有综合属性lexval

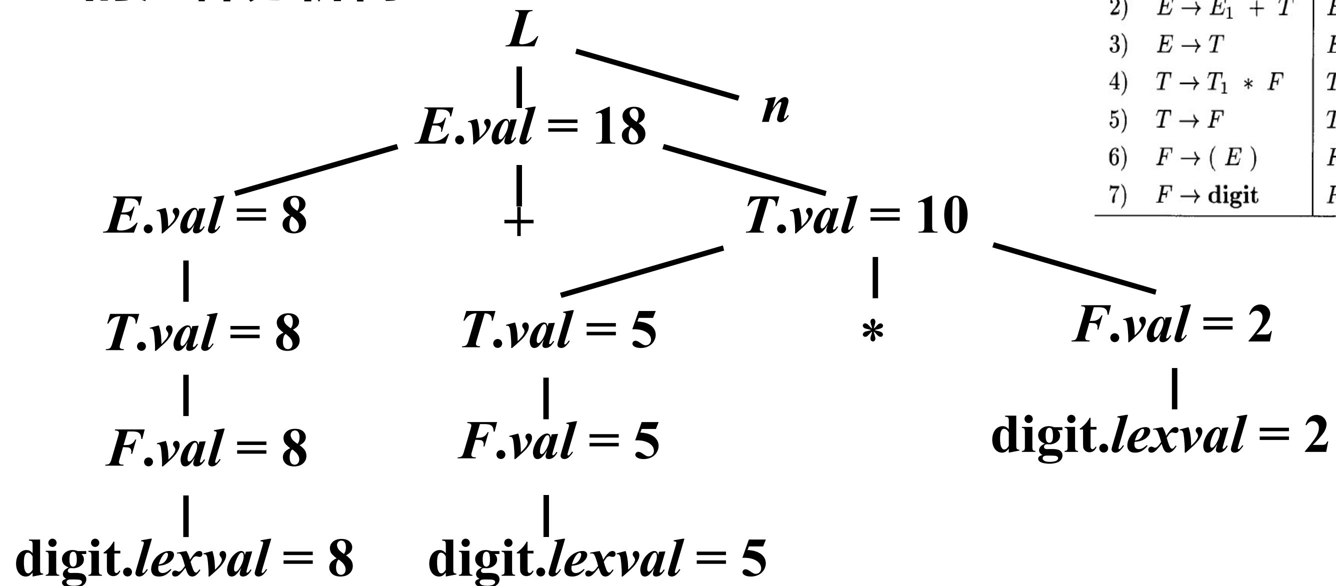
为不同位置的文法符号
设置下标，以示区分

	产生式	语义规则
1)	$L \to E\ \mathbf{n}$	$L.val = E.val$
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)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

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树

8+5∗2 n的注释分析树

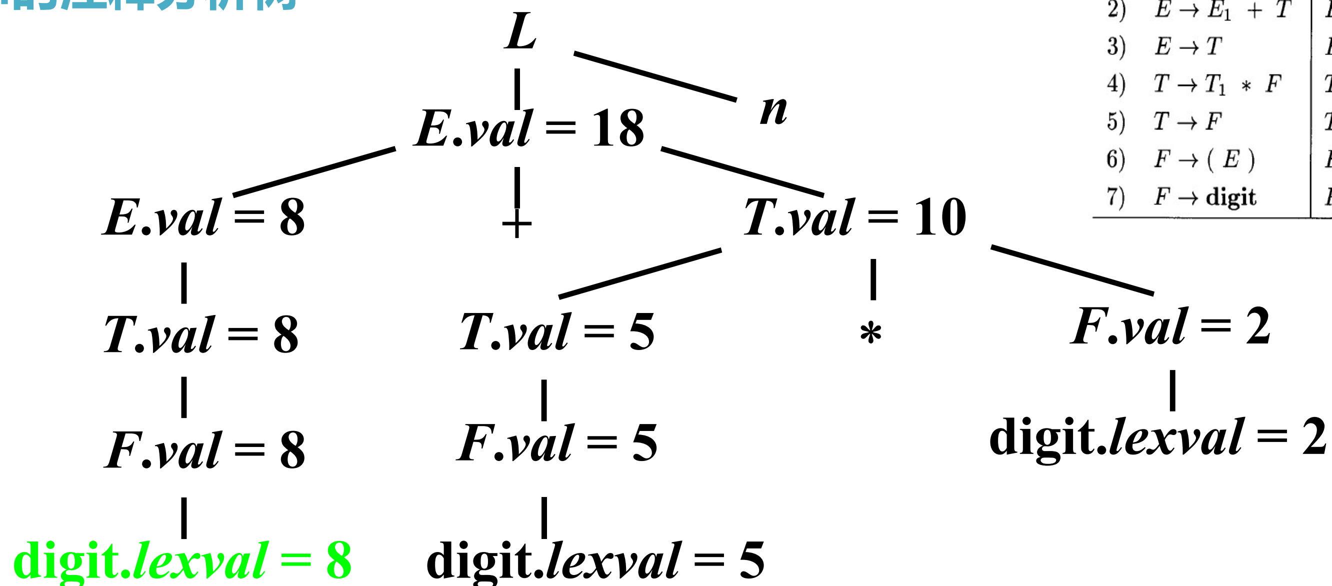


	产生式	语义规则
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**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

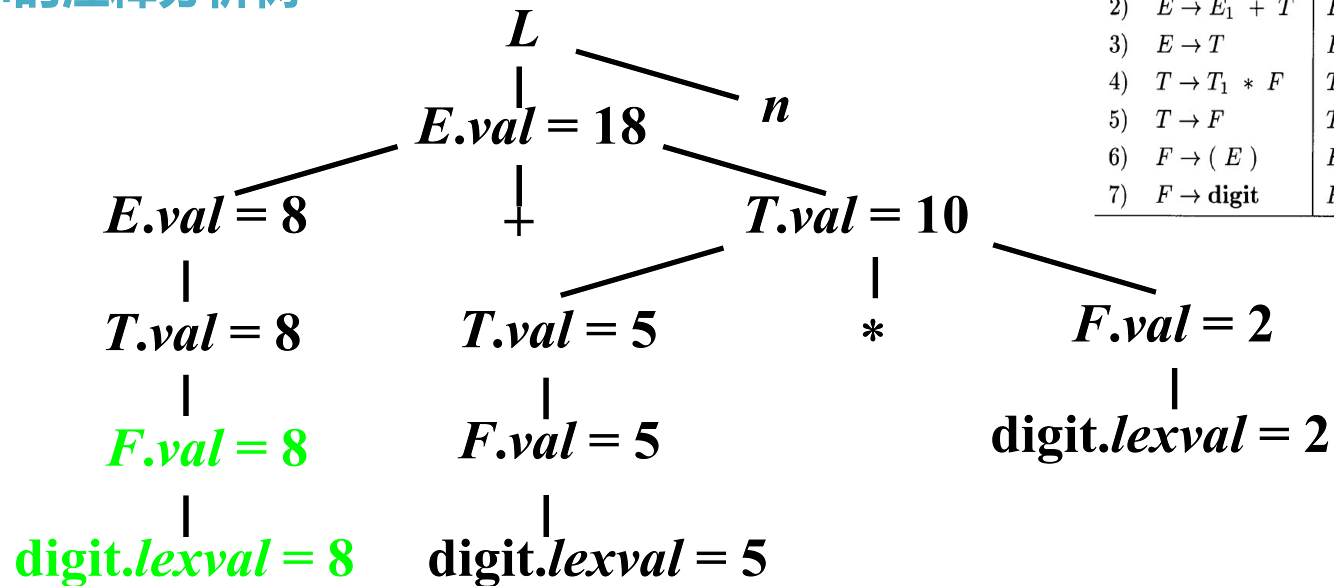


	产生式	语义规则
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

语法分析树上的SDD求值

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

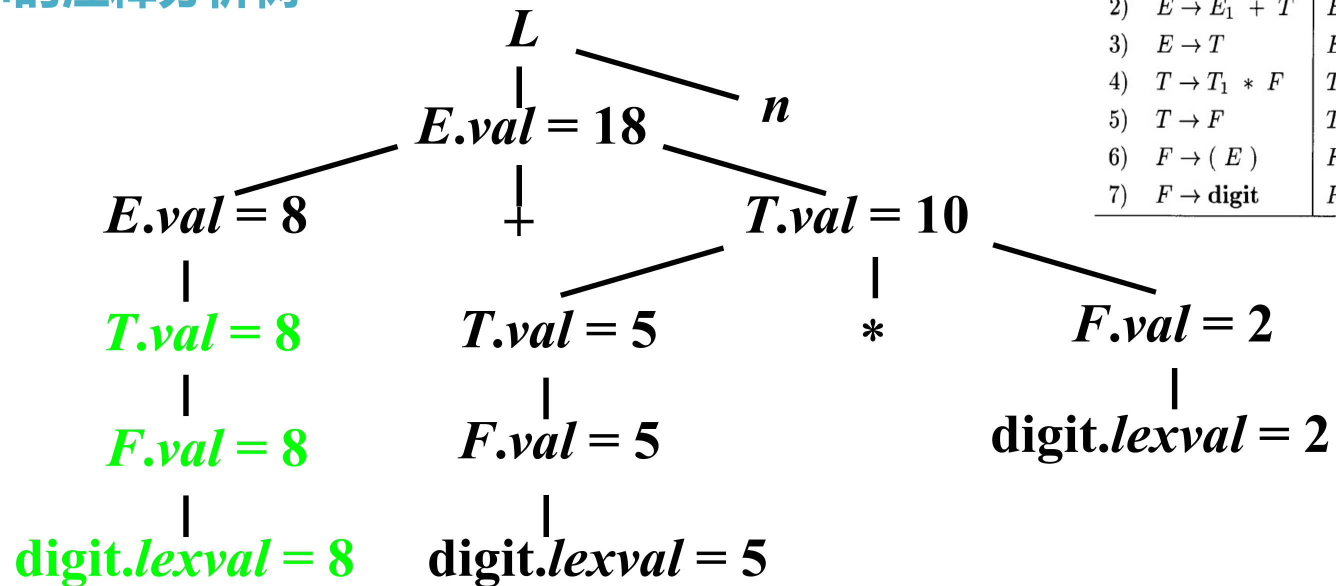


	产生式	语义规则
1)	$L \to E\ \mathbf{n}$	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

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

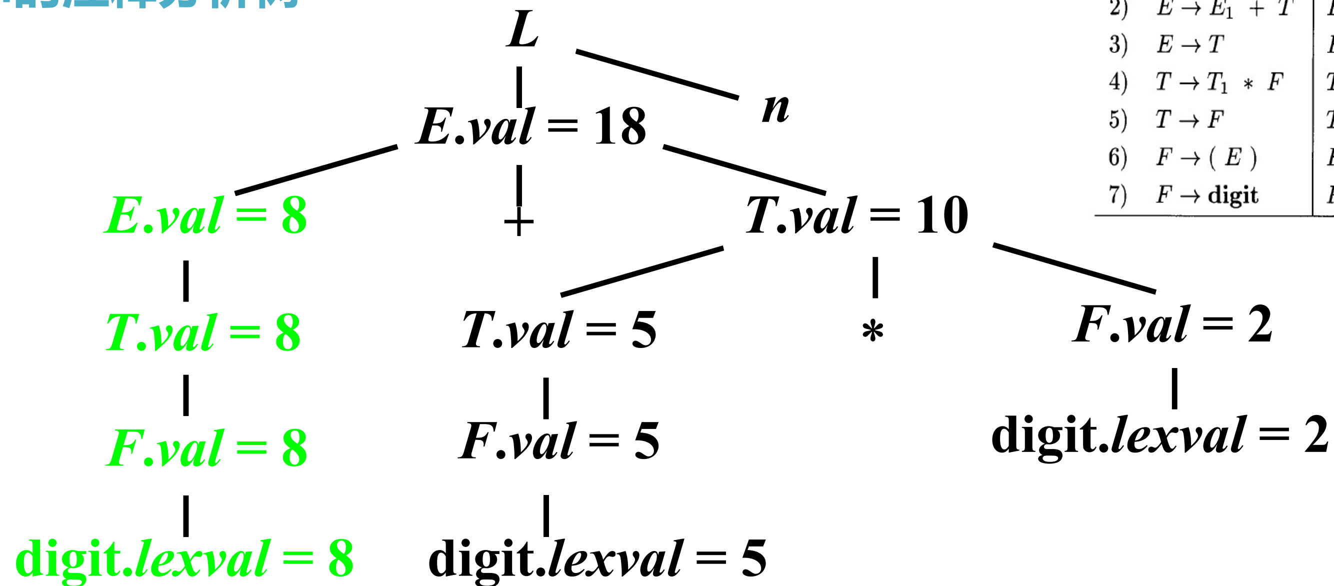


	产生式	语义规则
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

语法分析树上的SDD求值

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

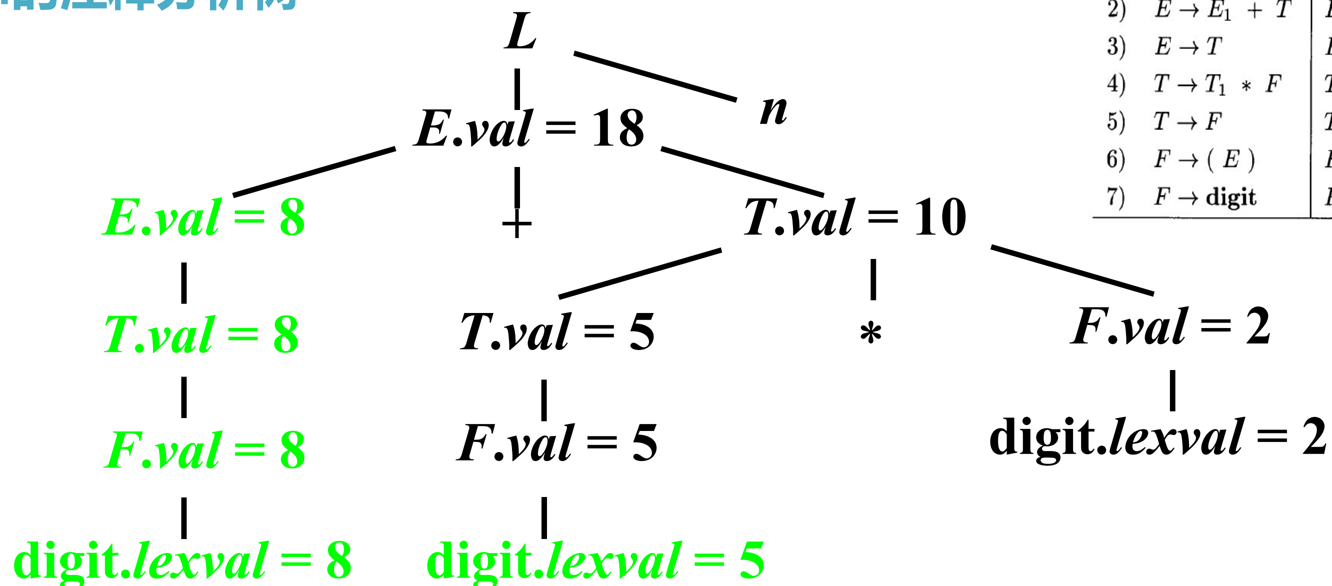


	产生式	语义规则
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

语法分析树上的SDD求值

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

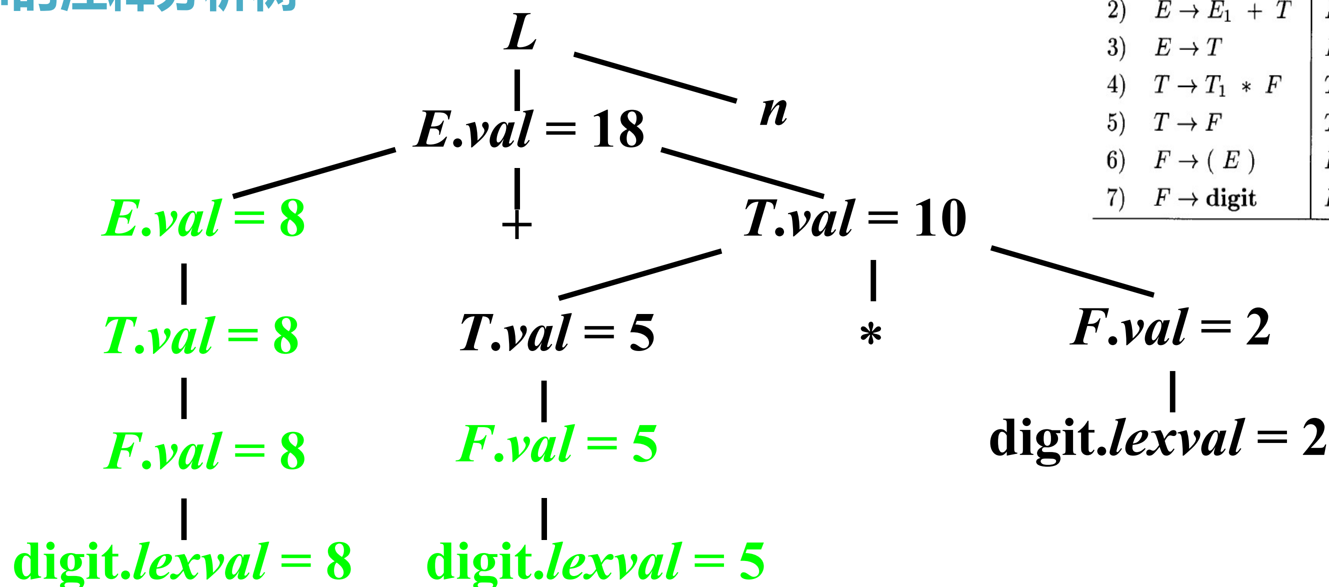


	产生式	语义规则
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

语法分析树上的SDD求值

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

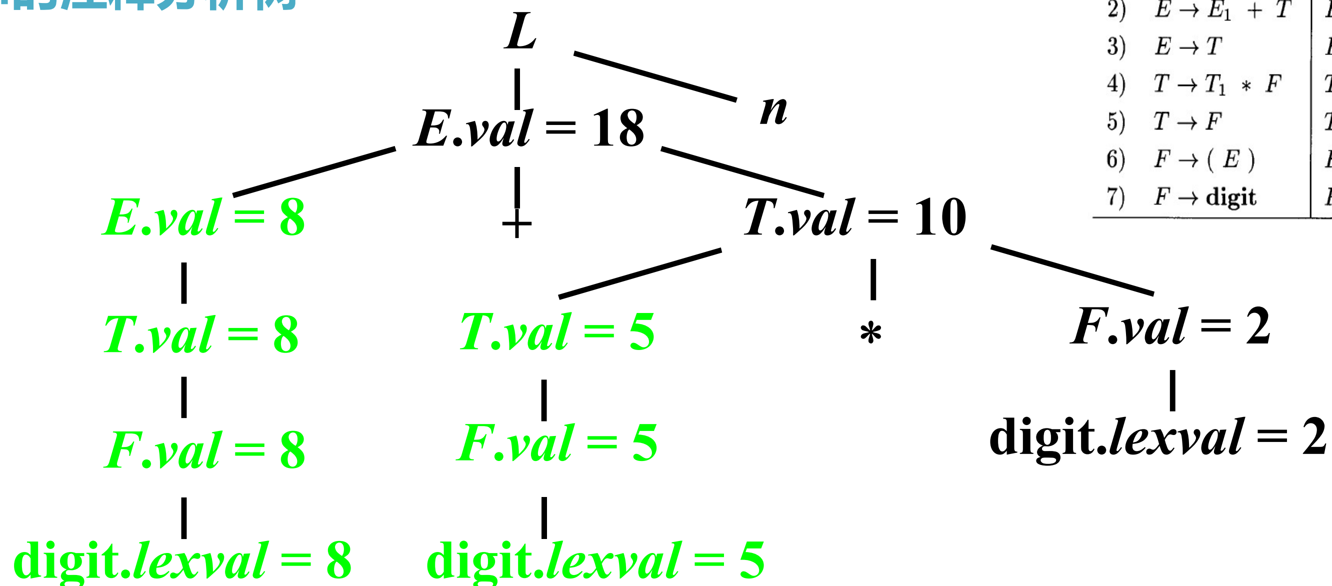


	产生式	语义规则
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

语法分析树上的SDD求值

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

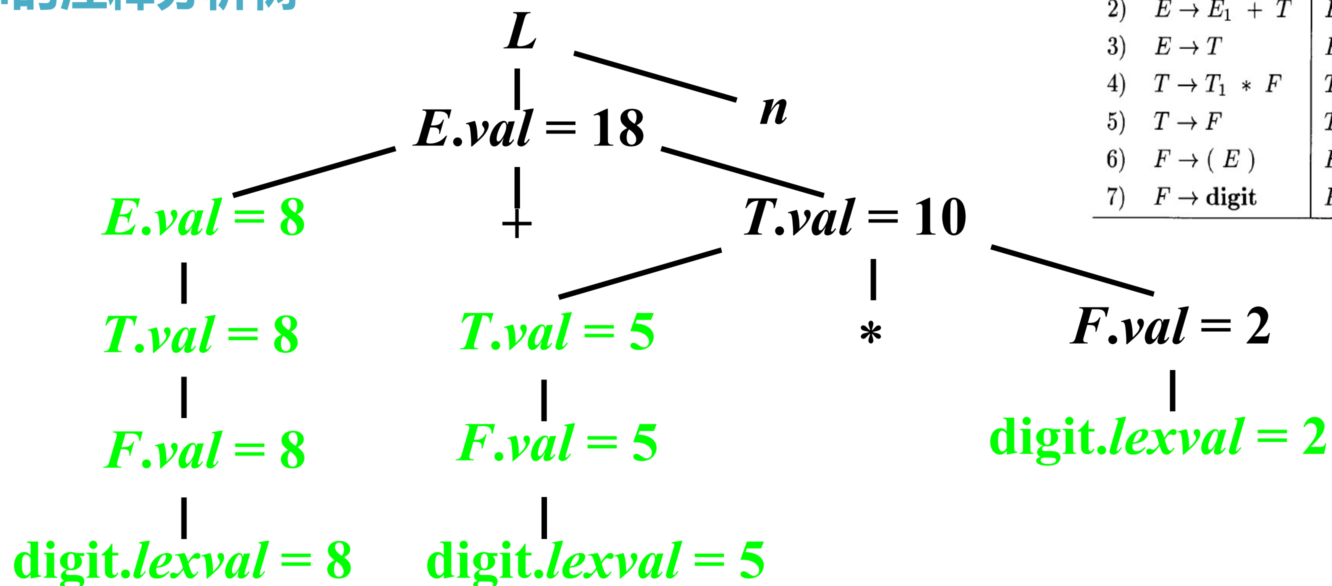


	产生式	语义规则
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**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

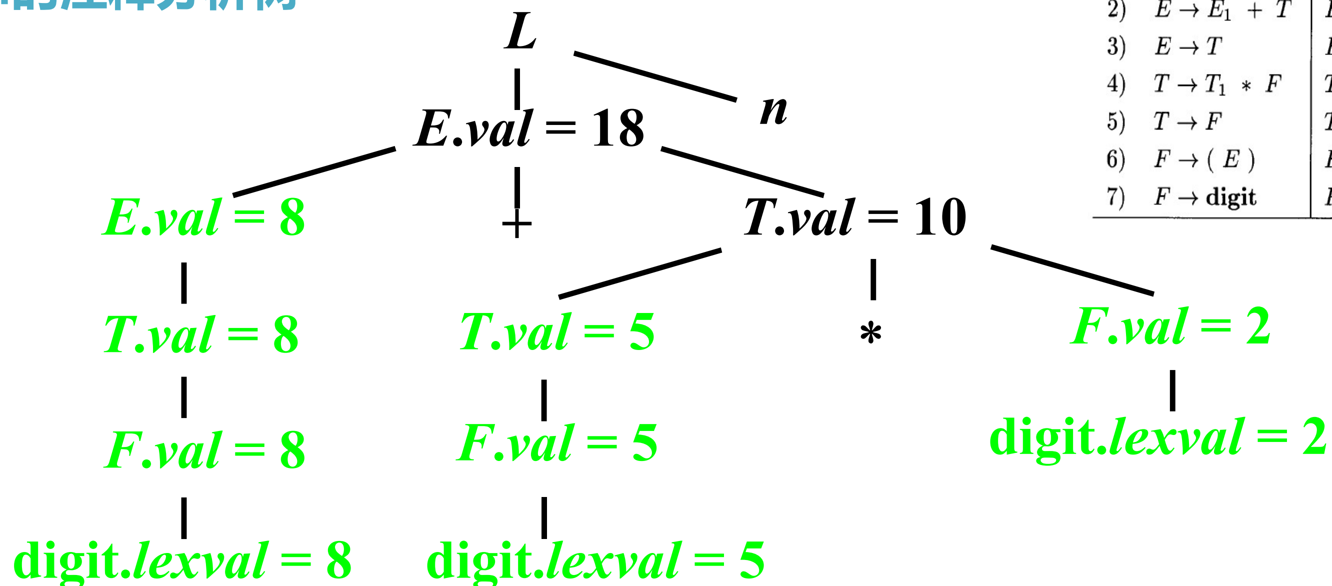


	产生式	语义规则
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val$ = **digit**.lexval

语法分析树上的SDD求值

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

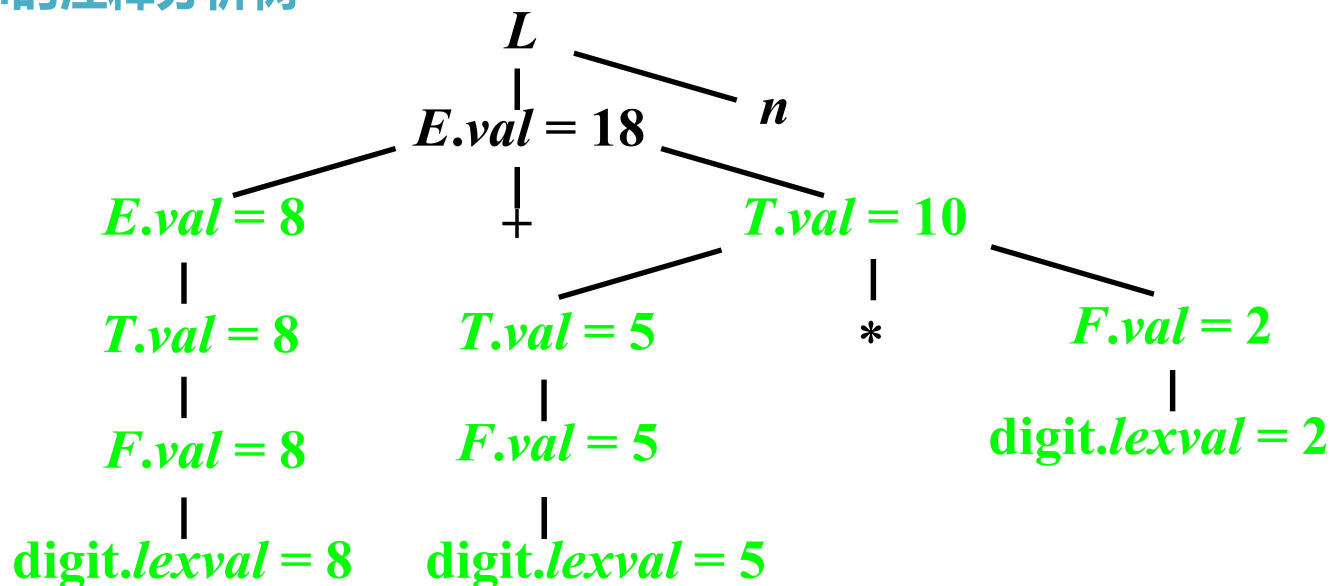


	产生式	语义规则
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

语法分析树上的SDD求值

分析树各结点属性的计算可以自下而上地完成

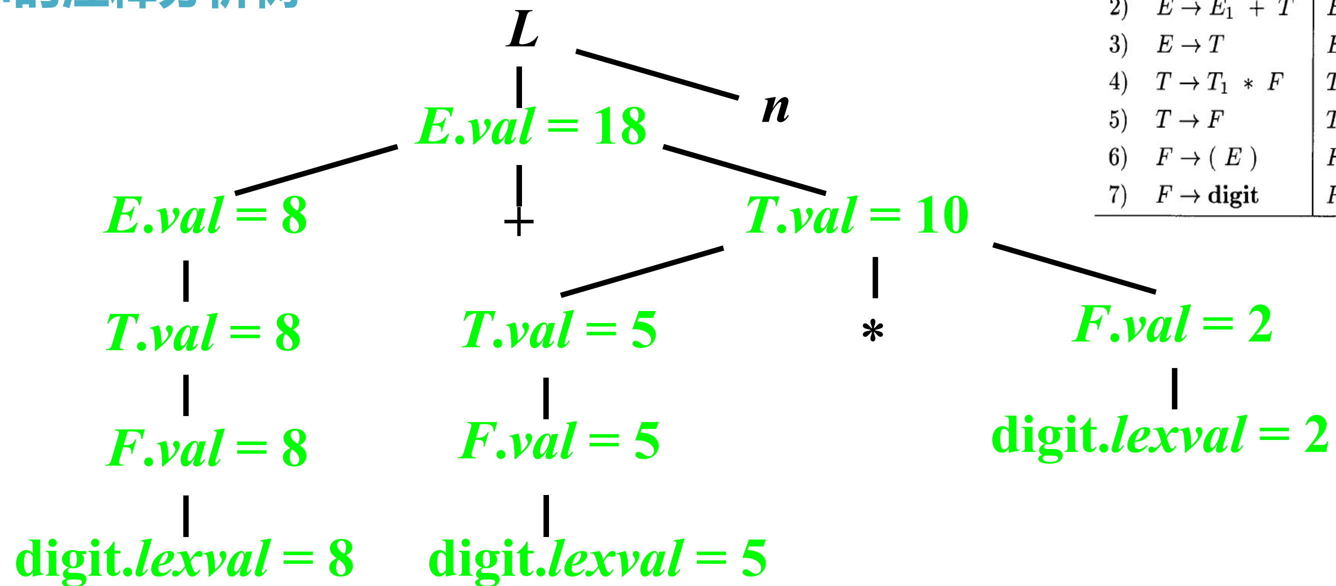
8+5∗2 n的注释分析树



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
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

两种属性

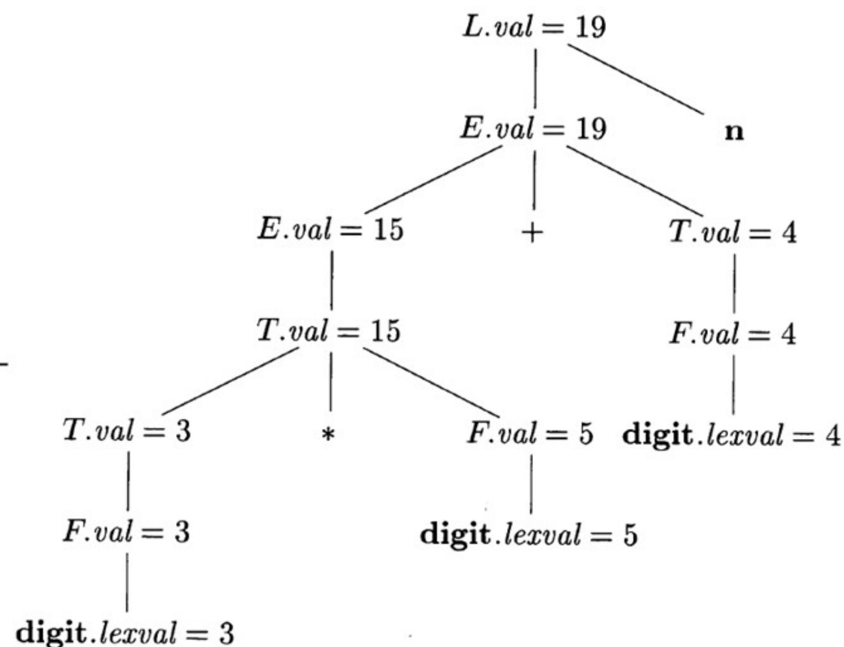
- **综合属性**
  - 在分析树结点N上的非终结符号A的属性值由N对应的产生式所关联的语义规则来定义，通过N的子结点或N本身的属性值来定义
- **继承属性**
  - 结点N的属性值由N的父结点所关联的语义规则来定义，依赖于N的父结点、N本身和N的兄弟结点上的属性值

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

综合属性例子

	产生式	语义规则
1)	$L \to E\ \mathbf{n}$	$L.val = E.val$
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)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

3*5+4n的注释分析树

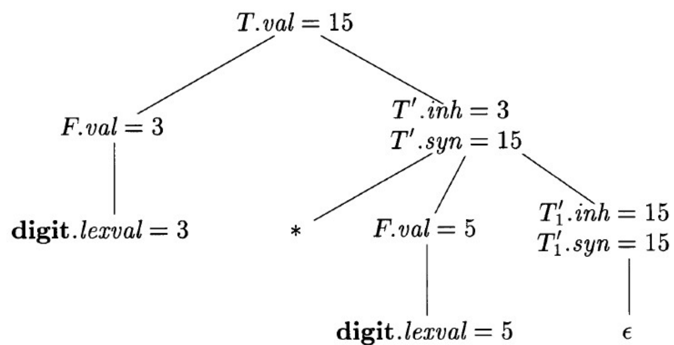


继承属性例子

产生式	语义规则
1) $L \to E\ \mathbf{n}$	$L.val = E.val$
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) $F \to \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

消除左递归

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

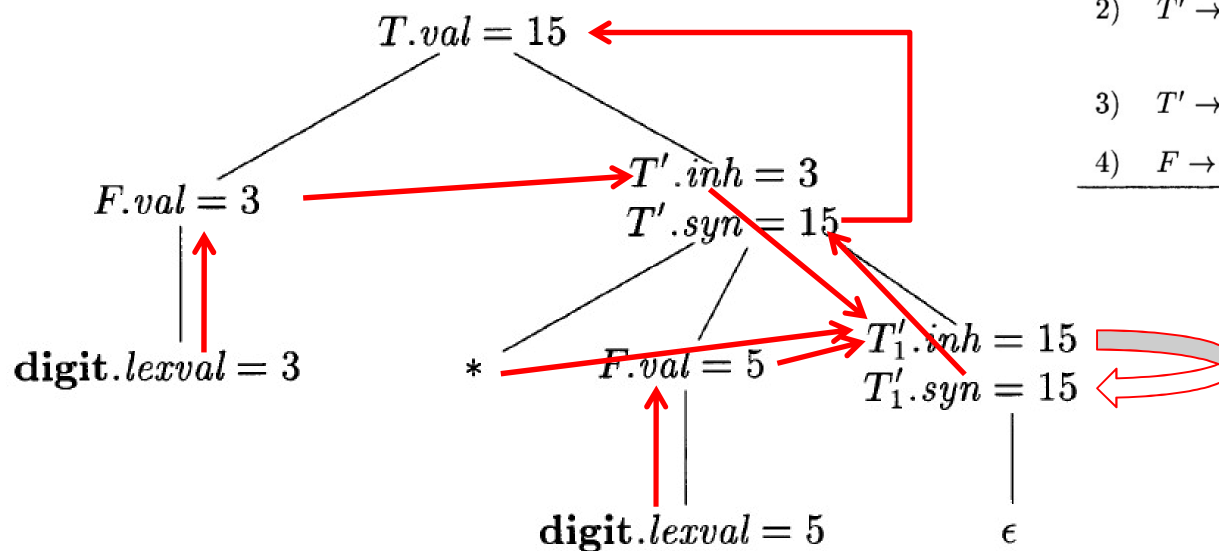


消除左递归之后，我们无法直接使用属性val进行处理：比如规则：T→FT’ T’ →*FT’ --- T对应的项中，第一个因子对应于F，而运算符却在T’中。需要继承属性来完成这样的计算

继承属性例子

■ **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) $T \to F\, T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \to *\, F\, T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \to \epsilon$	$T'.syn = T'.inh$
4) $F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

属性值的计算

- **按照分析树中分支对应的产生式，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$f(N_1.b_1,N_2.b_2,\ldots,N_k.b_k)$，那么我们需要先算出$N_1.b_1,N_2.b_2,\ldots,N_k.b_k$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**
  - 例如：如果SDD只包含综合属性？ --- 一定可以按照自底向上的方式求值
  - 再如：下面的SDD不能计算
    - $A \rightarrow B$　　　　A.s=B.i;　B.i=A.s+1;

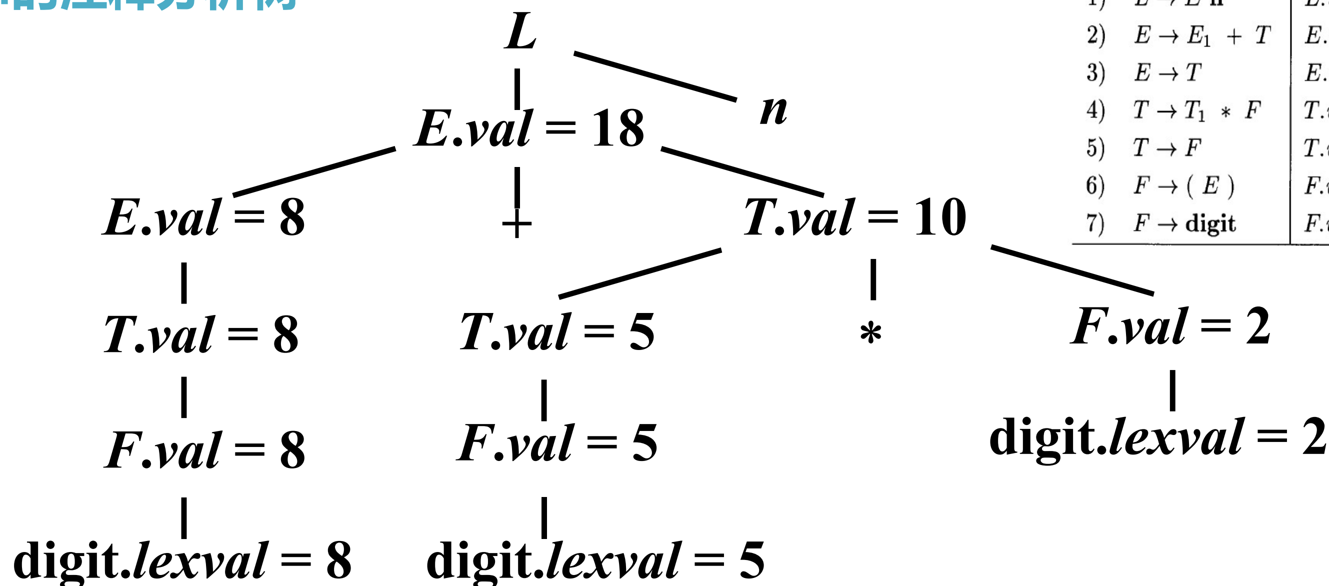
只包含综合属性的SDD

- **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LHS非终结符号的属性值
- **S属性的SDD可以和LR语法分析器一起实现**
  - 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
- **语义规则不应该有复杂的副作用**
  - 要求副作用不影响其它属性的求值
  - 没有副作用的SDD称为属性文法

只包含综合属性的SDD

S属性SDD的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



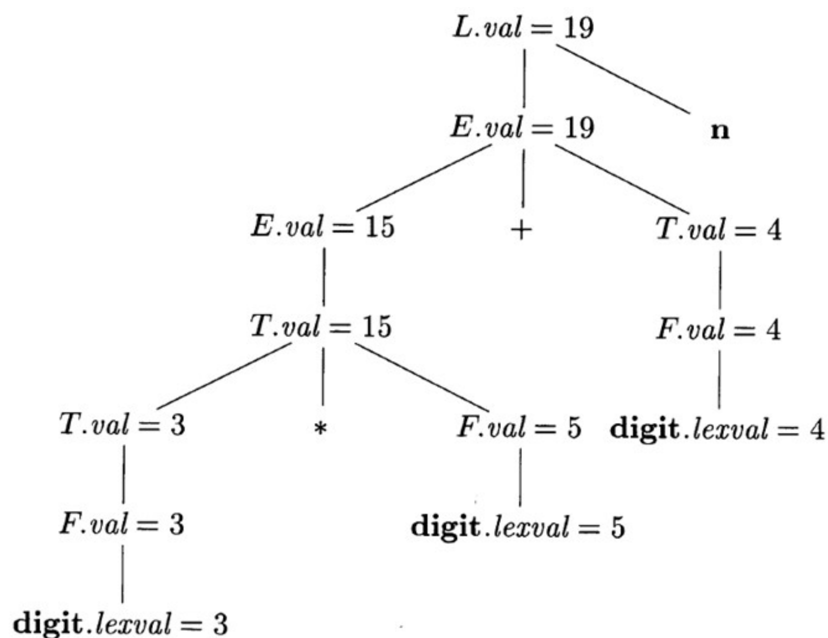
	产生式	语义规则
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

只包含综合属性的SDD

- S属性SDD的分析树各结点属性的计算可以**自下而上地完成**

3*5+4n

产生式	语义规则
1) $L \to E\ \mathbf{n}$	$L.val = E.val$
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) $F \to \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$



含有继承属性的SDD

- **左递归文法使用自顶向下分析时，要消除左递归**

产生式
1) $L \to E$ **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) $F \to$ **digit**

PRODUCTION
1) $T \to F\,T'$
2) $T' \to * \, F\,T'_1$
3) $T' \to \epsilon$
4) $F \to$ **digit**

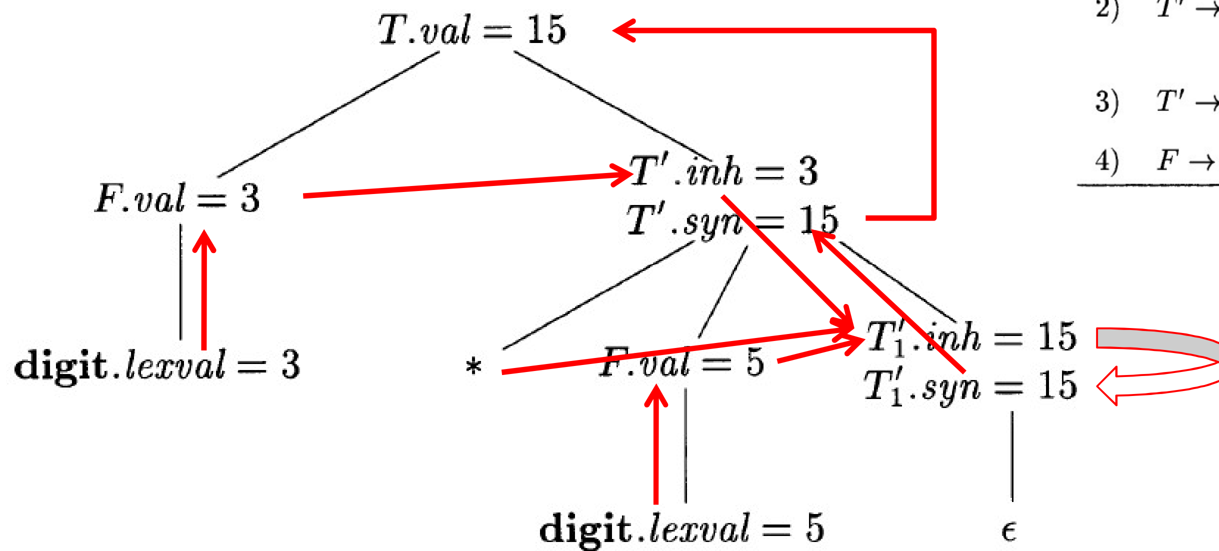
产生式	语义规则
1) $T \to F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \to * \, F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \to \epsilon$	$T'.syn = T'.inh$
4) $F \to$ **digit**	$F.val =$ **digit**$.lexval$

出现了继承属性！

含有继承属性的SDD

- ## 3*5注释分析树

  注意观察inh属性是如何传递的



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

SDD的求值顺序

- 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
- 使用**依赖图**来表示**计算顺序**
- 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

依赖图

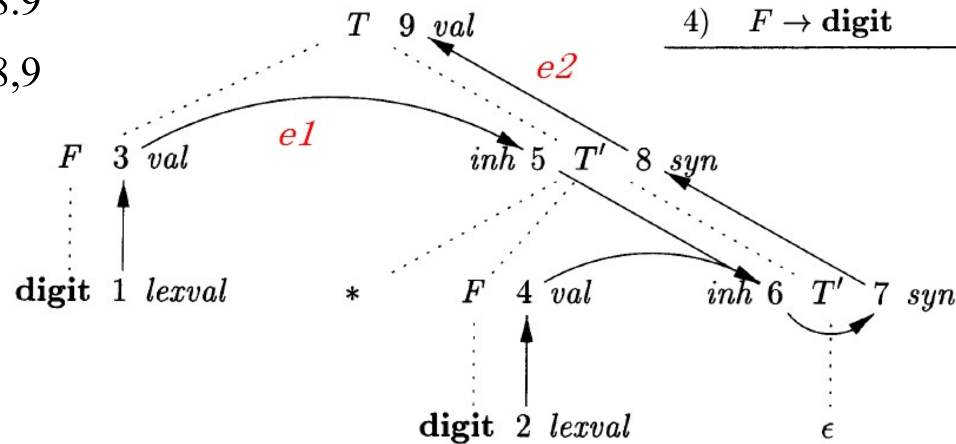
- **描述了某棵特定的分析树上各个属性实例之间的信息流（计算顺序）**
  - 从实例a_1到实例a_2的有向边表示计算a_2时需要a_1的值（必须先计算a_2，再计算a_1）
- **对于标号为X的分析树结点N，和X关联的每个属性a都对应依赖图的一个结点N.a**
- **结点N对应的产生式的语义规则通过X.c计算了A.b的值，且在分析树中X和A分别对应于N_1和N_2，那么从N_1.c到N_2.b有一条边**

依赖图的例子1

3*2的注释分析树；

- T → FT’ {T.val = T’.syn; T’.inh = F.val;}
  - 边e1、e2
- **可能的计算顺序：**
  - 1,2,3,4,5,6,7,8.9
  - 1,3,5,2,4,6,7,8,9

	产生式	语义规则
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \textbf{digit}$	$F.val = \textbf{digit}.lexval$



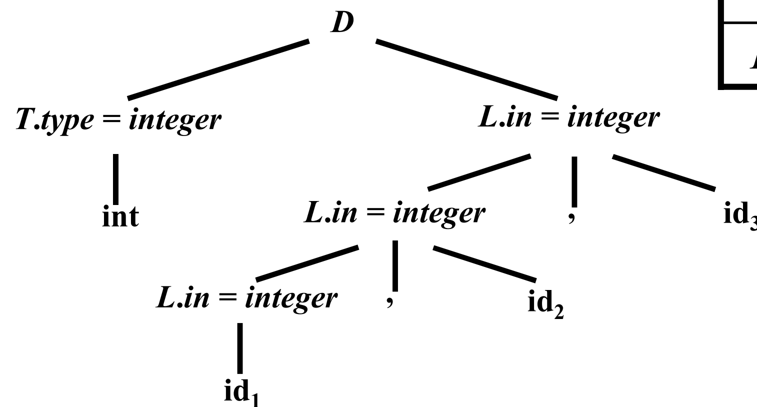
依赖图的例子2

再看一个例子（同样含有继承属性）：

int id$_1$, id$_2$, id$_3$

不可能像像综合属性那样自下而上标注属性

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



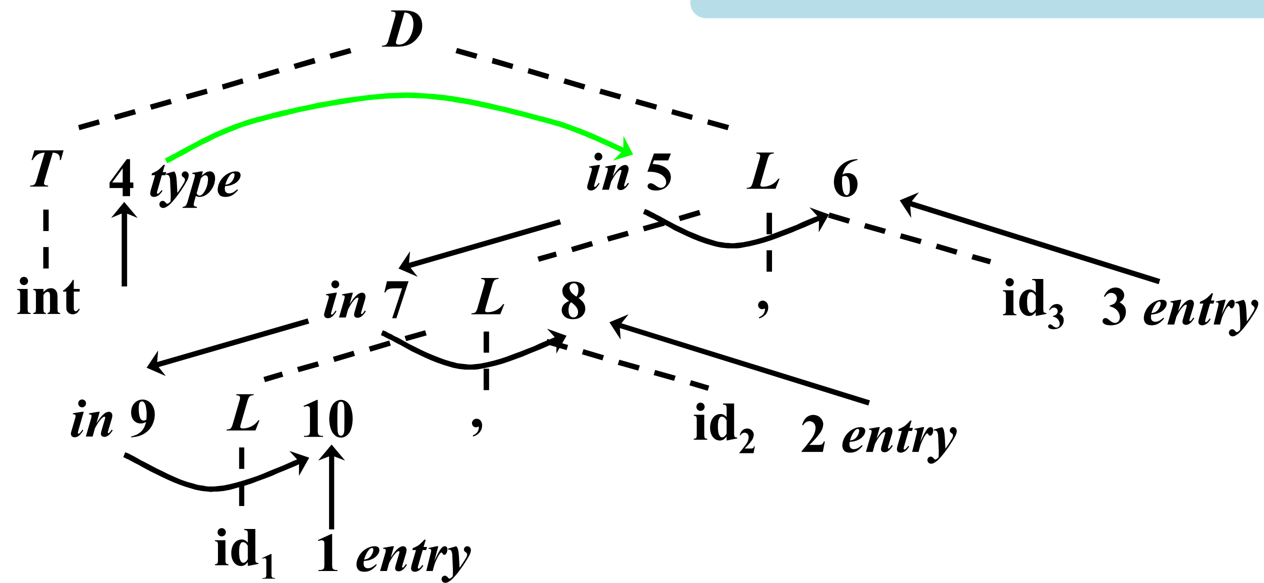
对于一个属性，不允许出现即是综合又是继承的现象

依赖图的例子2

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

  $D \rightarrow TL \quad L.in = T.type$

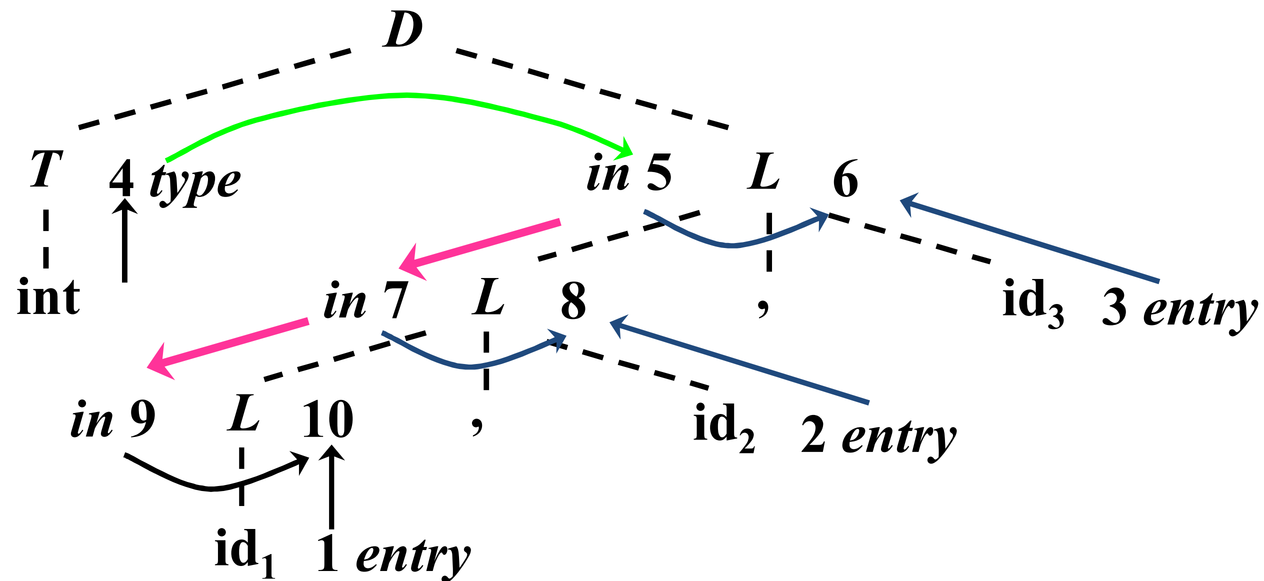
10个结点，每个属性一个结点



依赖图的例子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

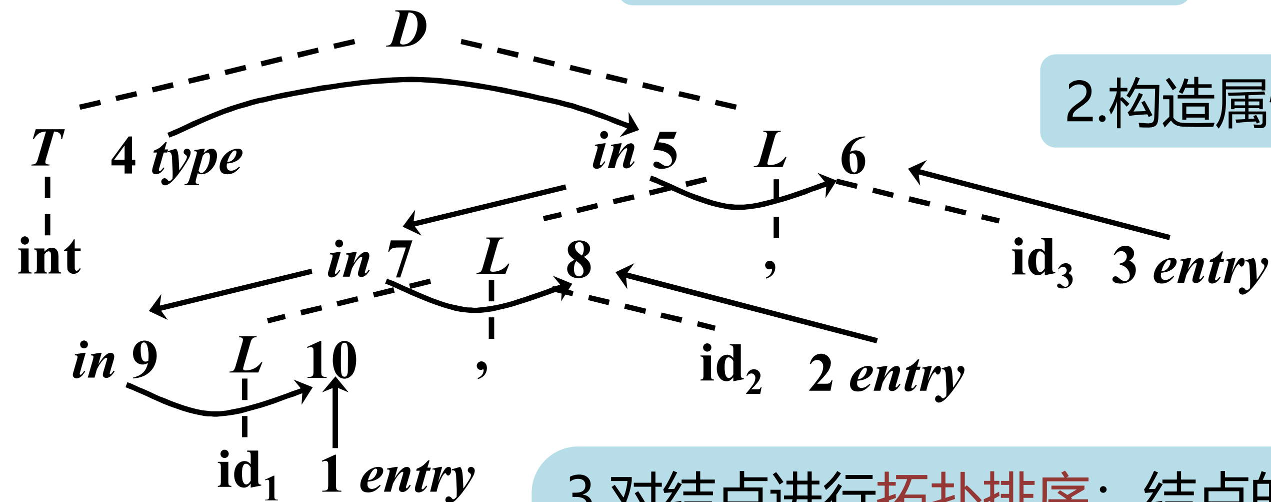
- **使用依赖图来表示计算顺序 ---拓扑顺序**
  - 显然，这些值的计算顺序应该形成一个偏序关系。如果依赖图中出现了环，表示属性值无法计算
- **给定一个SDD，很难判定是否存在一棵分析树，其对应的依赖图包含环**

拓扑顺序例子 (依赖图例2-含有继承属性)

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

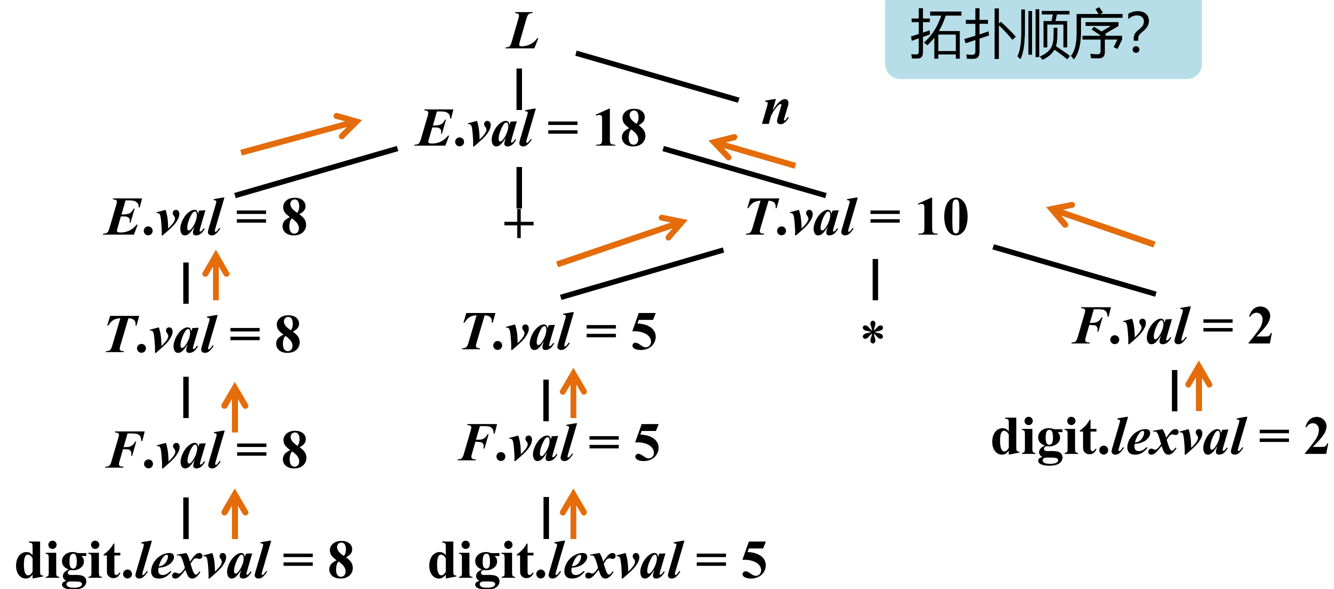


3.对结点进行拓扑排序：结点的一种排序，使得边只会从该次序中先出现的结点到后出现的结点，例：1，2，3，4，5，6，7，8，9，10 --- 不一定唯一

4.按拓扑排序的次序计算属性

拓扑顺序例子 (S属性SDD)

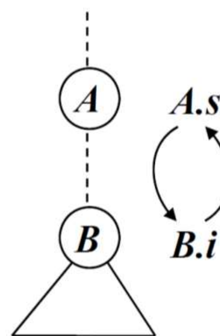
8+5∗2 n的计算顺序



两类SDD

- 对于只具有综合属性的*SDD*，可以按照任何自底向上的顺序计算它们的值
- 对于同时具有继承属性和综合属性的*SDD*，不能保证存在一个顺序来对各个节点上的属性进行求值
  - 例

产生式	语义规则
$A \rightarrow B$	$A.s = B.i$ $B.i = A.s + 1$



如果图中没有环，那么至少存在一个拓扑排序

两类SDD

- **从计算的角度看，给定一个*SDD*，很难确定是否存在某棵语法分析树，使得*SDD*的属性之间存在循环依赖关系**
- **但是存在*SDD*的有用子类，它们能够保证对每棵语法分析树都存在一个求值顺序，因为它们不允许产生带有环的依赖图: 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**

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RHS中的属性值来计算LHS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语法分析相容）**
  - 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

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SDD属于L属性SDD)
  - 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

每个*S*-属性定义都是*L*-属性定义

- **直观含义**
  - 在一个产生式所关联的各属性之间，依赖图的边可以从左到右，但不能从右到左 (因此称为L属性的，L是Left的首字母)
- **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**
- **与递归下降法相容**

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
$A \rightarrow B\ C$	$A.s = B.b;$ $B.i = f(C.c, A.s)$

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计算另一个属性值**
  - 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

- $L \rightarrow E\ n$　　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)	$T \rightarrow \textbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \textbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1\ ,\ \textbf{id}$	$L_1.inh = L.inh$ $addType(\textbf{id}.entry, L.inh)$
5)	$L \rightarrow \textbf{id}$	$addType(\textbf{id}.entry, L.inh)$

6.3 SDD的应用

语法制导翻译的应用

- **SDD的语义规则用来实现语义计算，目的根据实际需要而定**
- **上面例子中已经介绍了一些SDD的应用**
  - 计算值
  - 变量类型声明 (在符号表里填写内容addType)
  - 还能做什么?

语法制导翻译的应用

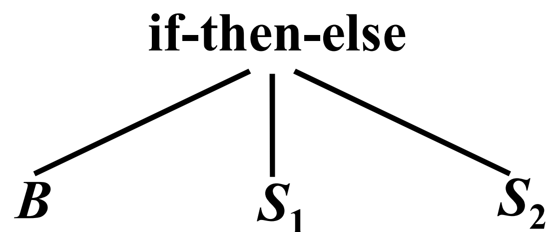
- **语法制导翻译技术主要应用**
  - 中间代码生成
    - 抽象语法树---一种中间代码表示形式
    - 三地址代码
    - 后缀、前缀表达式
  - 类型检查

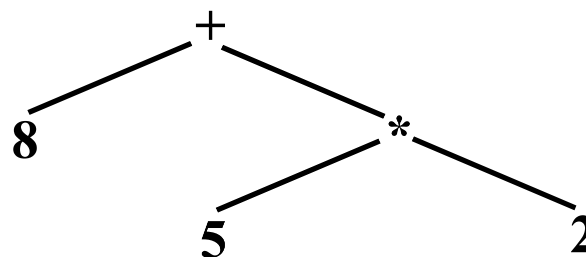
构造AST

■ IR：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

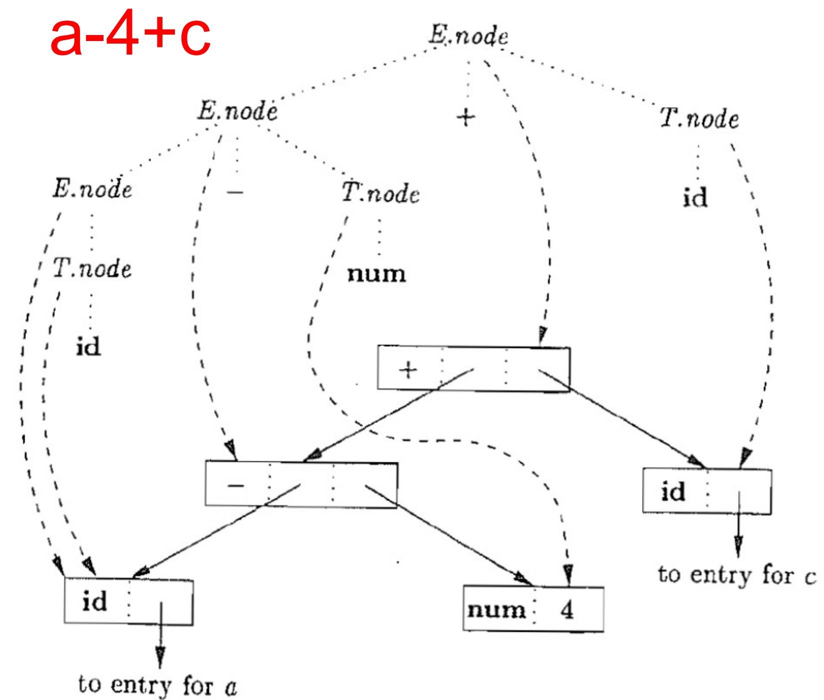
构造AST (S属性SDD)

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
1)	$E \rightarrow E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf($**id**, **id**$.entry)$
6)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf($**num**, **num**$.val)$

图 5-10　为简单表达式构造语法树

a-4+c

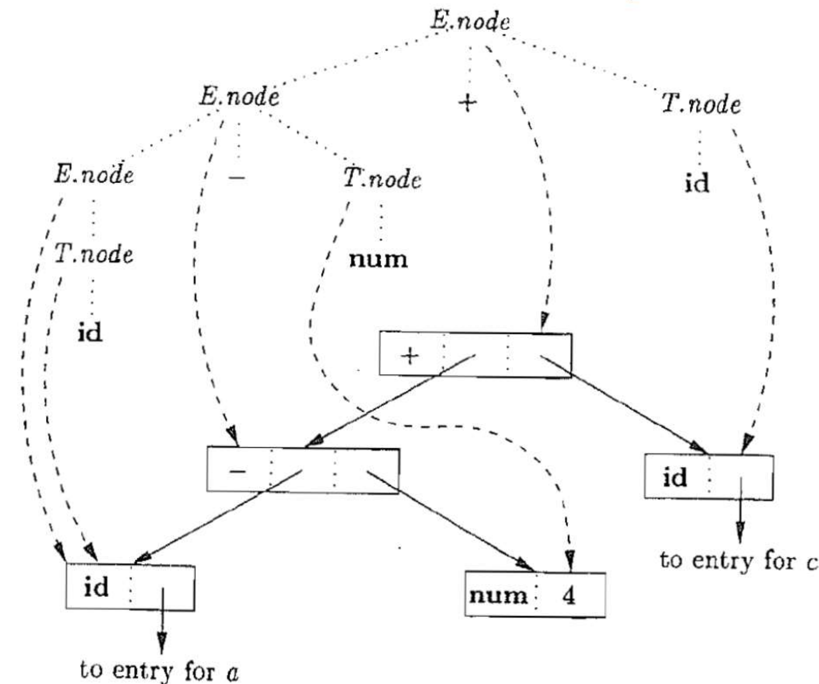


构造AST (S属性SDD)

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);



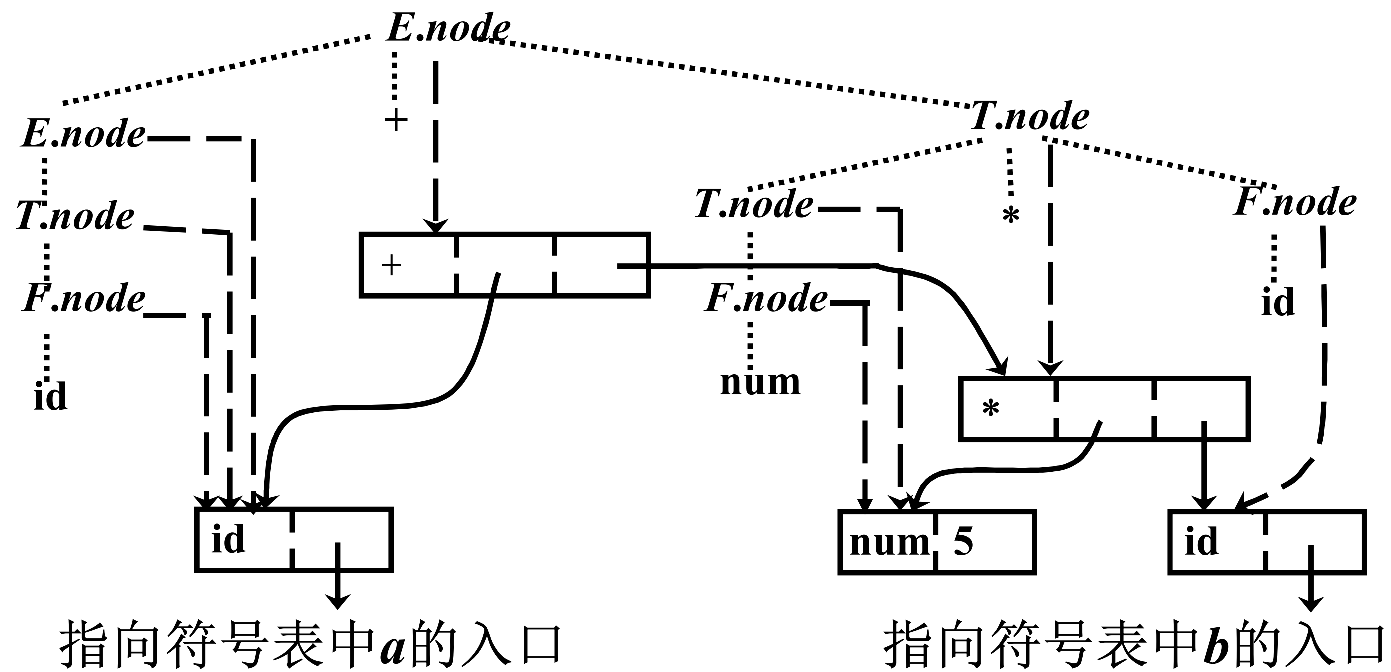
构造AST (S属性SDD)

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
$E \rightarrow E_1 + T$	$E.node = new\ Node($ '+', $E_1.node, T.node)$
$E \rightarrow T$	$E.node = T.node$
$T \rightarrow T_1 * F$	$T.node = new\ Node($ '*', $T_1.node, F.node)$
$T \rightarrow F$	$T.node = F.node$
$F \rightarrow (E)$	$F.node = E.node$
$F \rightarrow \mathbf{id}$	$F.node = new\ Leaf(\mathbf{id}, \mathbf{id}.entry)$
$F \rightarrow \mathbf{num}$	$F.node = new\ Leaf(\mathbf{num}, \mathbf{num}.val)$

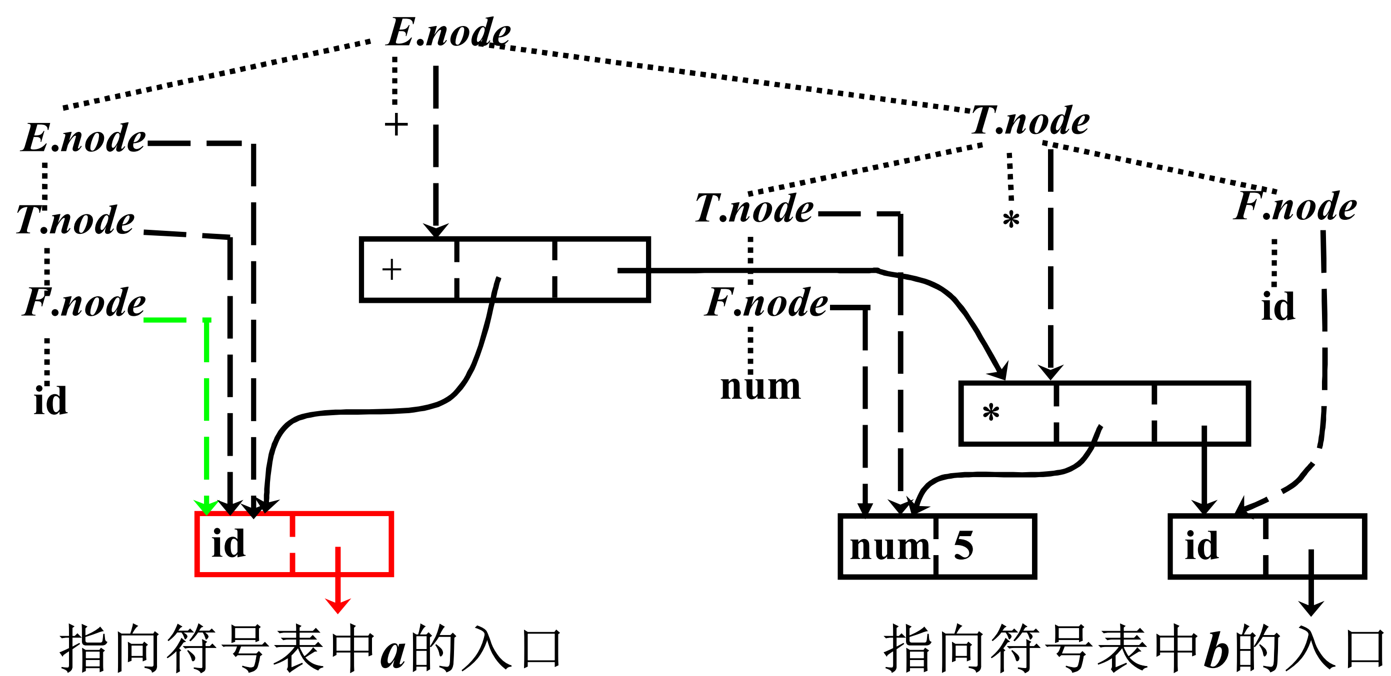
构造AST (S属性SDD)

a+5∗*b*的语法树的构造



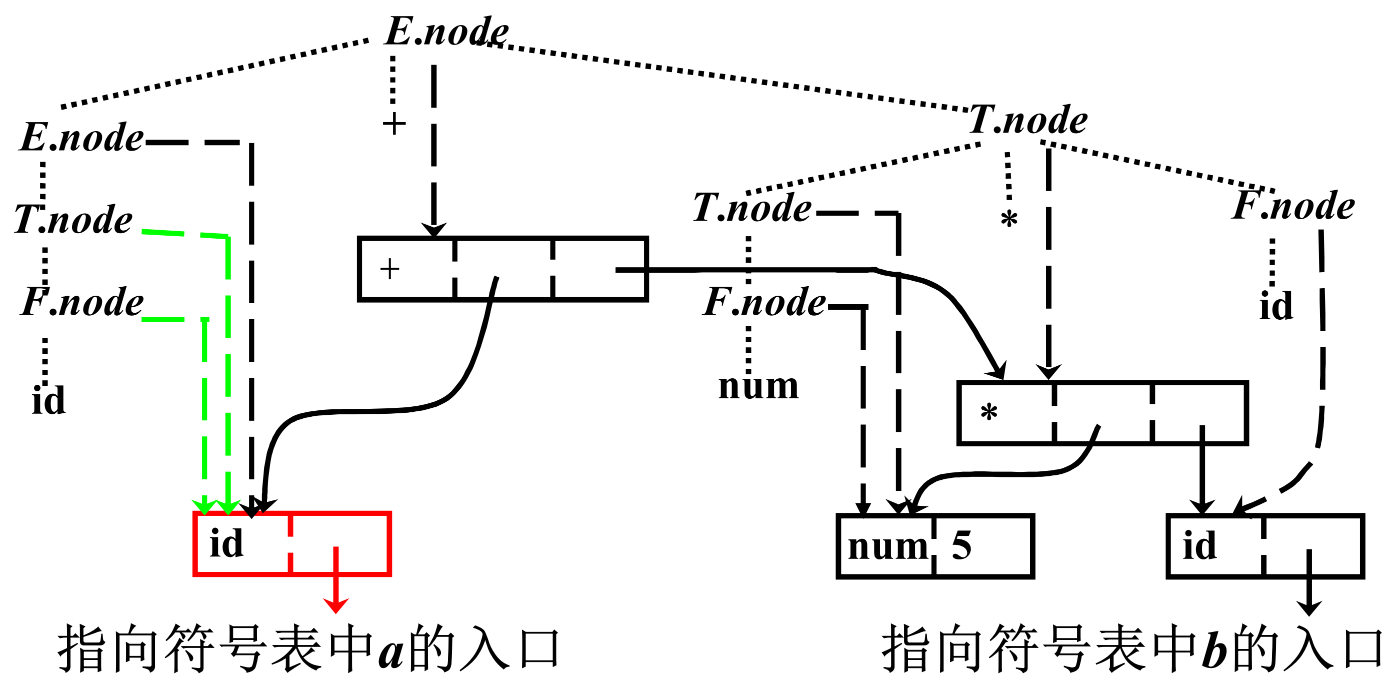
构造AST (S属性SDD)

a+5∗*b*的语法树的构造



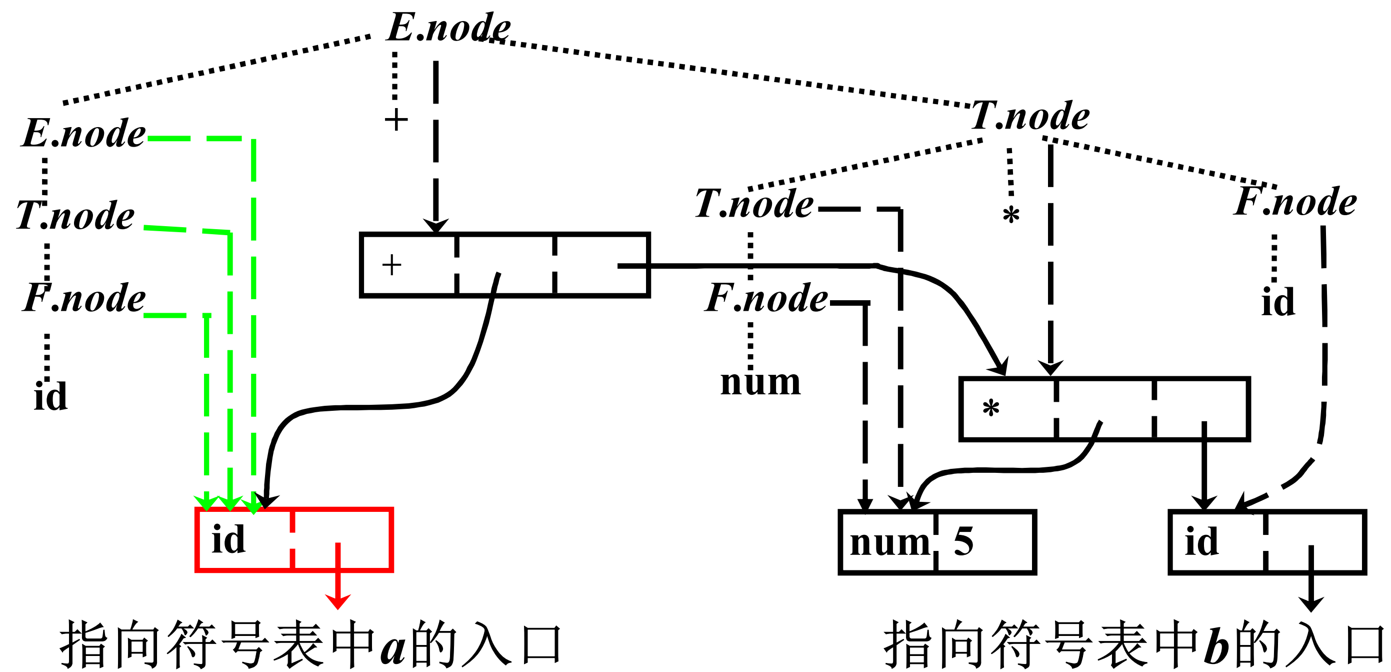
构造AST (S属性SDD)

a+5∗*b*的语法树的构造



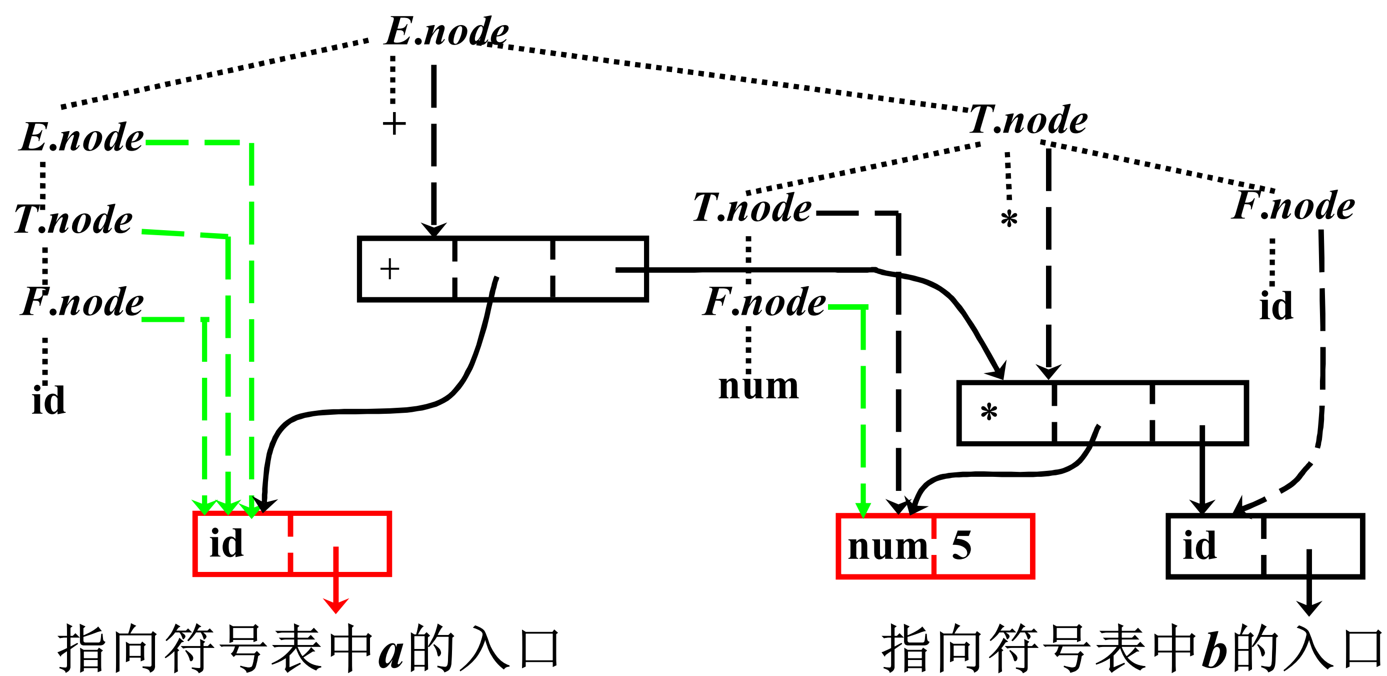
构造AST (S属性SDD)

a+5∗b的语法树的构造



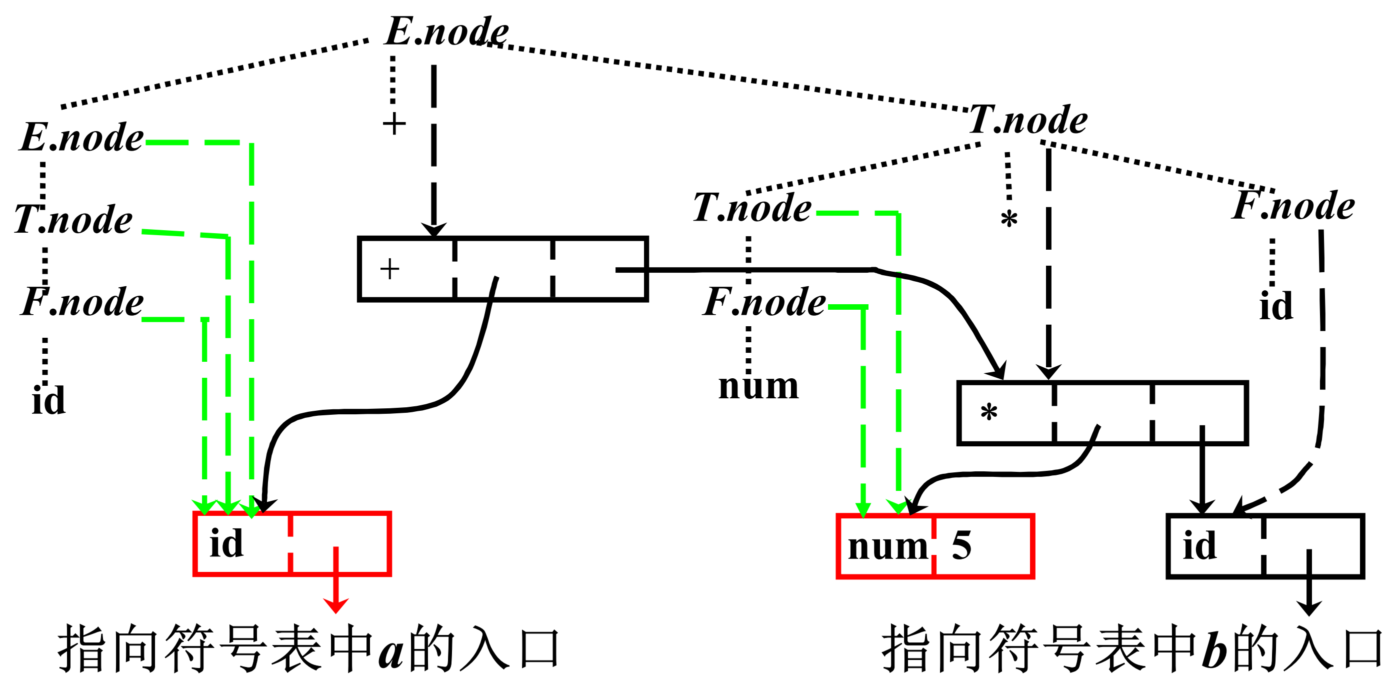
构造AST (S属性SDD)

a+5∗*b*的语法树的构造



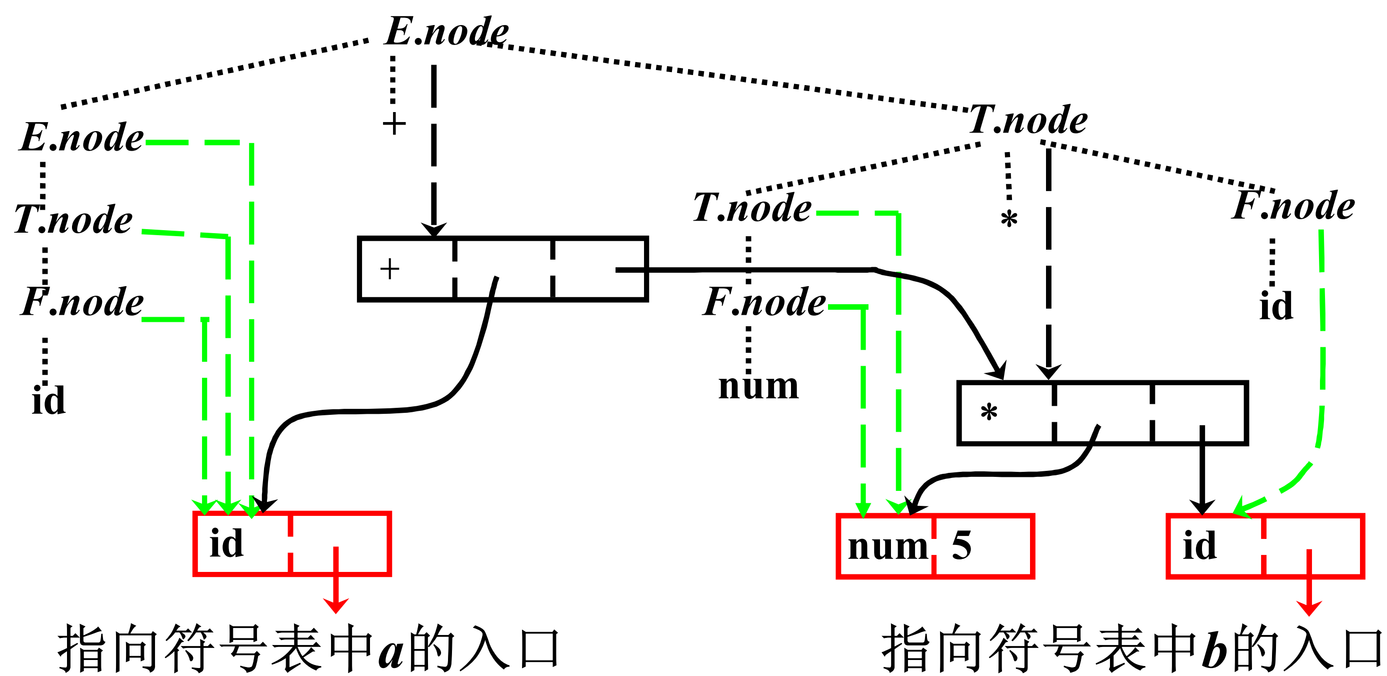
构造AST (S属性SDD)

$a+5*b$的语法树的构造



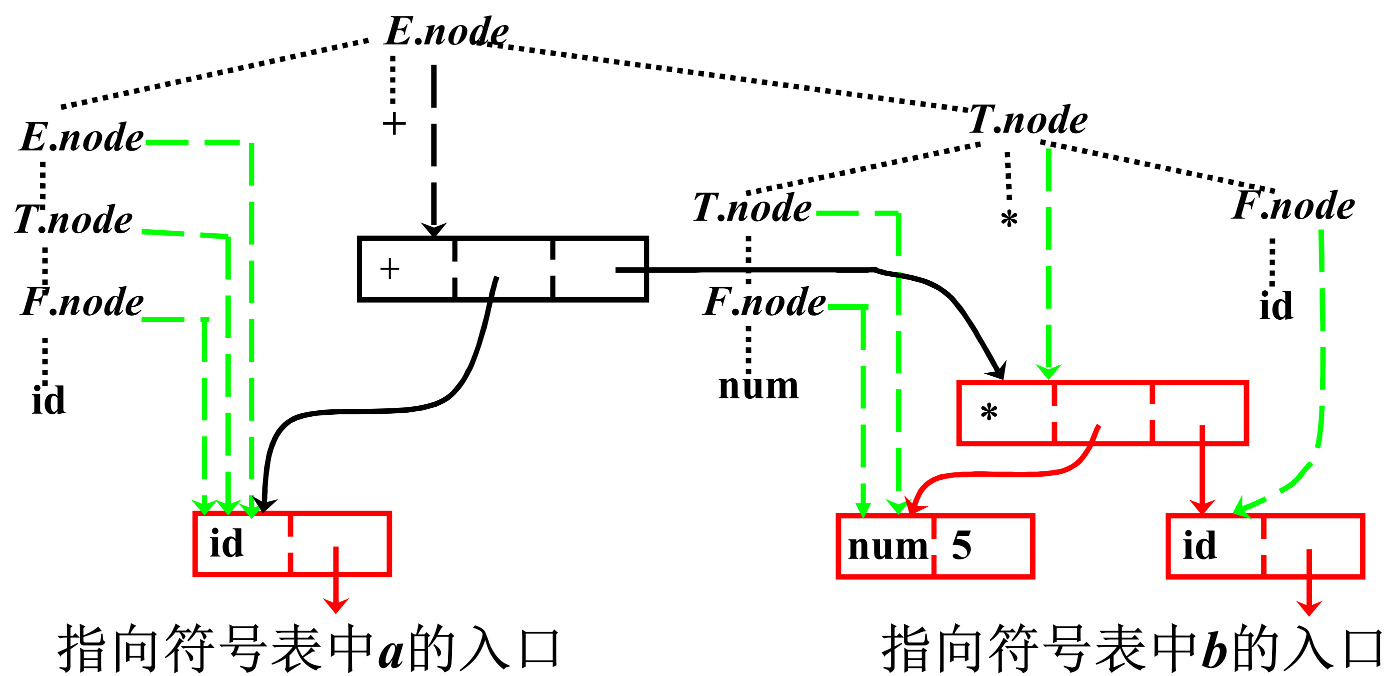
构造AST (S属性SDD)

a+5∗*b*的语法树的构造



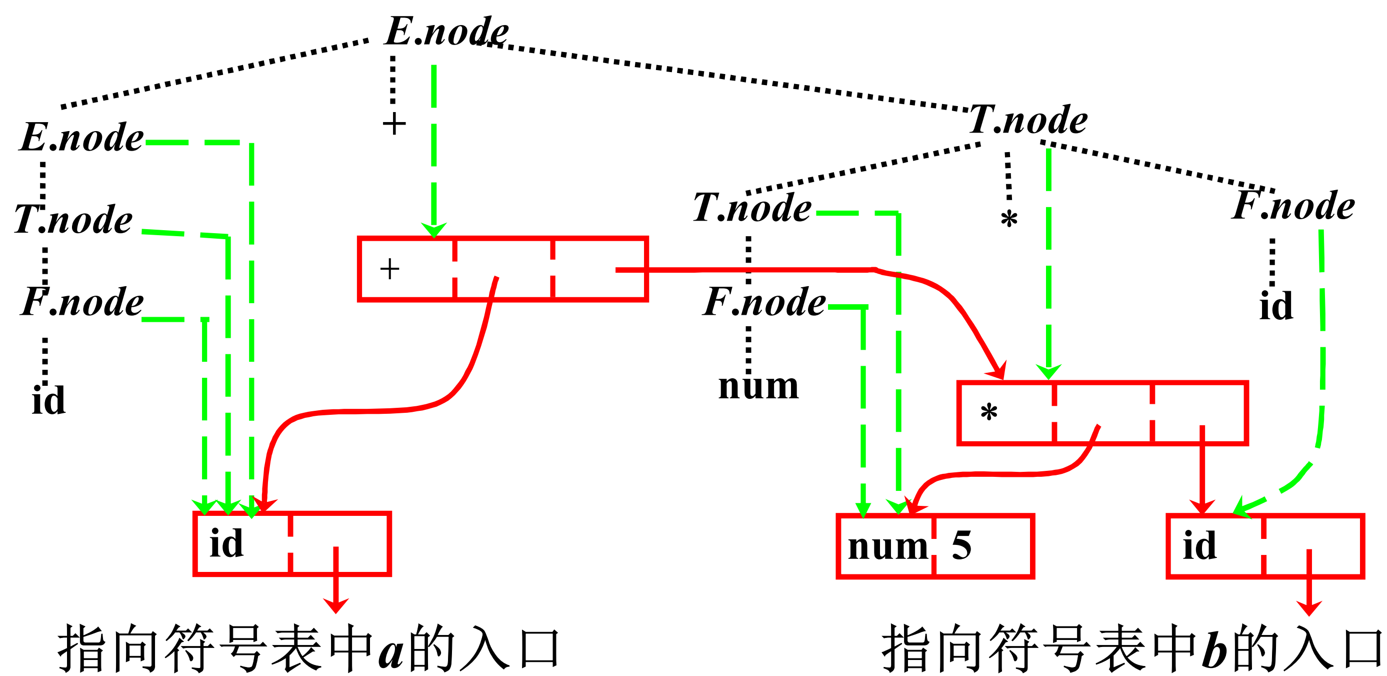
构造AST (S属性SDD)

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
1) $E \to E_1 + T$	$E.node = \mathbf{new}\ Node('+', E_1.node, T.node)$
2) $E \to E_1 - T$	$E.node = \mathbf{new}\ Node('-', E_1.node, T.node)$
3) $E \to T$	$E.node = T.node$
4) $T \to (E)$	$T.node = E.node$
5) $T \to \mathbf{id}$	$T.node = \mathbf{new}\ Leaf(\mathbf{id}, \mathbf{id}.entry)$
6) $T \to \mathbf{num}$	$T.node = \mathbf{new}\ Leaf(\mathbf{num}, \mathbf{num}.val)$

消除左递归

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

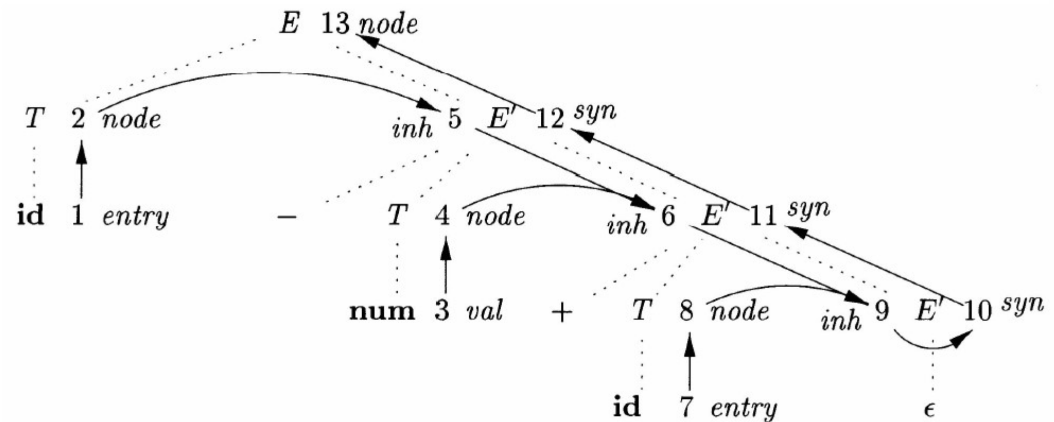
构造AST (L属性SDD)

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

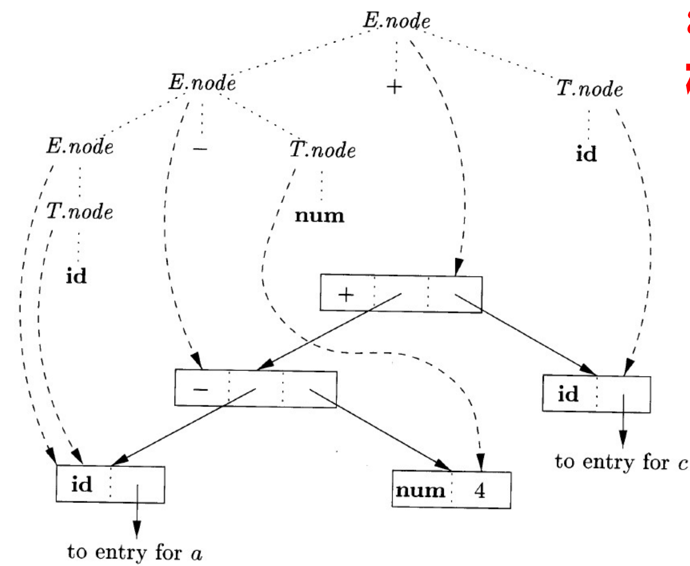
构造依赖图

a-4+c

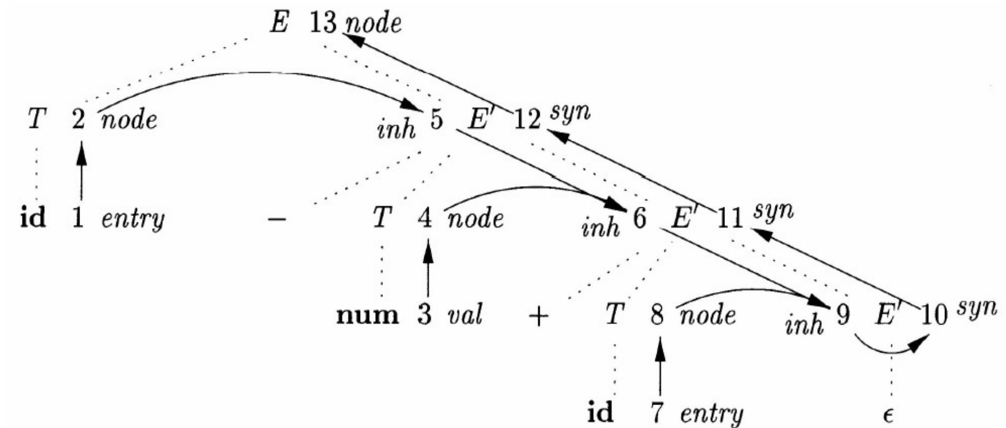


构造AST (L属性SDD)

例3：L属性SDD，构造语法树的语法制导定义（p205, 5.12)

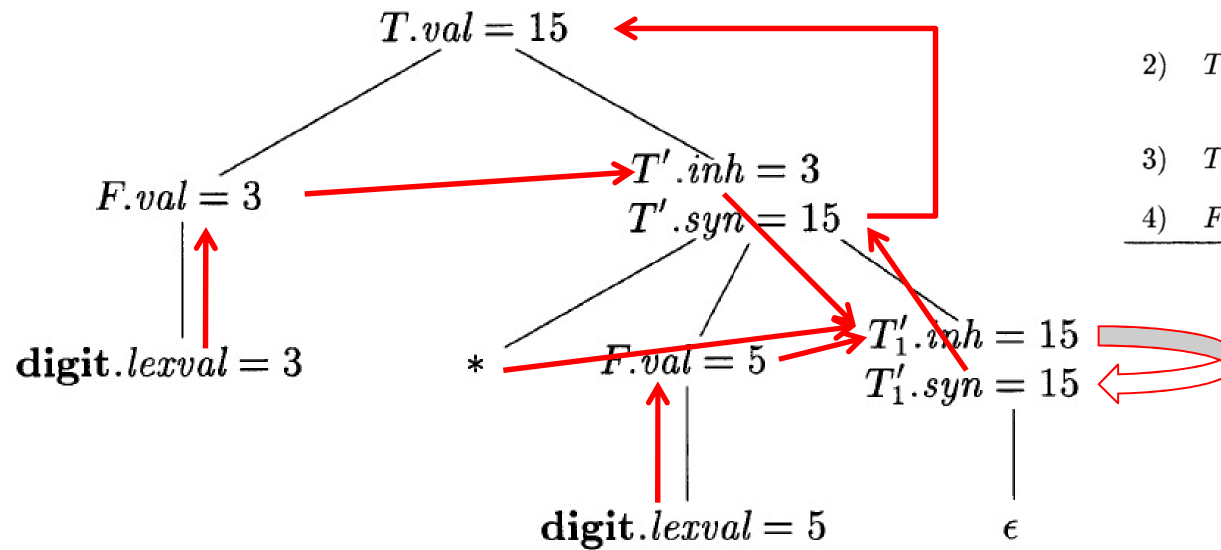


a-4+c：不同的语法分析树与抽象语法树



构造AST (L属性SDD)

- **和“3*5”例子一致**



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

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
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow [$ **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

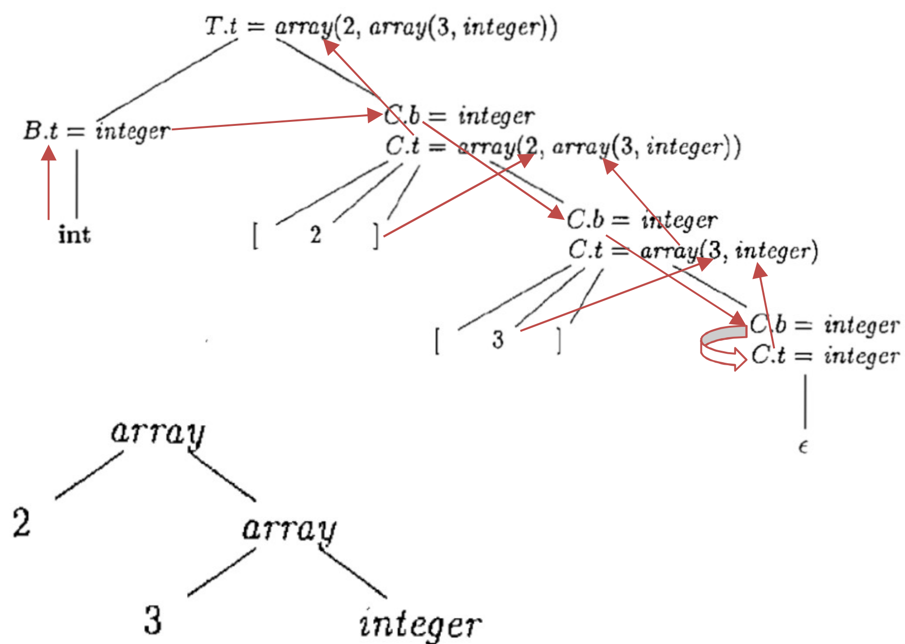
分量形如[3][4]：表示3X4的二维数组，e.g. int [3][4]

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**	$B.t = float$
$C \rightarrow [\ \mathbf{num}\ ]\ C_1$	$C.t = array\,(\mathbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$



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

作业

- 教材P198：5.1.2, 5.1.3 (表达式2)
- 教材P203：5.2.4
- 教材P207：5.3.1

6.4 递归下降过程中进行翻译

SDD的实现方法

- **先分析，后翻译---通用方法，适用于所有类型SDD**
  - SDD: 构造注释语法树（通用方法）
    - 建立语法分析树→添加注释成为注释语法分析树→画出依赖图→找出其中的拓扑排序(无环)，按照拓扑排序完成计算语法翻译
  - SDT: 添加语义动作虚拟结点（通用方法）
    - 建立语法分析树→将SDT语义动作作为虚拟结点添加至相应位置→前序遍历语法树，执行语义动作
    - 需要按照位置要求把SDD转换成SDT

SDD的实现方法

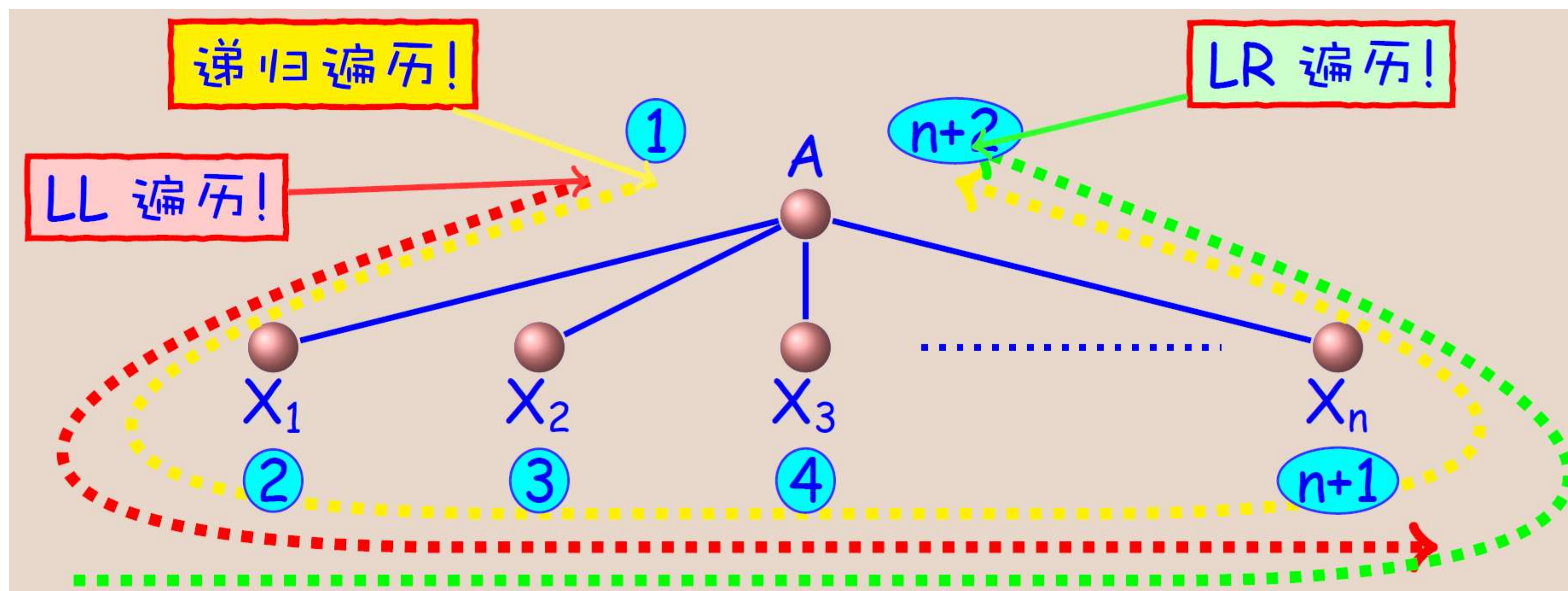
- 分析与翻译同步---非通用方法
  - 怎样的SDD可以和语法分析结合，实现同步？
    - 属性的计算次序一定受分析方法所限定的分析树结点建立次序的限制
      - 在对SDD的求值过程中，如果结点N的属性a依赖于结点M_1的属性a_1，M_2的属性a_2，…。那么我们必须先计算出M_i的属性，才能计算N的属性a
    - 分析树的结点是自左向右生成
    - 如果属性信息是自左向右流动，那么就有可能在分析的同时完成属性计算

SDD的实现方法

- **分析与翻译同步---非通用方法**
  - 三种语法分析树的遍历顺序（对应了三种语法分析方法）

- 递归下降分析法是对语法树的*递归遍历*，继承属性可以通过参数传递，而综合属性可以通过函数的返回值传递。
- LL 分析法对语法树*前序遍历*，但是对每个子树遍历后不再返回到树根，因此只能求解部分继承属性。
- LR 分析法对语法树*后序遍历*，由左边树叶开始索根，只能求解综合属性。

SDD的实现方法

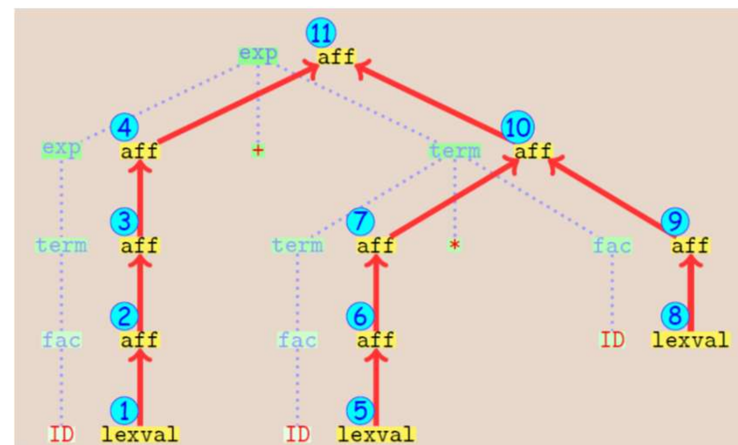


SDD的实现方法

- **分析与翻译同步---非通用方法**
  - 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

SDD的实现方法

- 分析与翻译同步---非通用方法
  - S属性SDD
    - LR分析：从注释语法树来看计算顺序和LR分析天然相容



SDD的实现方法

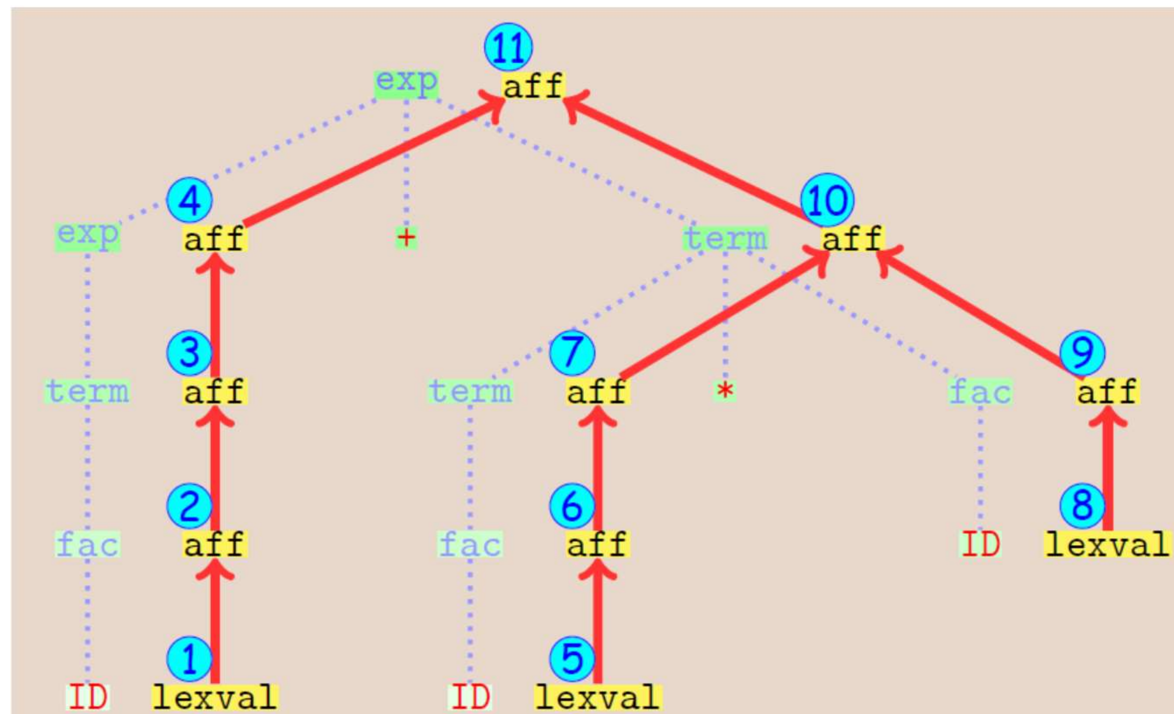
- **分析与翻译同步---非通用方法**
  - 边分析边翻译的方式能否用于继承属性?
    - 分析树的结点是自左向右生成；如果属性信息是自左向右流动，那么就有可能在分析的同时完成属性计算
  - L属性SDD
    - 递归下降分析+L属性SDD
    - LL分析 + L属性SDD? (后面讲)
    - LR分析 + L属性SDD (后面讲)

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RHS中的属性值来计算LHS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语法分析相容）**
  - 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**



S属性的SDD及其相容分析方法

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

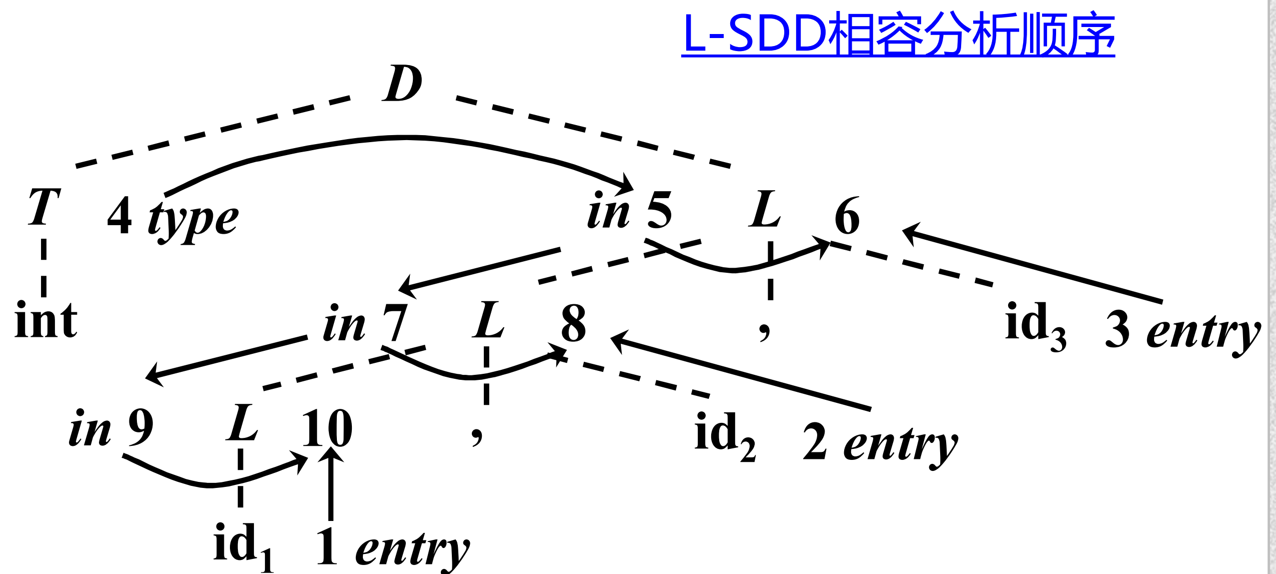
- **每个属性**
  - 要么是综合属性(S属性SDD属于L属性SDD)
  - 要么是继承属性，且产生式$A \rightarrow X_1X_2 \ldots X_n$中计算$X_i.a$的规则只能使用：
    - A的继承属性
    - X_i左边的文法符号X_j的继承属性或综合属性
    - X_i自身的继承或综合属性，且这些属性之间的依赖关系不形成环
- **依赖图的边**
  - 继承属性从左到右，从上到下
  - 综合属性从下到上
- **在扫描过程中计算一个属性值时，和它相关的依赖属性都已经计算完毕**

L属性的SDD及其相容分析方法

例如：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*(id.*entry*, *L.in*)
$L \rightarrow$ id	*addType*(id.*entry*, *L.in*)

L-SDD相容分析顺序



作业

- 文法G定义为

$$T \rightarrow a[L] \mid a$$
$$L \rightarrow L\ L \mid T$$

其中: ‘a’, ‘[’, 和 ‘]’ 为终结符, T 和 L 是非终结符, T 是文法开始符号.

该文法所生成语句是树结构的前序遍历序列. 现需将该序列转换为后序遍历序列，如“a[b c[d e[f]] g]”转换为“[b [d [f]e]c g]a”. 为此设计以下属性：a.lexeme 为终结符a 所对应的词素(形); T .postorder 和L.postorder记录T 和L 所表示的语法成分所对应的后序遍历序列.

(1) 试为上述属性设计语法制导定义SDD；

(2) 试画出语句“w[x[y] z]”的注释(附注) 语法树.

6.5 语法制导的翻译方案SDT（简要介绍）

语法制导的翻译方案SDT

- **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

语法制导的翻译方案SDT

- *SDT*是在产生式右部嵌入了程序片段的*CFG*，这些程序片段称为语义动作。按照惯例，语义动作放在花括号内
  - 例

$$D \rightarrow T \{ L.inh = T.type \} L$$
$$T \rightarrow \textbf{int} \{ T.type = int \}$$
$$T \rightarrow \textbf{real} \{ T.type = real \}$$
$$L \rightarrow \{ L_1.inh = L.inh \} L_1, \textbf{id}$$
$$\ldots$$

一个语义动作在产生式中的位置决定了这个动作的执行时间

语法制导的翻译方案SDT

- ***SDD***
  - 是关于语言翻译的高层次规格说明
  - 隐蔽了许多具体实现细节，使用户不必显式地说明翻译发生的顺序
- ***SDT***
  - 可以看作是对*SDD*的一种补充，是*SDD*的具体实施方案
  - 显式地指明了语义规则的计算顺序，以便说明某些实现细节

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)

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

■ 将S-SDD转换为(后缀) SDT

S-SDD

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.$lexval$

SDT

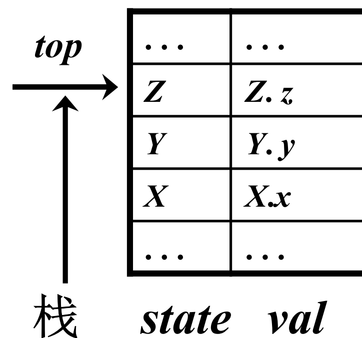
(1) $L \to E$ n { $L.val = E.val$ }
(2) $E \to E_1 + T$ { $E.val = E_1.val + T.val$ }
(3) $E \to T$ { $E.val = T.val$ }
(4) $T \to T_1 * F$ { $T.val = T_1.val \times F.val$ }
(5) $T \to F$ { $T.val = F.val$ }
(6) $F \to (E)$ { $F.val = E.val$ }
(7) $F \to$ digit { $F.val$ = digit.$lexval$ }

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

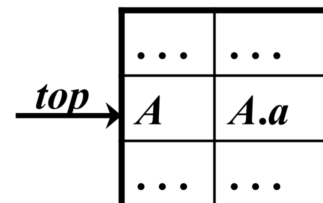
该SDD基本文法是LR的，并且SDD是S属性的，所以构造的后缀SDT可以和LR分析的规约步骤一起正确执行

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
$L \rightarrow E\ \mathbf{n}$	$\{\ \mathrm{print}(stack[top-1].val);\ top = top - 1;\ \}$
$E \rightarrow E_1 + T$	$\{\ stack[top-2].val = stack[top-2].val + stack[top].val;\ top = top - 2;\ \}$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$\{\ stack[top-2].val = stack[top-2].val \times stack[top].val;\ top = top - 2;\ \}$
$T \rightarrow F$	
$F \rightarrow (\ E\ )$	$\{\ stack[top-2].val = stack[top-1].val;\ top = top - 2;\ \}$
$F \rightarrow \mathbf{digit}$	

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 –

$E \rightarrow T\ R$

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

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

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

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

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

$\ldots\ \{print(8)\}\{print(5)\}\{print(+)\}\text{addop}\ T\{print(-)\}R$

$\ldots\ \{print(8)\}\{print(5)\}\{print(+)\}\{print(2)\}\{print(-)\}$

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

判断SDT可否用特定分析技术实现

- **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

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

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

判断SDT可否用特定分析技术实现例子

- **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

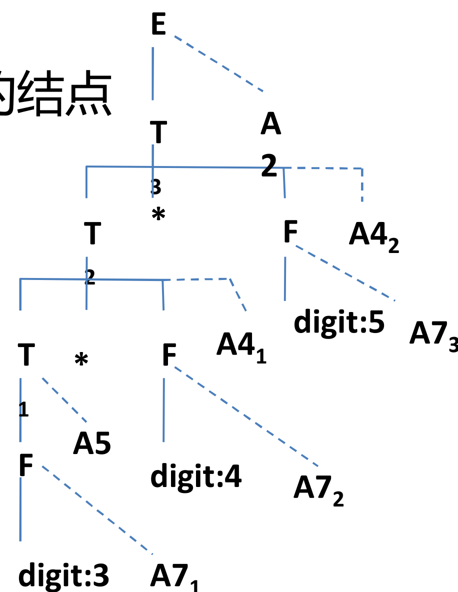
- **SDT的基本实现方法(不和语法分析同步进行)**
  - 忽略语义动作，对输入进行语法分析，产生一个语法分析树
  - 检查每个内部结点N，如果产生式为A → α，将α中各个动作当做N的附加子结点加入，使得N的子结点从左到右和α中符号及动作完全一致
  - 对分析树进行前序遍历，在访问虚拟结点时执行相应动作

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

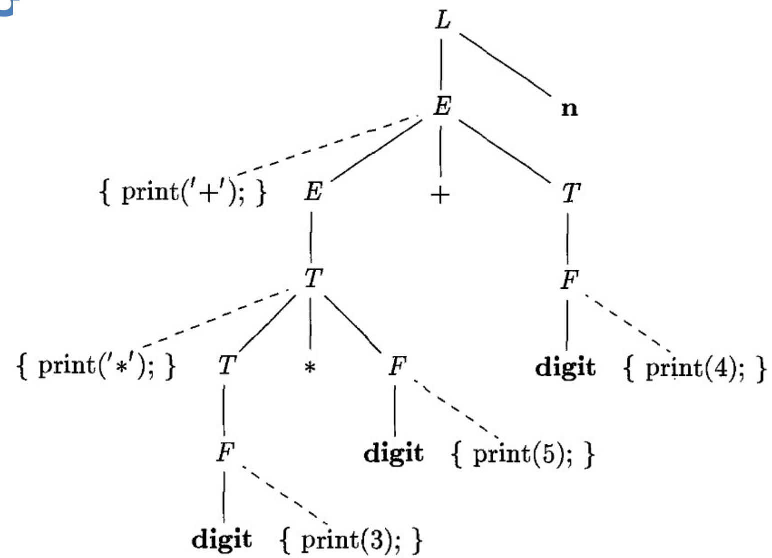
L	$\rightarrow$	E **n**	$\{ \text{print}(E.val); \}$ A1
E	$\rightarrow$	$E_1 + T$	$\{ E.val = E_1.val + T.val; \}$ A2
E	$\rightarrow$	T	$\{ E.val = T.val; \}$ A3
T	$\rightarrow$	$T_1 * F$	$\{ T.val = T_1.val \times F.val; \}$ A4
T	$\rightarrow$	F	$\{ T.val = F.val; \}$ A5
F	$\rightarrow$	(E)	$\{ F.val = E.val; \}$ A6
F	$\rightarrow$	**digit**	$\{ F.val = \textbf{digit}.lexval; \}$ 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**
  - $E \to E_1 + T$ {print('+');}
  - $E \to T$
- **转换后得到**
  - $E \to T\ R$
  - $R \to +\ T$ {print ('+');} R
  - $R \to \varepsilon$

A → Aα | β　　β(α)*

转换为：A → βA', A' → αA' | ε

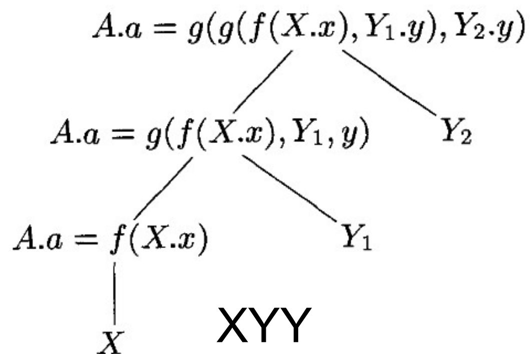
当一个SDD的动作有计算属性的值(而非仅仅打印输出)，我们必须小心处理。

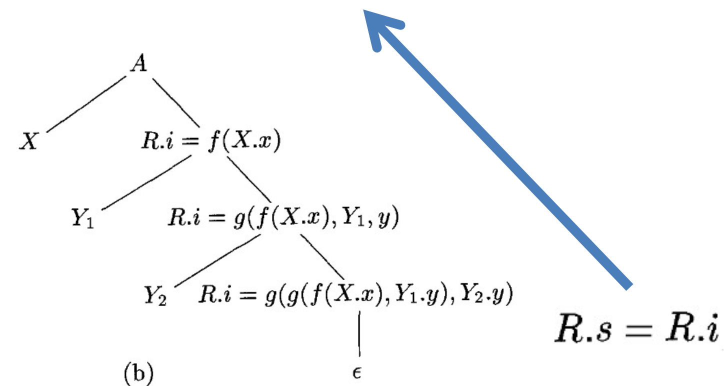
消除左递归时SDT的一般转换形式

- **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。**

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

- **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**





$R.s = R.i$

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

$E_1.val$

$$S \rightarrow B$$

$$B \rightarrow B_1\ B_2$$

$$B \rightarrow B_1 \text{ sub } B_2$$

$$B \rightarrow \text{text}$$

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

$E_1.val$

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

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）**

$S \rightarrow$ $\{B.ps = 10\}$
$\quad B \quad \{S.ht = B.ht\}$

$B \rightarrow$ $\{B_1.ps = B.ps\}$
$\quad B_1 \quad \{B_2.ps = B.ps\}$
$\quad B_2 \quad \{B.ht = max(B_1.ht, B_2.ht)\}$

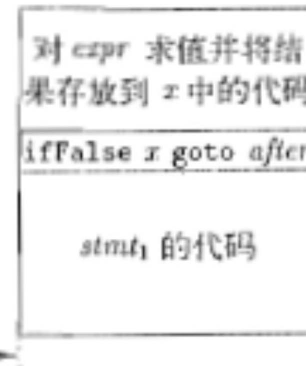
$B \rightarrow$ $\{B_1.ps = B.ps\}$
$\quad B_1$
$\quad \text{sub} \quad \{B_2.ps = shrink(B.ps)\}$
$\quad B_2 \quad \{B.ht = disp(B_1.ht, B_2.ht)\}$

$B \rightarrow \text{text}$ $\{B.ht = \text{text}.h \times B.ps\}$

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

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

$A \to M\ B\ C$

$M \to \{M.i = A.i;\ M.s = f(M.i);\}$

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

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**

$$
\begin{array}{lll}
S \rightarrow \textbf{while} \; (& \{ \; L1 = new(); \; L2 = new(); \; C.false = S.next; \; C.true = L2; \; \} \\
\quad\quad C \;) & \{ \; S_1.next = L1; \; \} \\
\quad\quad S_1 & \{ \; S.code = \textbf{label} \; \| \; L1 \; \| \; C.code \; \| \; \textbf{label} \; \| \; L2 \; \| \; S_1.code; \; \}
\end{array}
$$

6.6 L属性定义的递归下降实现

递归下降语法分析

■ 例子

P:
(1) Z → aBd   {a}
(2) B → d   {d}
(3) B → c   {c}
(4) B → bB   {b}

a   b   c   d

```
Z ( )
{
if (token == a)
 {  match(a);
    B( );
    match(d);
 }
else error();
}
```

```
B ( )
{
 case token of
  d:  match(d);break;
  c:  match(c); break;
  b:{  match(b);
       B( ); break;}
  other: error();
}
```

```
void main()
{read();
 Z();  }
```

递归下降语法分析

把分析器的构造方法推广到翻译方案的实现

产生式$R \to +TR \mid \varepsilon$ 的分析过程

```
void R( ) {
if (lookahead == '+' ) {
        match ( '+' ); T( ); R( );
}
else  /∗ 什么也不做 ∗/
}
```

*L*属性定义的递归下降实现（基本方法）

- **使用递归下降的语法分析器**
  - 每个非终结符号对应一个函数
  - 函数的参数接受继承属性
  - 返回值包含了综合属性
- **在函数体中**
  - 首先选择适当的产生式
  - 使用局部变量来保存属性
  - 对于产生式体中的终结符号，读入符号并获取其（经词法分析得到的）综合属性
  - 对于非终结符号，使用适当的方式调用相应函数，并记录返回值

L属性定义的递归下降实现（基本方法）

- **例1**

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

```
string S(label next) {
        string Scode, Ccode; /* 存放代码片段的局部变量 */
        label L1, L2; /* 局部标号*/
        if ( 当前输入== 词法单元while ) {
                读取输入；
                检查 '(' 是下一个输入符号，并读取输入;
                L1 = new();
                L2 = new();
                Ccode = C(next, L2);
                检查 ')' 是下一个输入符号，并读取输入；
                Scode = S(L1);
                return("label" || L1 || Ccode || "label" || L2 || Scode);
        }
        else /* 其他语句类型 */
}
```

L属性定义的递归下降实现（基本方法）

例2

$B \to B_1 B_2 \mid B_1 \text{ sub } B_2 \mid (B_1) \mid \text{text}$

$$
\begin{aligned}
S &\to B \\
B &\to T\ B_1 \mid T \\
T &\to F \text{ sub } T_1 \mid F \\
F &\to (\ B\ ) \mid \text{text}
\end{aligned}
$$

	产生式	语义动作
1)	$S \to$	$\{ B.ps = 10; \}$
	B	
2)	$B \to$	$\{ B_1.ps = B.ps; \}$
	B_1	$\{ B_2.ps = B.ps; \}$
	B_2	$\{ B.ht = \max(B_1.ht, B_2.ht);$
		$B.dp = \max(B_1.dp, B_2.dp); \}$
3)	$B \to$	$\{ B_1.ps = B.ps; \}$
	B_1 sub	$\{ B_2.ps = 0.7 \times B.ps; \}$
	B_2	$\{ B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps);$
		$B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps); \}$
4)	$B \to ($	$\{ B_1.ps = B.ps; \}$
	$B_1\ )$	$\{ B.ht = B_1.ht;$
		$B.dp = B_1.dp; \}$
5)	$B \to$ text	$\{ B.ht = getHt(B.ps, \text{text}.lexval);$
		$B.dp = getDp(B.ps, \text{text}.lexval); \}$

```
(float, float) T(float ps) {
    float h1, h2, d1, d2; /* 用于存放高度和深度的局部变量*/
    /* F(ps) 代码开始 */
    if (当前输入 == '(' ) {
        读取下一个输入;
        (h1, d1) = B(ps);
        if (当前输入 != ')' )语法错误: 期待 ')';
        读取下一个输入;
    }
    else if ( 当前输入 == text ) {
        令 t 等于词法值 text.lexval ;
        读取下一个输入;
        h1 = getHt(ps, t);
        d1 = getDp(ps, t);
    }
    else 语法错误：期待 text 或者 '(';
    /* F(ps) 代码结束 */
    if ( 当前输入 == sub ) {
        读取下一个输入;
        (h2, d2) = T(0.7 * ps);
        return (max(h1, h2 - 0.25 * ps), max(d1, d2 + 0.25 * ps));
    }
    return (h1, d1);
}
```

*L*属性定义的递归下降实现（边扫描边计算属性）

- **当属性值的体积很大时，对属性值进行运算的效率很低**
  - 比如code（代码）可能是一个上百K的串，对其进行并置等运算会比较低效
- **可逐步生成属性的各个部分，并增量式添加到最终的属性值中**
- **条件：**
  - 存在一个主属性，且主属性是综合属性
  - 在各产生式中，主属性是通过产生式体中各个非终结符号的主属性连接（并置）得到的，同时还会连接一些其它的元素
  - 各非终结符号的主属性的连接顺序和它在产生式体中的顺序相同

L属性定义的递归下降实现（边扫描边计算属性）

- **基本思想**
  - 只需要在适当的时候“发出”非主属性的元素，即把这些元素拼接到适当的地方
- **例**

```
string S(label next) {
        string Scode, Ccode; /* 存放代码片段的局部变量 */
        label L1, L2; /* 局部标号*/
        if ( 当前输入== 词法单元while ) {
                读取输入；
                检查 '(' 是下一个输入符号，并读取输入；
                L1 = new();
                L2 = new();
                Ccode = C(next, L2);
                检查 ')' 是下一个输入符号，并读取输入；
                Scode = S(L1);
                return("label" || L1 || Ccode || "label" || L2 || Scode);
        }
        else /* 其他语句类型 */
}
```

```
void S(label next) {
        label L1, L2; /* 局部标号 */
        if ( 当前输入 == 词法单元 while ) {
                读取输入；
                检查 '(' 是下一个输入符号，并读取输入；
                L1 = new();
                L2 = new();
                print("label", L1);
                C(next, L2);
                检查 ')' 是下一个输入符号，并读取输入；
                print("label", L2);
                S(L1);
        }
        else /* 其他语句类型 */
}
```

L属性定义的递归下降实现（边扫描边计算属性）

- **基本思想**
  - 只需要在适当的时候“发出”非主属性的元素，即把这些元素拼接到适当的地方
- **例**

$$
\begin{array}{lll}
S \rightarrow \textbf{while}\ (& \{ L1 = new();\ L2 = new();\ C.false = S.next;\ C.true = L2; \} \\
\quad C\ ) & \{ S_1.next = L1; \} \\
\quad S_1 & \{ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code; \}
\end{array}
$$

$$
\begin{array}{llll}
S & \rightarrow & \textbf{while}\ (& \{ L1 = new();\ L2 = new();\ C.false = S.next; \\
 & & & \quad C.true = L2;\ print(\texttt{"label"}, L1); \} \\
 & & C\ ) & \{ S_1.next = L1;\ print(\texttt{"label"}, L2); \} \\
 & & S_1 &
\end{array}
$$

L属性定义的递归下降实现（边扫描边计算属性）

- **例　左递归的消除引起继承属性**

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

L属性定义的递归下降实现

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

$T + T + T + \ldots$

$E \to T \quad \{R.i = T.nptr\}$
$\quad R \quad \{E.nptr = R.s\}$

$R \to +$
$\quad T \quad \{R_1.i = mkNode(\text{'+'}, R.i, T.nptr)\}$
$\quad R_1 \quad \{R.s = R_1.s\}$

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

$T \to F \quad \{W.i = F.nptr\}$
$\quad W \quad \{T.nptr = W.s\}$

$W \to *$
$\quad F \quad \{W_1.i = mkNode(\text{'*'}, W.i, F.nptr)\}$
$\quad W_1 \quad \{W.s = W_1.s\}$

$W \to \varepsilon \quad \{W.s = W.i\}$

F产生式部分不再给出

L属性定义的自上而下计算

```
syntaxTreeNode∗ R (syntaxTreeNode∗ i) {
    syntaxTreeNode ∗nptr, ∗i1, ∗s1, ∗s;
    char addoplexeme;

    if (lookahead == '+' ) { /∗ 产生式 R → +T R  ∗/
            addoplexeme = lexval;
            match('+' ); nptr = T();
            i1 = mkNode(addoplexeme, i , nptr);
            s1 = R (i1); s = s1;
    }
    else  s = i;  /∗ 产生式 R → ε  ∗/
    return s;
}
```

R : i, s

T : $nptr$

$+$: $addoplexeme$

6.7 L属性定义的LL分析实现

L属性定义的LL实现

难点在哪里?

- LL分析和递归下降异同：都是top-down，但递归下降有回溯，而LL是一直向下
- LL语法分析器实现了一个最左推导的过程。假设推导S =>* ωα 中：
  - ω 是已经匹配完成的输入串，而α的开头为A，A将以A →BC 产生式展开 (查表得)。那么，当前栈顶为A，下面将弹出A替换为BC (B为栈顶)
  - 如果C有继承属性C.i依赖于A的继承属性和B的属性，对于L属性文法，当计算到C时，A的继承属性和B的所有属性都应该可用了(思考why?)
  - 但是当分析到C的时候，A和B都早已不在栈内，其相关联的属性值也消失，因此无法计算C.i

L属性定义的LL实现

- **实现技巧**
  - 将所需的A的继承属性值拷贝到对C的继承属性求值的动作记录中
  - 将所需的B的继承/综合属性都临时拷贝到栈中B之下的一个记录中(top-1)，留待C.i计算的时候使用
- **为什么这种临时拷贝是可行的?**
  - 所有拷贝都发生在对某个非终结符号的一次展开时创造的不同文法符号记录之间，因为产生式可知，这些文法符号的相对位置也可知，所以他们在栈中的位置(相距于栈顶距离)也可以，即，可以正确寻址数据项，部分数据项是其他项 (文法符号)属性值的临时拷贝

L属性定义的LL实现

扩展语法栈

- 基础文法本身要是LL的，然后把SDD改写为内嵌的SDT
  - 计算X的继承属性的动作插入到产生式体中对应的X的左边；计算产生式头的综合属性的动作在产生式的最右边
- 扩展语法分析栈，存放语义动作，以及需要存储所有属性求值所需的某些数据项，部分数据项是其他项 (文法符号)属性值的临时拷贝

L属性定义的LL实现

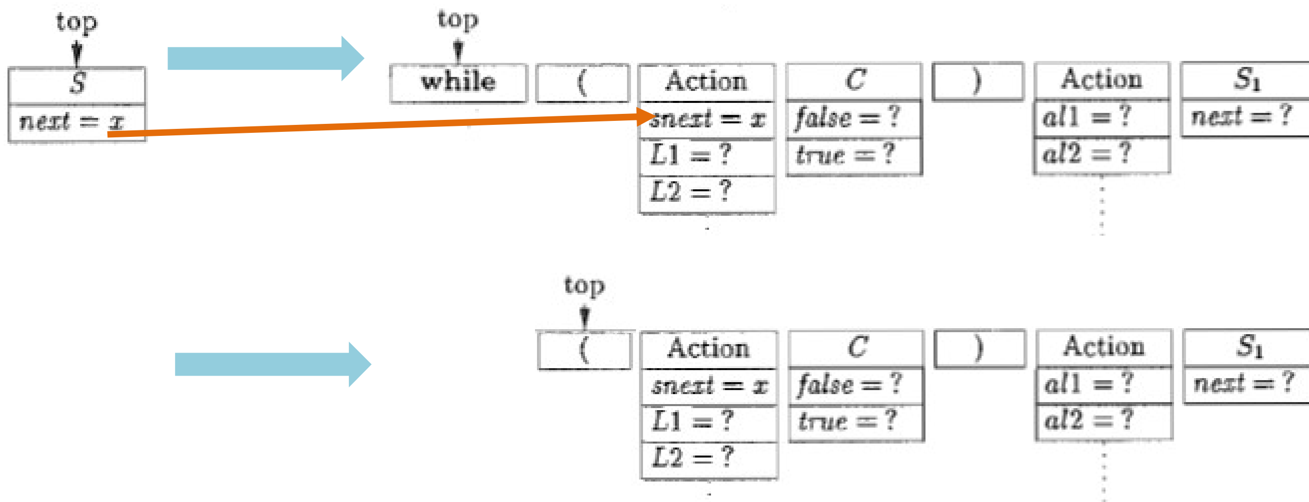
扩展语法栈

- 扩展语法分析栈，除了保存对应于文法符号 (终结符和非终结符)的item之外，还增加以下两类Item:
  - 动作记录（action-record）：即将被执行的语义动作
  - 综合记录（synthesize-record）：非终结符的综合属性值
- 这两类item在分析栈中的位置如下:
  - (非终结符X的继承属性位于它自身的记录item中)，而对这些属性求值的代码则使用仅靠在X的栈记录之上的动作记录项来表示
  - 非终结符X的综合属性放在单独的综合记录项中，在栈里面仅靠在X记录项之下
  - 与L属性文法内嵌式SDT的生成原则基本一致

L属性定义的LL实现

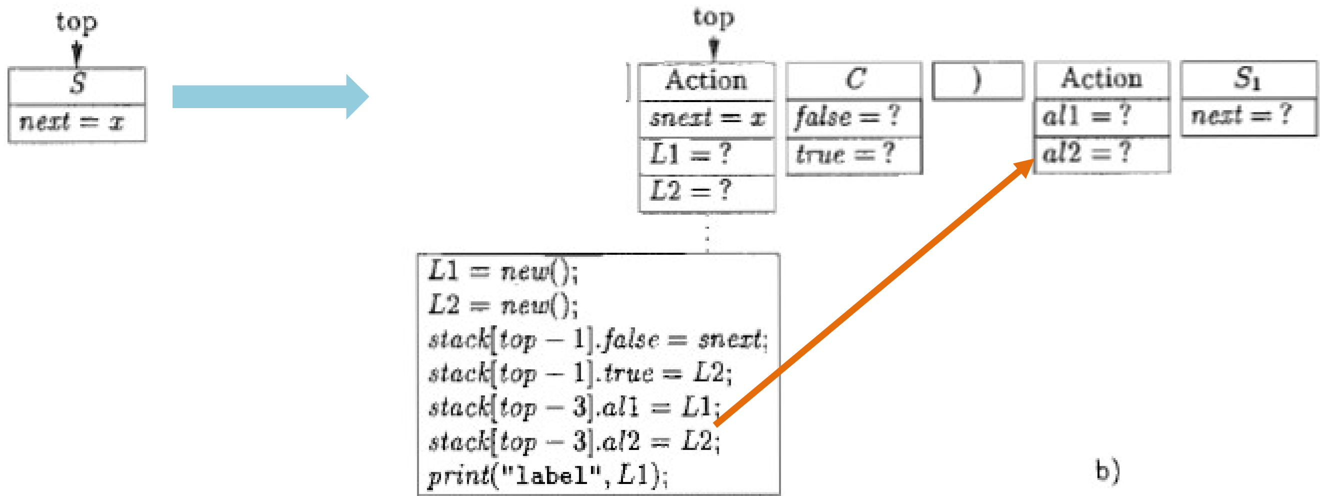
- 不断拷贝属性值，实现与LL分析同步的翻译
  - 例1

$$
\begin{array}{lll}
S & \rightarrow & \textbf{while}\ (\quad \{\ L1 = new();\ L2 = new();\ C.false = S.next; \\
 & & \qquad\quad C.true = L2;\ print(\texttt{"label"}, L1);\ \} \\
 & & C\ )\qquad \{\ S_1.next = L1;\ print(\texttt{"label"}, L2);\ \} \\
 & & S_1
\end{array}
$$



L属性定义的LL实现

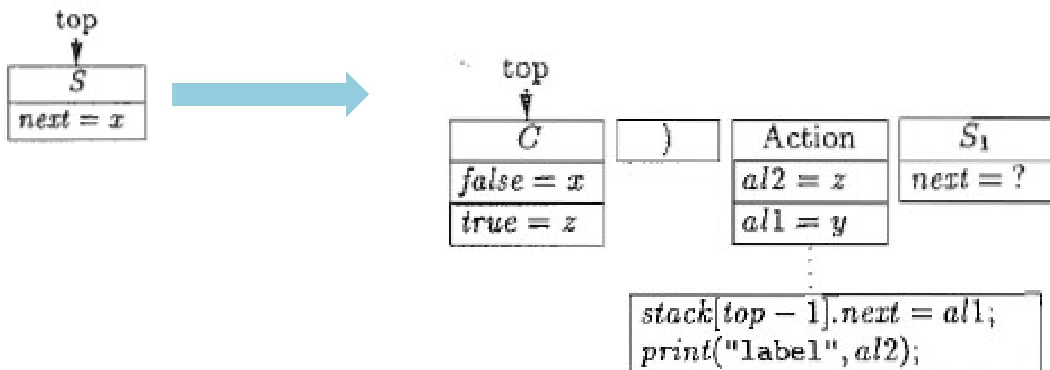
- 不断拷贝属性值，实现与LL分析同步的翻译
  - 例1



L属性定义的LL实现

- **不断拷贝属性值，实现与LL分析同步的翻译**
  - 例1

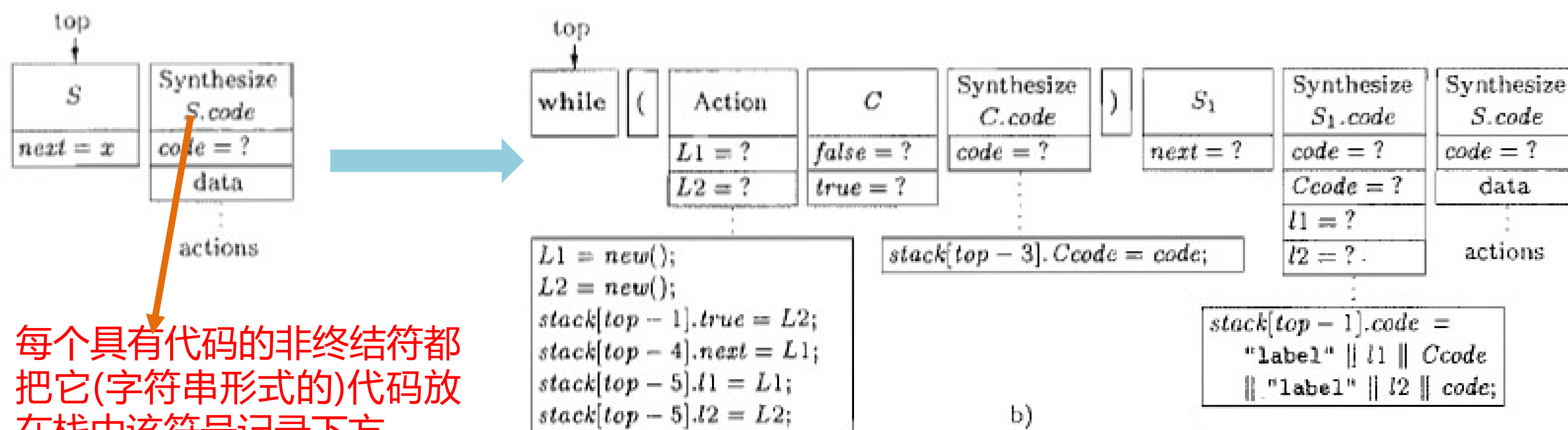
$S \rightarrow$ **while** (	$\{ L1 = new(); L2 = new(); C.false = S.next; C.true = L2; \}$
C)	$\{ S_1.next = L1; \}$
S_1	$\{ S.code =$ **label** $\Vert L1 \Vert C.code \Vert$ **label** $\Vert L2 \Vert S_1.code; \}$



L属性定义的LL实现

- **不断拷贝属性值，实现与LL分析同步的翻译**
  - 例2

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



6.7 L属性定义的LR分析实现

L属性定义的LR实现

我们可以处理 LR 文法上的 L 属性 SDD 吗?

在 5.4.1 节中,我们看到在 LR 文法上的每个 S 属性 SDD 都可以在自底向上语法分析过程中实现。根据 5.3.5 节,LL 文法上的每个 L 属性都可以在自顶向下语法分析中实现。因为 LL 文法类是 LR 文法类的一个真子集,并且 S 属性 SDD 类是 L 属性 SDD 类的一个真子集,那么我们能否以自底向上的方式处理每个 LR 文法和每个 L 属性 SDD 呢?

如下面的直观论述指出的,我们不能这么做。假设我们有一个 LR 文法的产生式 $A \to BC$,并且有一个继承属性 $B.i$,它依赖于 A 的继承属性。当我们规约到 B 的时候,我们还没有看到由 C 生成的输入,因此不能确定会扫描到产生式 $A \to BC$ 的体。因此,我们在此时还不能计算 $B.i$,因为我们不能确定是否使用和这个产生式相关联的规则。

也许我们可以等到已经归约得到 C,并且知道必须把 BC 归约到 A 时才进行计算。然而,即使到那个时候,我们仍然不知道 A 的继承属性,因为即使在归约之后,我们仍然不能确定包含这个 A 的是哪个产生式的体。我们可以说这个决定也应该推迟,因此也需要将 $B.i$ 的计算进一步推迟。如果我们继续这样推迟,我们很快会发现必须把所有的决定推迟到对整个输入的语法分析完成之后再进行。实质上,这就是"先构造语法分析树,再执行翻译"的策略。

L属性定义的LR实现

- **它能实现任何基于LL(1)文法的*L*属性定义**
- **也能实现许多（但不是所有的）基于LR(1) 的*L*属性定义**

删除翻译方案中嵌入的动作

为了进行LR规约，把动作改为后缀

改造完之后不再是**LR(1)**的文法

$E \rightarrow T\ R$

$R \rightarrow +\ T\ \{print\ (\text{'+'})\}R_1 \mid -\ T\ \{print\ (\text{'–'})\}R_1 \mid \varepsilon$

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

在文法中加入产生ε的标记非终结符，让每个嵌入动作由不同标记非终结符*M*代表，并把该动作放在产生式$M \rightarrow \varepsilon$的右端

$E \rightarrow T\ R$

$R \rightarrow +\ T\ M\ R_1 \mid -\ T\ N\ R_1 \mid \varepsilon$

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

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

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

这些动作的一个重要特点：没有引用原来产生式文法符号的属性；

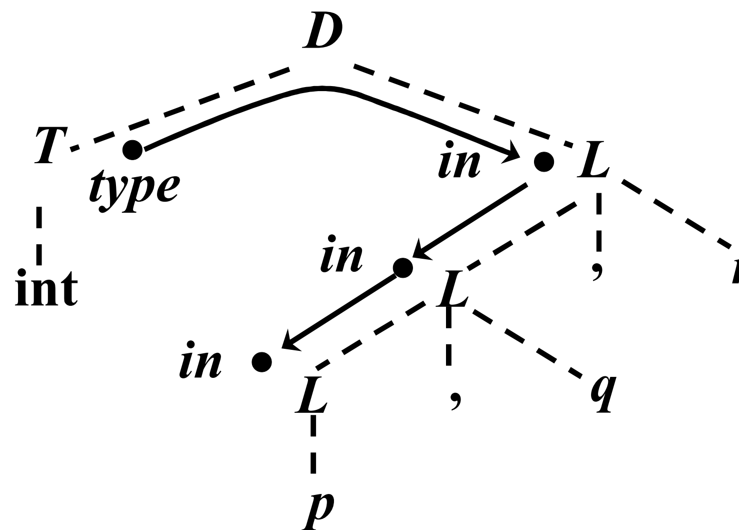
动作引用了属性怎么办？

L属性定义的LR实现

■ Case 1: 分析栈上的继承属性---位置可预测

例 int p, q, r

$D \rightarrow T \quad \{L.in = T.type\}$
$\qquad\qquad L$
$T \rightarrow$ int $\{T.type = integer\}$
$T \rightarrow$ real $\{T.type = real\}$
$L \rightarrow \quad \{L_1.in = L.in\}$
$\quad L_1$, id $\{addtype\ (\text{id}.entry, L.in)\}$
$L \rightarrow$ id $\{addtype\ (\text{id}.entry, L.in)\}$



L属性定义的LR实现

Case 1: 分析栈上的继承属性---位置可预测

例 int p, q, r

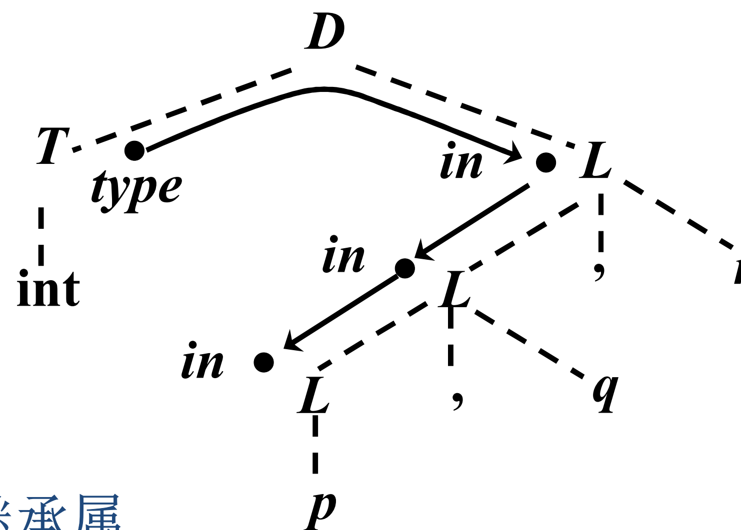
$D \to T$ {$L.in = T.type$}
L

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

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

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

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



继承属性的计算可以略去，引用继承属性的地方改成引用其他符号的综合属性

L属性定义的LR实现

■ Case 1: 分析栈上的继承属性---位置可预测

例 int p, q, r

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

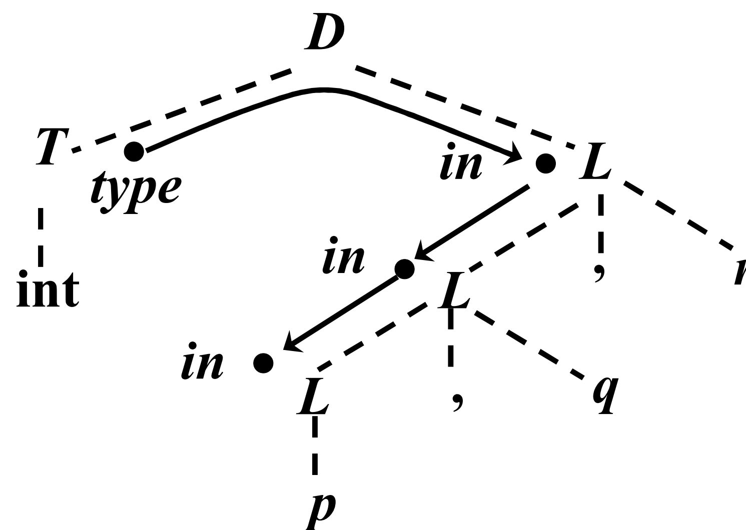
$T \rightarrow$ int {$T.type = integer$}

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

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

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

产 生 式	代 码 段
$D \rightarrow TL$	
$T \rightarrow$ int	$val[top] = integer$
$T \rightarrow$ real	$val[top] = real$
$L \rightarrow L_1$, id	$addType(val[top], val[top-3])$
$L \rightarrow$ id	$addType(val[top], val[top-1])$



L属性定义的LR实现

- **Case 2: 分析栈上的继承属性---位置不可预测**

$S \rightarrow aAC$	$C.i = A.s$
$S \rightarrow bABC$	$C.i = A.s$
$C \rightarrow c$	$C.s = g(C.i)$

增加标记非终结符，使得位置可以预测

$S \rightarrow aAC$	$C.i = A.s$
$S \rightarrow bABMC$	$M.i = A.s; C.i = M.s$
$C \rightarrow c$	$C.s = g(C.i)$
$M \rightarrow \varepsilon$	$M.s = M.i$

如果考虑*M.s*的可计算性

L属性定义的LR实现

Case 2: 分析栈上的继承属性---位置不可预测

继承属性是某个综合属性的一个函数

$S \rightarrow aAC$	$C.i = f(A.s)$
$C \rightarrow c$	$C.s = g(C.i)$

增加标记非终结符，把$f(A.s)$的计算移到对标记非终结符归约时进行

$S \rightarrow aANC$	$N.i = A.s$; $C.i = N.s$
$N \rightarrow \varepsilon$	$N.s = f(N.i)$
$C \rightarrow c$	$C.s = g(C.i)$

L属性定义的LR实现

- **例 数学排版语言EQN**

$S \rightarrow \quad \{B.ps = 10\}$
$\quad B \quad \{S.ht = B.ht\}$

$B \rightarrow \quad \{B_1.ps = B.ps\}$
$\quad B_1 \quad \{B_2.ps = B.ps\}$
$\quad B_2 \quad \{B.ht = max(B_1.ht, B_2.ht)\}$

$B \rightarrow \quad \{B_1.ps = B.ps\}$
$\quad B_1$
$\quad \text{sub} \quad \{B_2.ps = shrink(B.ps)\}$
$\quad B_2 \quad \{B.ht = disp(B_1.ht, B_2.ht)\}$

$B \rightarrow \text{text} \quad \{B.ht = \text{text}.h \times B.ps\}$

位置不可预测：

$B \rightarrow B_1B_2 \mid B_1 \text{ sub } B_2$

L属性定义的LR实现

产 生 式	语 义 规 则
$S \to LB$	$B.ps = L.s$; $S.ht = B.ht$
$L \to \varepsilon$	$L.s = 10$ 将$B.ps$存入栈中，便于引用
$B \to B_1\ MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s; B.ht = max(B_1.ht, B_2.ht)$
$M \to \varepsilon$	$M.s = M.i$
$B \to B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp(B_1.ht, B_2.ht)$
$N \to \varepsilon$	$N.s = shrink(N.i)$
$B \to$ text	$B.ht = \text{text}.h \times B.ps$

L属性定义的LR实现

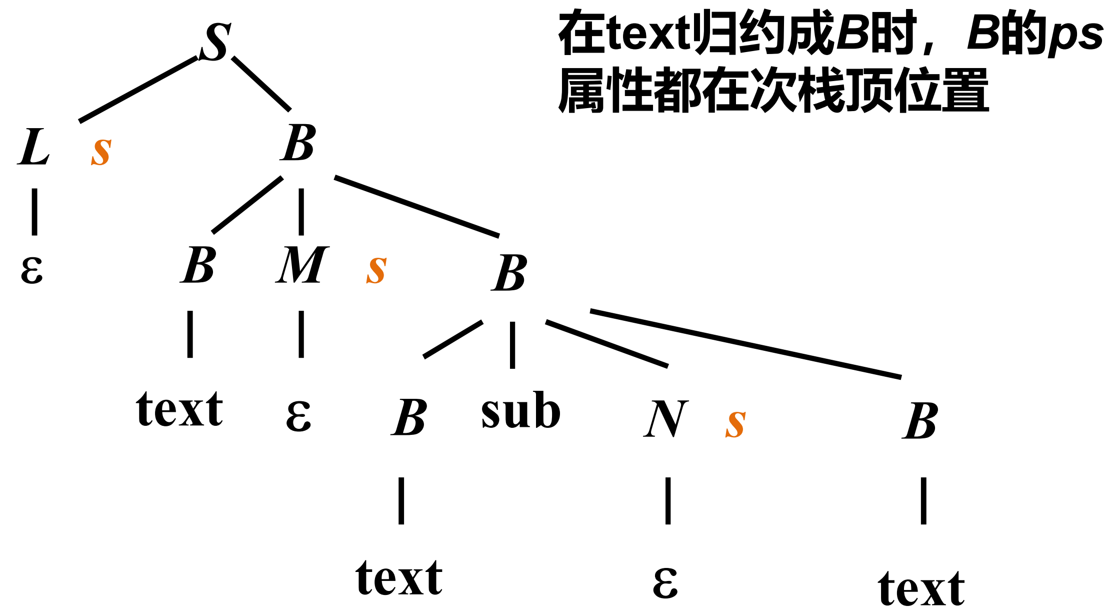
产 生 式	语 义 规 则
$S \to LB$	$B.ps = L.s$; $S.ht = B.ht$
$L \to \varepsilon$	$L.s = 10$ 将$B.ps$存入栈中，便于引用
$B \to B_1\ MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s$;$B.ht = max(B_1.ht, B_2.ht)$
$M \to \varepsilon$	$M.s = M.i$ 单纯为了属性位置可预测
$B \to B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp\ (B_1.ht, B_2.ht)$
$N \to \varepsilon$	$N.s = shrink(N.i)$
$B \to$ text	$B.ht = \text{text}.h \times B.ps$

L属性定义的LR实现

产 生 式	语 义 规 则
$S \to LB$	$B.ps = L.s$; $S.ht = B.ht$
$L \to \varepsilon$	$L.s = 10$ 将$B.ps$存入栈中，便于引用
$B \to B_1\ MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s$;$B.ht = max(B_1.ht, B_2.ht)$
$M \to \varepsilon$	$M.s = M.i$ 单纯为了属性位置可预测
$B \to B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp\ (B_1.ht, B_2.ht)$
$N \to \varepsilon$	$N.s = shrink(N.i)$ 兼有计算功能
$B \to$ text	$B.ht = \text{text}.h \times B.ps$

L属性定义的LR实现

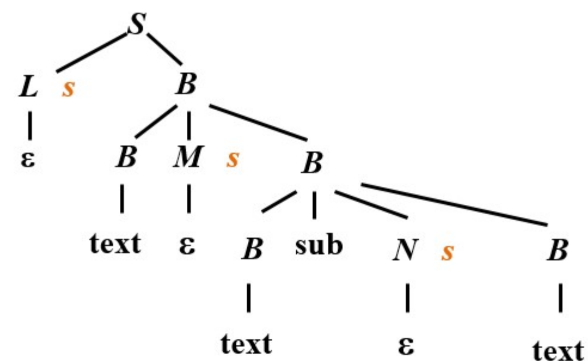
举例说明



在text归约成*B*时，*B*的*ps*属性都在次栈顶位置

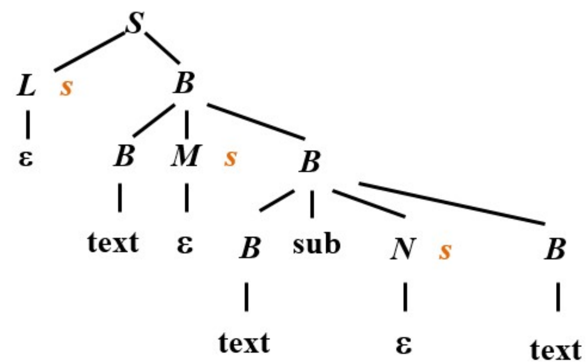
L属性定义的LR实现

产 生 式	语 义 规 则
$S \to LB$	$B.ps = L.s$; $S.ht = B.ht$
$L \to \varepsilon$	$L.s = 10$
$B \to B_1 MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s$;$B.ht = max(B_1.ht, B_2.ht)$
$M \to \varepsilon$	$M.s = M.i$
$B \to B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp(B_1.ht, B_2.ht)$
$N \to \varepsilon$	$N.s = shrink(N.i)$
$B \to$ text	$B.ht =$ text.$h \times B.ps$



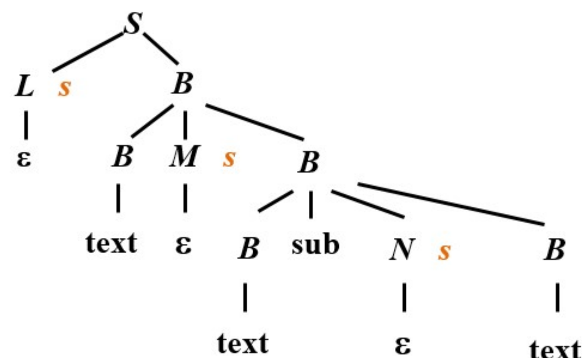
L属性定义的LR实现

产 生 式	代 码 段
$S \to LB$	$val[top-1] = val[top]$
$L \to \varepsilon$	$L.s = 10$
$B \to B_1 MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s$;$B.ht = max(B_1.ht, B_2.ht)$
$M \to \varepsilon$	$M.s = M.i$
$B \to B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp(B_1.ht, B_2.ht)$
$N \to \varepsilon$	$N.s = shrink(N.i)$
$B \to$ text	$B.ht =$ text$.h \times B.ps$



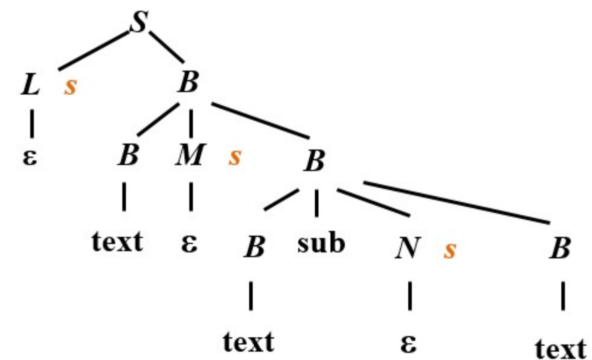
L属性定义的LR实现

产 生 式	代 码 段
$S \to LB$	$val[top-1] = val[top]$
$L \to \varepsilon$	$val[top+1] = 10$
$B \to B_1 MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s$; $B.ht = max(B_1.ht, B_2.ht)$
$M \to \varepsilon$	$M.s = M.i$
$B \to B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp(B_1.ht, B_2.ht)$
$N \to \varepsilon$	$N.s = shrink(N.i)$
$B \to$ text	$B.ht = \text{text}.h \times B.ps$



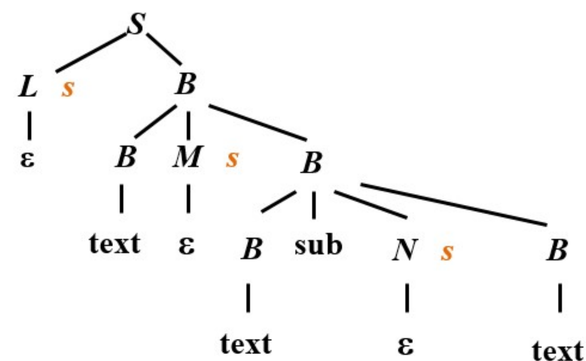
L属性定义的LR实现

产 生 式	代 码 段
$S \to LB$	$val[top-1] = val[top]$
$L \to \varepsilon$	$val[top+1] = 10$
$B \to B_1 MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \to \varepsilon$	$M.s = M.i$
$B \to B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp(B_1.ht, B_2.ht)$
$N \to \varepsilon$	$N.s = shrink(N.i)$
$B \to$ text	$B.ht = \text{text}.h \times B.ps$



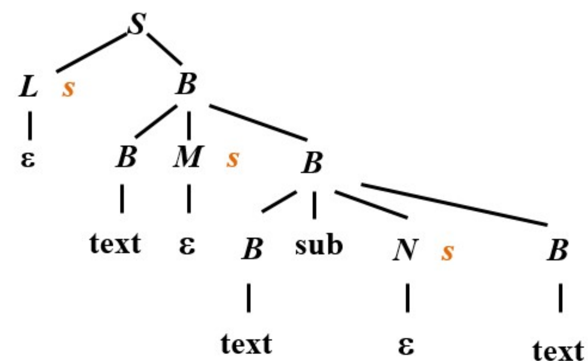
L属性定义的LR实现

产 生 式	代 码 段
$S \to LB$	$val[top-1] = val[top]$
$L \to \varepsilon$	$val[top+1] = 10$
$B \to B_1 MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \to \varepsilon$	$val[top+1] = val[top-1]$
$B \to B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp(B_1.ht, B_2.ht)$
$N \to \varepsilon$	$N.s = shrink(N.i)$
$B \to$ text	$B.ht = \text{text}.h \times B.ps$



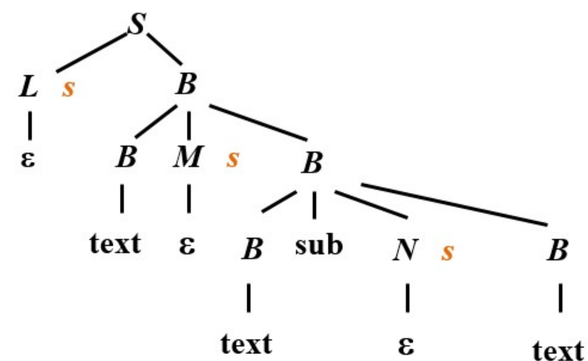
L属性定义的LR实现

产 生 式	代 码 段
$S \to LB$	$val[top-1] = val[top]$
$L \to \varepsilon$	$val[top+1] = 10$
$B \to B_1 MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \to \varepsilon$	$val[top+1] = val[top-1]$
$B \to B_1$ sub NB_2	$val[top-3] = disp\ (val[top-3], val[top]\ )$
$N \to \varepsilon$	$N.s = shrink(N.i)$
$B \to$ text	$B.ht =$ text$.h \times B.ps$



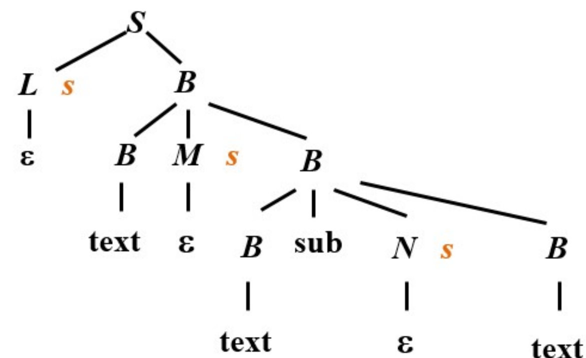
L属性定义的LR实现

产 生 式	代 码 段
$S \to LB$	$val[top-1] = val[top]$
$L \to \varepsilon$	$val[top+1] = 10$
$B \to B_1 MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \to \varepsilon$	$val[top+1] = val[top-1]$
$B \to B_1$ sub NB_2	$val[top-3] = disp\ (val[top-3], val[top]\ )$
$N \to \varepsilon$	$val[top+1] = shrink(val[top-2])$
$B \to$ text	$B.ht = \text{text}.h \times B.ps$



L属性定义的LR实现

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1 MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1$ sub NB_2	$val[top-3] = disp\ (val[top-3], val[top]\ )$
$N \rightarrow \varepsilon$	$val[top+1] = shrink(val[top-2])$
$B \rightarrow$ text	$val[top] = val[top] \times val[top-1]$



Thank you!