# 第五章 语法制导的翻译

**许畅**

南京大学计算机系

2017年春季

# 介绍

- 使用上下文无关文法引导语言的翻译
  - CFG的非终结符号代表了语言的某个构造
  - 程序设计语言的构造由更小的构造组合而成
  - 一个构造的语义可以由小构造的含义综合而来
    - 比如：表达式x + y的类型由x、y的类型和运算符+决定
  - 也可以从附近的构造继承而来
    - 比如：声明"int x"中x的类型由它左边的类型表达式决定

# 语法制导定义和语法制导翻译

- **语法制导定义**
  - 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
    - $E \rightarrow E_1 + T$ $\quad$ $E.code = E_1.code \mid\mid T.code \mid\mid \text{'+'}$
  - 属性*code*代表表达式的逆波兰表示，规则说明加法表达式的逆波兰表示由两个分量的逆波兰表示并置，然后加上'+'得到
- **语法制导翻译**
  - 在产生式体中加入语义动作，并在适当时候执行动作
    - $E \rightarrow E_1 + T$ $\quad$ { print '+'; }

# 语法制导的定义 (SDD)

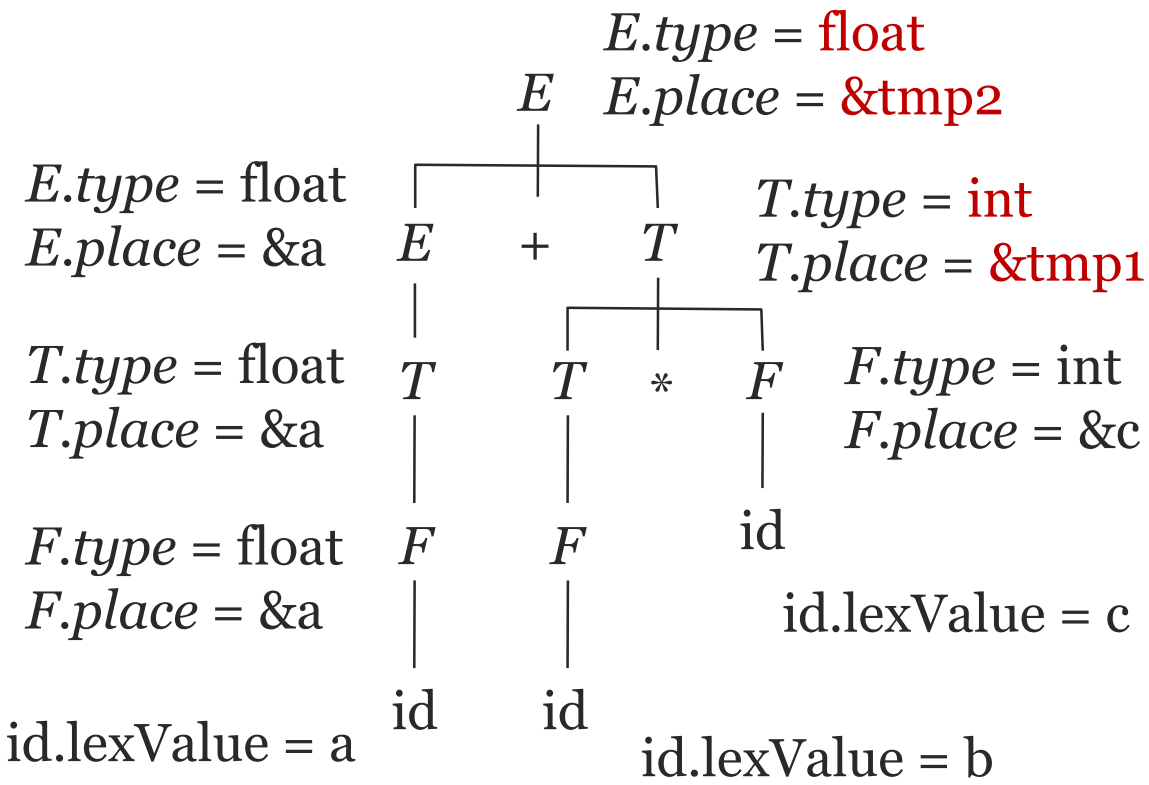
- **Syntax-Directed Definition** (SDD) 是上下文无关文法和属性/规则的结合
  - 属性和文法符号相关联，按照需要来确定各个文法符号需要哪些属性
  - 规则和产生式相关联
- 对于文法符号$X$和属性$a$，我们用$X.a$表示分析树中某个标号为$X$的结点的值
- 一个分析树结点和它的分支对应于一个产生式规则，而对应的语义规则确定了这些结点上的属性的取值和计算

# 分析树和属性值 (1)

- 假设我们需要知道一个表达式的类型，以及对应代码将它的值存放的内存地址
  - 我们需要两个属性：*type*, *place*
- 产生式规则：$E \rightarrow E_1 + T$
  - 假设只有int/float类型
  - $E.type$ = if ($E_1.type$ == $T.type$) $T.type$ else float;
  - $E.place$ = newTempPlace(); // 返回一个新的内存位置
- 产生式规则：$F \rightarrow$ id
  - $F.type$ = lookupIDTable(id.lexValue)->*type*;
  - $F.place$ = lookupIDTable(id.lexValue)->*address*;

# 分析树和属性值 (2)

- 输入a + b * c的语法分析树以及属性值




(1) 假设a, b, c是已声明的全局变量，a的类型为float，b和c的类型为int

(2) 中间未标明的*T*和*F*的*type*和*place*是int和&b

# 综合属性和继承属性

- **综合属性 (Synthesized Attribute)**
  - 结点*N*的属性值由*N*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 (Inherited Attribute)**
  - 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
- 几条约束
  - 不允许*N*的继承属性通过*N*的子结点上的属性来定义，但允许*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 (来自词法分析)，但无继承属性

# SDD的例子

- 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$E_1$和$T$的$val$值...
- 终结符号digit有综合属性lexval

| 产生式 | 语义规则 |
|---|---|
| 1) $L \rightarrow E\ \mathbf{n}$ | $L.val = E.val$ |
| 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) $F \rightarrow \mathbf{digit}$ | $F.val = \mathbf{digit}.\mathrm{lexval}$ |

# S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头部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/文法符号可以附加相应的属性值
  - 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**

# 语法分析树上的SDD求值 (1)

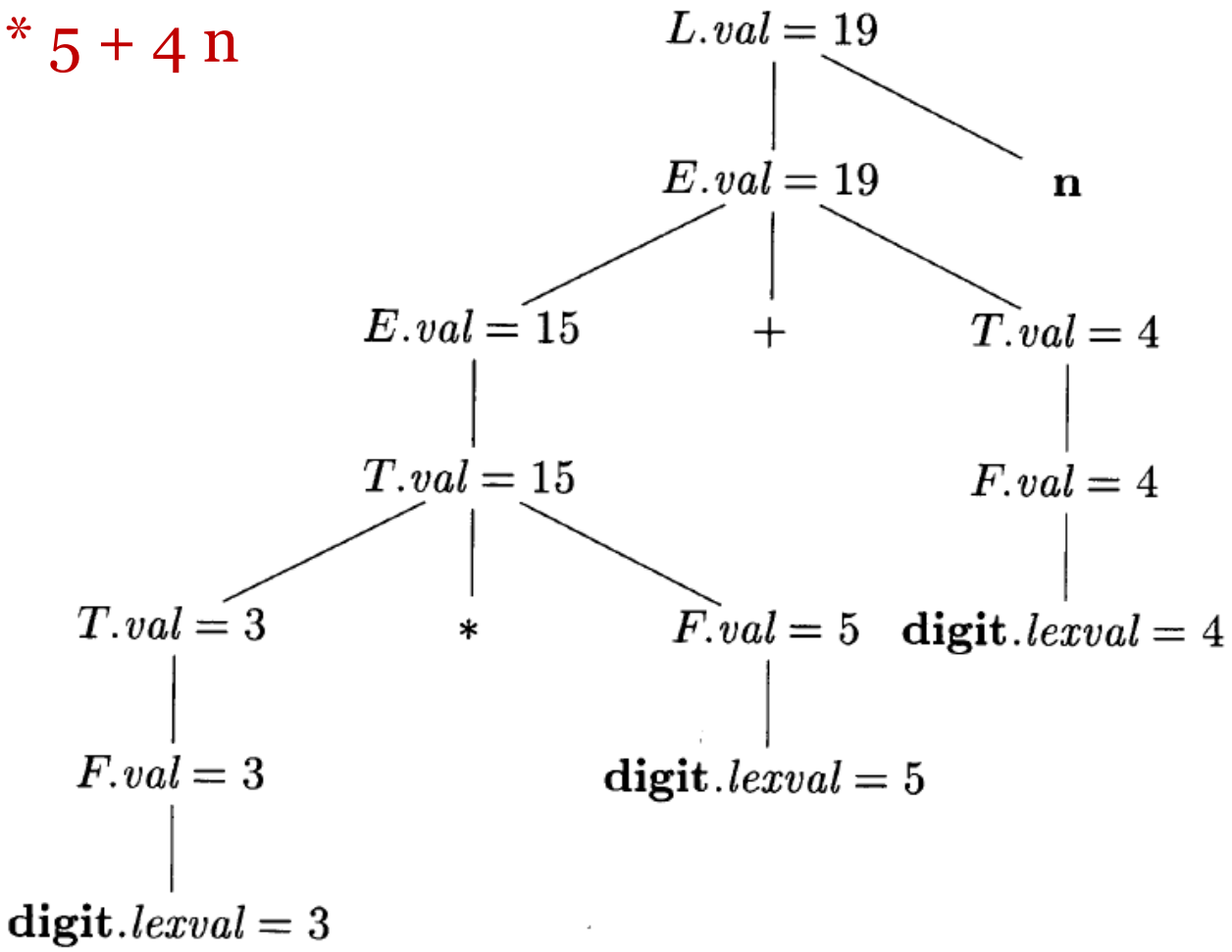
- 实践中很少先构造语法分析树再进行SDD求值，但在分析树上求值有助于翻译方案的可视化，便于理解
- **注释语法分析树**
  - 包含了各个结点的各属性值的语法分析树
- 步骤
  - 对于任意的输入串，首先构造出相应的分析树
  - 给各个结点 (根据其文法符号) 加上相应的属性
  - 按照语义规则计算这些属性的值

# 语法分析树上的SDD求值 (2)

- 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
  - 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
  - $A \rightarrow B \qquad A.s = B.i \qquad B.i = A.s + 1$

# 注释分析树的例子

3 * 5 + 4 n



# 适用于自顶向下分析的SDD (1)

- 前面的文法存在左递归，我们无法用自顶向下的分析方法进行处理
- 但消除左递归之后，我们无法直接使用属性*val*进行处理
  - 比如规则：$T \rightarrow FT'$　　$T' \rightarrow *FT' \mid \varepsilon$
  - $T$对应的项中，第一个因子对应$F$，而运算符却在$T'$中
  - 需要用继承属性来完成这样的计算

比较$T \rightarrow T*F \mid F$

# 适用于自顶向下分析的SDD (2)

比较$T \rightarrow T * F \mid F$

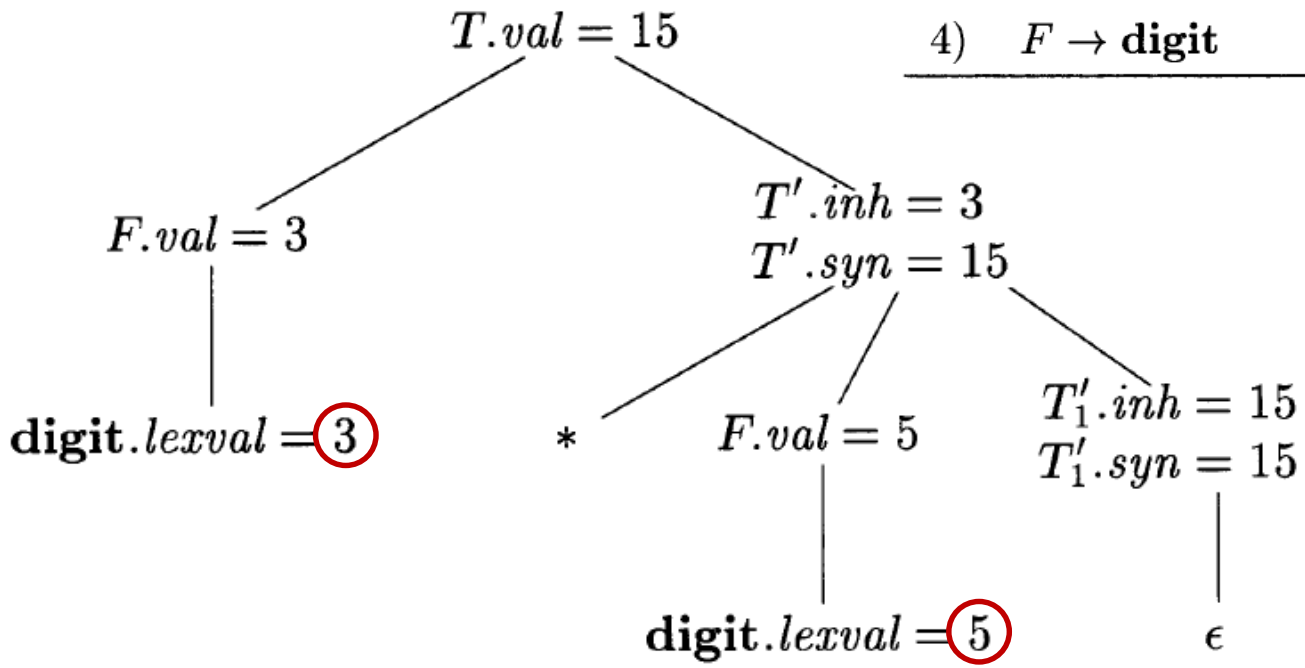
| 产生式 | 语义规则 |
|---|---|
| 1) $T \rightarrow F\,T'$ | $T'.inh = F.val$<br>$T.val = T'.syn$ |
| 2) $T' \rightarrow *\,F\,T'_1$ | $T'_1.inh = T'.inh \times F.val$<br>$T'.syn = T'_1.syn$ |
| 3) $T' \rightarrow \epsilon$ | $T'.syn = T'.inh$ |
| 4) $F \rightarrow \mathbf{digit}$ | $F.val = \mathbf{digit}.lexval$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

$T'$的属性$inh$实际上继承了相应的*号的左运算分量

# 3 * 5的注释分析树

- 观察*inh*属性的传递

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



# 消除直接左递归时语义规则的处理

- 假设
  - $A \rightarrow A_1\, Y \qquad A.a = g(A_1.a,\ Y.y)$
  - $A \rightarrow X \qquad A.a = f(X.x)$
- 那么
  - $A \rightarrow X\, R \qquad R.i = f(X.x);\ A.a = R.s$
  - $R \rightarrow Y\, R_1 \qquad R_1.i = g(R.i,\ Y.y);\ R.s = R_1.s$
  - $R \rightarrow \varepsilon \qquad R.s = R.i$

$R$即是我们以前消除左递归时引入的$A'$

# 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$a_i$，才能计算$N$的属性$a$
- 使用**依赖图**来表示计算顺序
  - 这些值的计算顺序形成一个偏序关系，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
- 对于分析树结点$N$，与$N$关联的每个属性$a$都对应依赖图的一个结点$N.a$

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

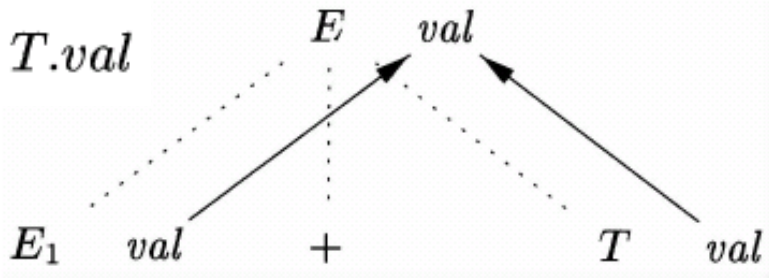
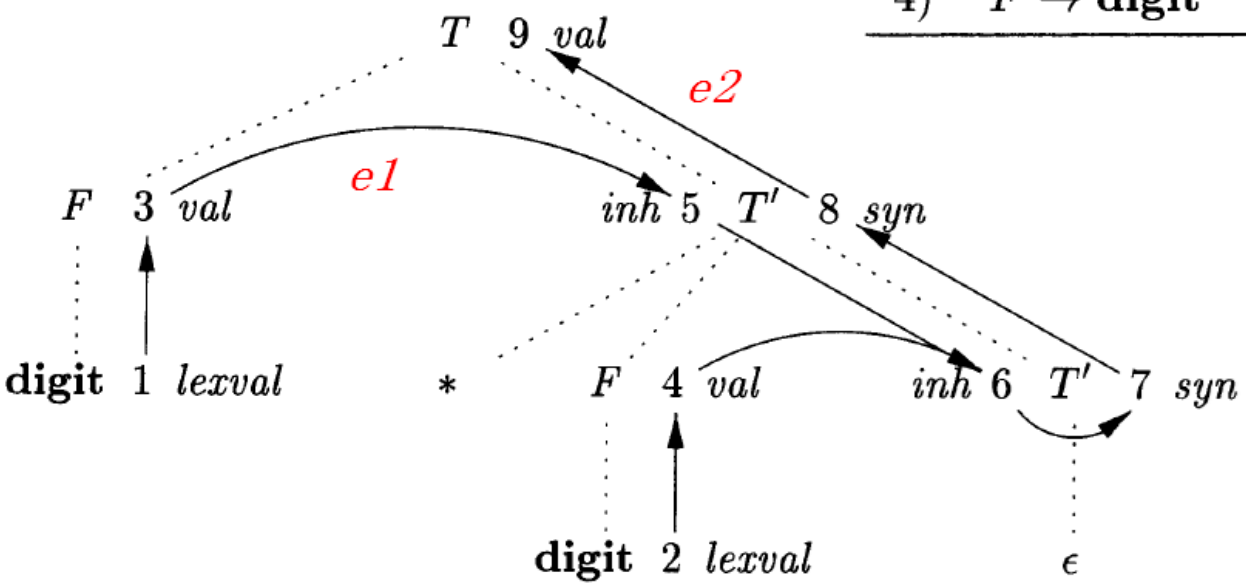



图 5-6　$E.val$ 由 $E_1.val$ 和 $T.val$ 综合得到

# 依赖图的例子

- 3 * 2的注释分析树
- $T \rightarrow F\,T'$
  - $T'.inh = F.val$
    $T.val = T'.syn$
  - 边e1, e2

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



# 属性值的计算顺序

- 各个属性值需要按照依赖图的拓扑顺序计算
  - 如果依赖图中存在环，则属性计算无法进行
- 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
- 但是特定类型的SDD一定不包含环，且有固定的计算顺序
  - 如：S属性的SDD，L属性的SDD

# S属性的SDD

- 每个属性都是综合属性，都是根据子构造的属性计算出父构造的属性
- 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，可以和自底向上或自顶向下的语法分析过程一起计算
  - 自底向上
    - 在构造分析树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)
  - 自顶向下
    - 在递归子程序法中，在过程$A()$的最后计算$A$的属性(此时$A$调用的其他过程(对应于其子结构)已经调用完毕)

# 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，各子结点的属性已计算完毕
    对N的各个属性求值;
}
```

- 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

- 每个属性
  - 是综合属性，或
  - 是继承属性，且$A \rightarrow X_1X_2...X_n$中计算$X_i.a$的规则只能用
    - $A$的继承属性，或
    - $X_i$左边的文法符号$X_j$的继承属性或综合属性，或
    - $X_i$自身的继承或综合属性(这些属性间的依赖关系不形成环)
- 特点
  - 依赖图中的边
    - 综合属性从下到上
    - 继承属性从上到下，或从左到右
  - 计算一个属性值时，它所依赖的属性值都已计算完毕

# L属性SDD和自顶向下语法分析 (1)

- 在递归子程序法中实现L属性
  - 对于每个非终结符号$A$，其所对应过程的参数为继承属性，返回值为综合属性
- 在处理规则$A \rightarrow X_1X_2...X_n$时
  - 在调用$X_i()$之前计算$X_i$的继承属性值，然后以它们为参数调用$X_i()$
  - 在该产生式对应代码的最后计算$A$的综合属性
  - 如果所有文法符号的属性计算按上面的方式进行，计算顺序必然和依赖关系一致

# L属性SDD和自顶向下语法分析 (2)

- L属性SDD其属性总可以按如下方式计算

```
L_dfvisit(n)
{
    for m = 从左到右n的每个子节点 do
    {
        计算m的继承属性;
        L_dfvisit(m);
    }
    计算n的综合属性;
}
```

# L属性SDD的例子及反例

- 非L属性的例子
  - $A \rightarrow BC \qquad A.s = B.b; \qquad B.i = f(C.c, A.s)$

- L属性的例子

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

# 具有受控副作用的语义规则

- 属性文法没有副作用，但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，然后通过函数来添加新的标识符
- **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
  - 总在最后执行，不影响其它属性的求值
- 变量声明SDD中的副作用
  - *addType*将标识符的类型信息加入标识符表中
  - 只要标识符不被重复声明，其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) $L \rightarrow L_1\ ,\ \textbf{id}$ | $L_1.inh = L.inh$<br>$addType(\textbf{id}.entry, L.inh)$ |
| 5) $L \rightarrow \textbf{id}$ | $addType(\textbf{id}.entry, L.inh)$ |

# SDD的应用例子

- 抽象语法树的构造
- 基本类型和数组类型的L属性定义

# 构造抽象语法树的SDD

- **抽象语法树**
  - 每个结点代表一个语法结构，对应于运算符
  - 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的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
- 抽象语法树的表示方法
  - 每个结点用一个对象表示
  - 对象有多个域
    - 叶子结点中只存放词法值
    - 内部结点中存放了op值和参数(通常指向其它结点)

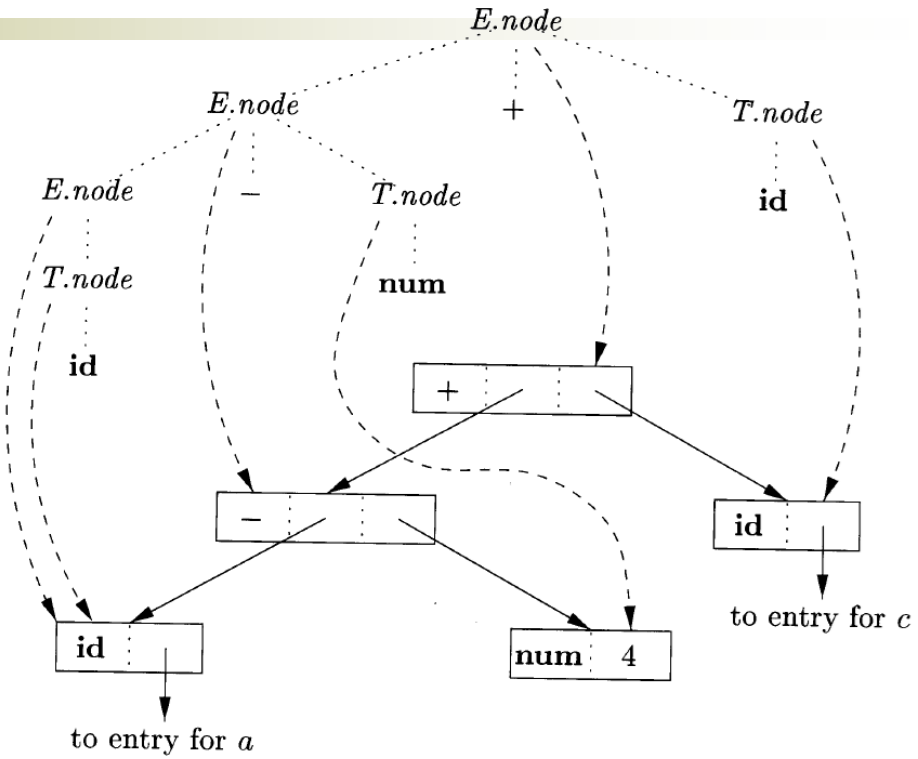
# 构造简单表达式的抽象语法树的SDD

- 属性$E.node$指向$E$对应的抽象语法树的根结点

| 产生式 | 语义规则 |
|---|---|
| 1) $E \rightarrow E_1 + T$ | $E.node = \textbf{new}\ Node('+', E_1.node, T.node)$ |
| 2) $E \rightarrow E_1 - T$ | $E.node = \textbf{new}\ Node('-', E_1.node, T.node)$ |
| 3) $E \rightarrow T$ | $E.node = T.node$ |
| 4) $T \rightarrow (\ E\ )$ | $T.node = E.node$ |
| 5) $T \rightarrow \textbf{id}$ | $T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$ |
| 6) $T \rightarrow \textbf{num}$ | $T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$ |

# 表达式抽象语法树的构造过程

- 输入
  - a − 4 + c




- 步骤

$p_1$ = new Leaf(id, entry_a);

$p_2$ = new Leaf(num, 4);

$p_3$ = new Node('−', $p_1$, $p_2$);

$p_4$ = new Leaf(id, entry_c);

$p_5$ = new Node('+', $p_3$, $p_4$);

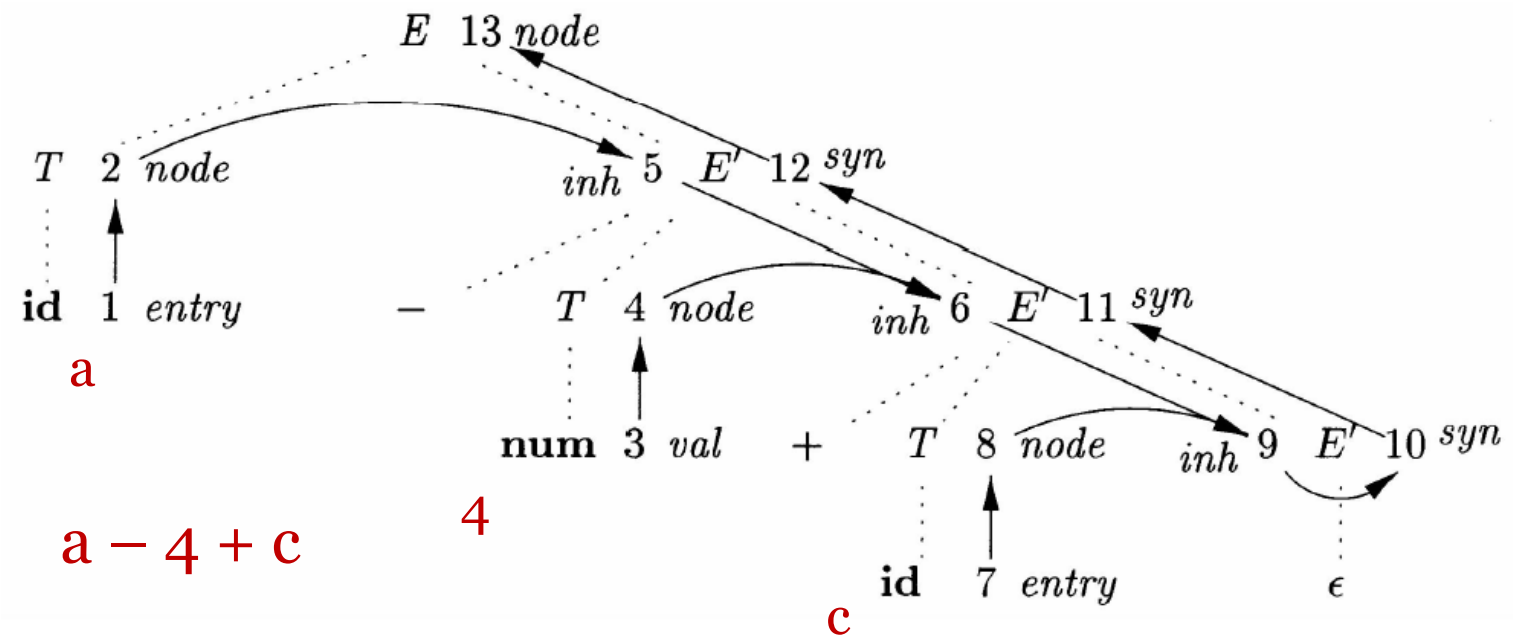
# 自顶向下方式处理的L属性定义 (1)

- 消除左递归时，按照规则得到如下SDD

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

# 自顶向下方式处理的L属性定义 (2)

- 对这个SDD，各属性值的计算过程与原来S属性定义中的计算过程一致
- 继承属性可以把值从一个结构传递到另一个并列的结构，也可把值从父结构传递到子结构



# 类型结构

- 简化的类型表达式的语法
  - $T \rightarrow B\ C$
  - $B \rightarrow$ int | float
  - $C \rightarrow$ [num] $C$ | ε
- 生成类型表达式的SDD

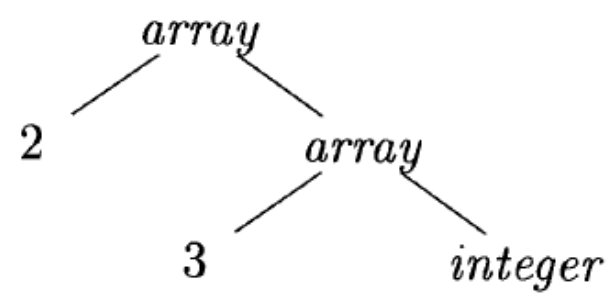
| 产生式 | 语义规则 |
|---|---|
| $T \rightarrow B\ C$ | $T.t = C.t$ |
| | $C.b = B.t$ |
| $B \rightarrow$ **int** | $B.t = integer$ |
| $B \rightarrow$ **float** | $B.t = float$ |
| $C \rightarrow [\ \mathbf{num}\ ]\ C_1$ | $C.t = array(\mathbf{num}.val,\ C_1.t)$ |
| | $C_1.b = C.b$ |
| $C \rightarrow \epsilon$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

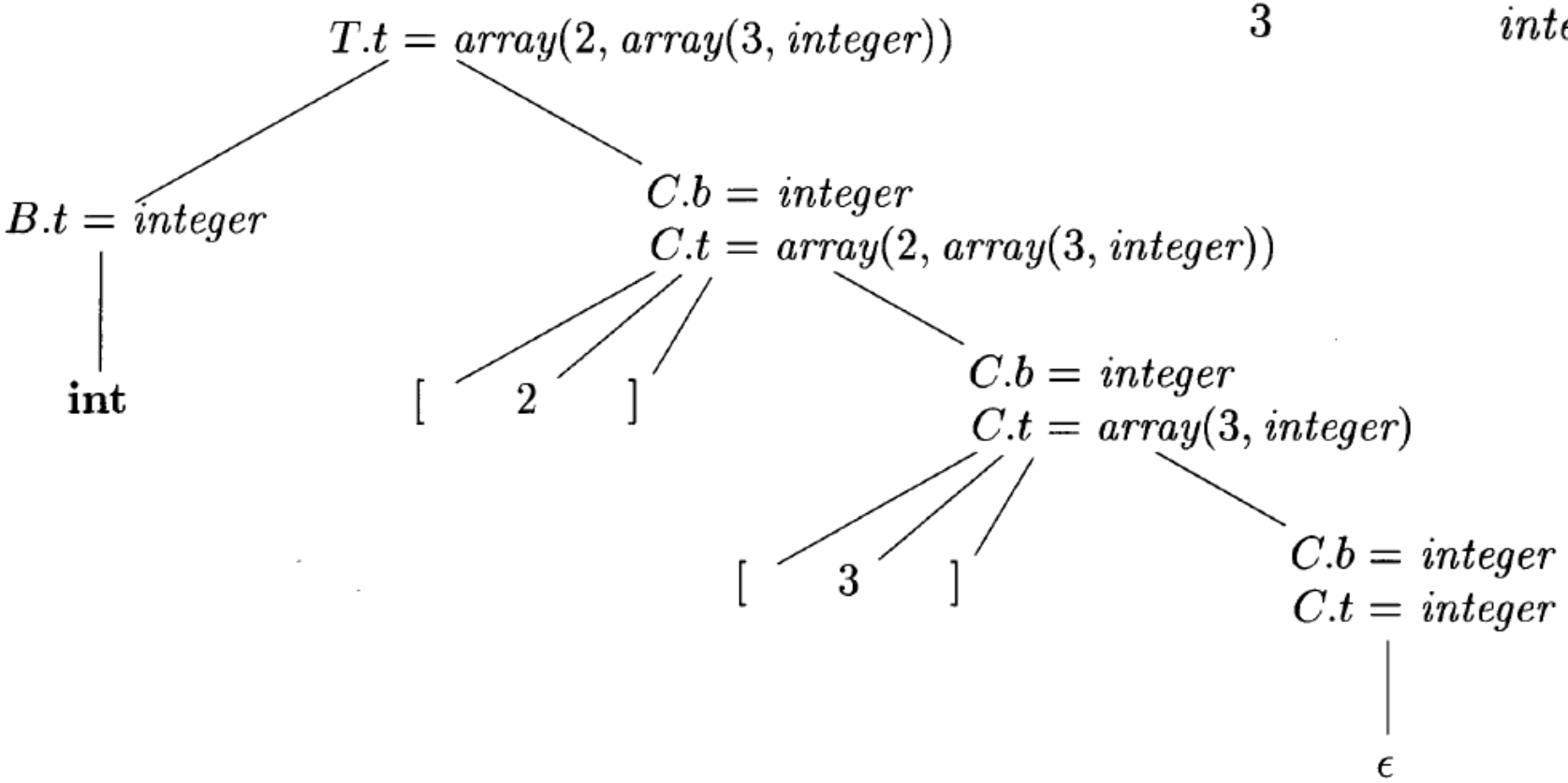
# 类型的含义

- 类型包括两个部分：$T \rightarrow B\ C$
  - 基本类型 $B$
  - 分量 $C$
- 分量形如[2][3]
  - 表示2×3的二维数组
  - 如：int [2][3]
- 数组构造算符array
  - array(2, array(3, int))表示抽象的2×3的二维数组

# 类型表达式的生成过程

- 输入：int [2][3]



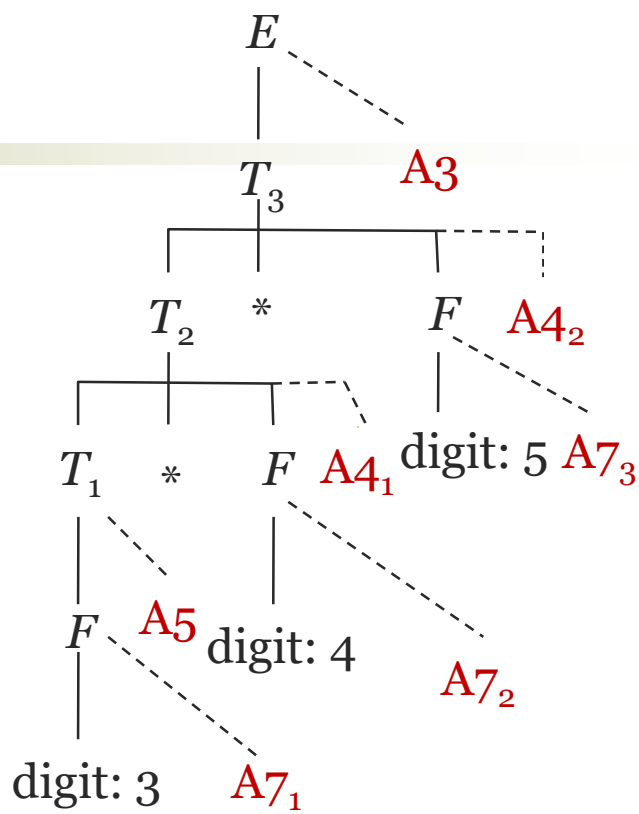



# 语法制导的翻译方案

- **语法制导的翻译方案** (SDT) 是在产生式体中嵌入语义动作 (程序片断) 的上下文无关文法
- SDT的基本实现方法
  - 建立语法分析树
  - 将语义动作看作是虚拟结点
  - 从左到右、深度优先地遍历分析树，在访问虚拟结点时执行相应的语义动作
- 用SDT实现两类重要的SDD
  - 基本文法是LR的，SDD是S属性的
  - 基本文法是LL的，SDD是L属性的

# 例子

- 语句3 * 4 * 5的分析树
- 动作执行顺序
  - $A7_1, A5, A7_2, A4_1, A7_3, A4_2, A3$
  - 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 |

# 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
- 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非终结符号$M_i$，其产生式为$M_i \rightarrow \varepsilon$
  - 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

# SDT可否用特定分析技术实现的例子

- 新文法
  - $L \rightarrow E\ \mathrm{n}\ M_1$ $\quad M_1 \rightarrow \varepsilon$
  - $E \rightarrow E + T\ M_2$ $\quad M_2 \rightarrow \varepsilon$
  - $E \rightarrow T\ M_3$ $\quad M_3 \rightarrow \varepsilon$
  - ……

| | | | | |
|---|---|---|---|---|
| $L$ | $\rightarrow$ | $E\ \mathbf{n}$ | $\{\ \mathrm{print}(E.val);\ \}$ | 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 |

# 后缀翻译方案

- 文法可以自底向上分析 (即LR的) 且其SDD是S属性的，必然可以构造出后缀SDT
- **后缀SDT**：所有动作都在产生式最右端的SDT
- 构造方法
  - 将每个语义规则看作是一个赋值语义动作
  - 将所有的语义动作放在规则的最右端

# 后缀翻译方案的例子

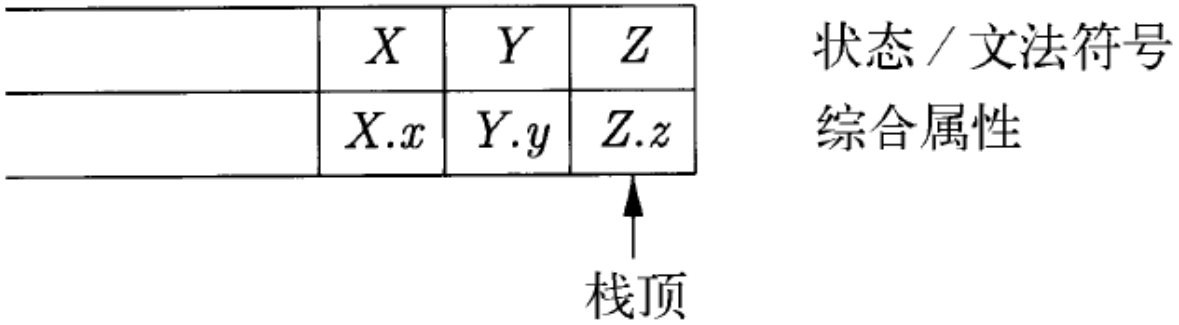
- 实现桌上计算器的后缀SDT

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

- 注意动作中对属性值的引用
  - 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

# 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（或状态）都附带一个这样的union类型的值
  - 在按照产生式$A \rightarrow XYZ$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



# 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*]指向这个栈的栈顶
  - *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中存放了什么值

# 后缀SDT的栈实现

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

注意：$stack[top-i]$和文法符号的对应

# 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
- 对一般的SDT，都可以先建立分析树(语义动作作为虚拟结点)，然后进行前序遍历并执行动作
- 不是所有的SDT都可以在分析过程中实现
  - 后缀SDT以及L属性对应的SDT可以在分析时完成

# 消除左递归时SDT的转换 (1)

- 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，然后消除左递归
- 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 ('+');} $R$
  - $R \rightarrow \varepsilon$

# 消除左递归时SDT的转换 (2)

- 如果涉及属性值的计算，则有通用的解决方案
- 假设
  - $A \rightarrow A_1 Y \{ A.a = g(A_1.a, Y.y) \}$
  - $A \rightarrow X \{ A.a = f(X.x) \}$
- 那么
  - $A \rightarrow X \{ R.i = f(X.x) \} R \{ A.a = R.s \}$
  - $R \rightarrow Y \{ R_1.i = g(R.i, Y.y) \} R_1 \{ R.s = R_1.s \}$
  - $R \rightarrow \varepsilon \{ R.s = R.i \}$

# L属性的SDT

- 除了通用的SDT实现技术，若基础文法是LL的，则可以将L属性SDD转换成一个SDT，该SDT可以在自顶向下的分析过程中实现
- 从L属性的SDD到SDT的转换
  - 将每个语义规则看作是一个赋值语义动作
  - 将赋值语义动作放到相应产生式 $(A \rightarrow X_1 X_2 \ldots X_n)$ 的适当位置
    - 计算$X_i$继承属性的动作插入到产生式体中$X_i$的左边
    - 计算产生式头$A$综合属性的动作在产生式的最右边

# while语句的SDD和SDT

- 产生式$S \rightarrow$ while $(C)\ S_1$
  - 为while语句生成中间代码
  - 主要说明语句控制流中的标号生成
- while语句的含义
  - 首先对$C$求值，若为真，则控制转向$S_1$的开始处
  - 若为假，则转向while语句的后续语句开始处
  - $S_1$结束时，要能够跳转到while语句的代码开始处

# L属性的SDD的例子

- SDD

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

- 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

# 转换为SDT

- 语义动作
  - (a) $L1$ = new( )和$L2$ = 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) $S.code = \ldots$：计算$S$的综合属型
- 根据放置语义动作的规则得到如下SDT
  - (b)在$C$之前，(c)在$S_1$之前，(d)在最右端
  - (a)可以放在最前面

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

# L属性SDD的实现

- 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

# 递归下降实现L属性SDD的例子

```
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(); C.false C.true
        L2 = new();
        Ccode = C(next, L2);
        检查 ')' 是下一个输入符号 , 并读取输入;
        Scode = S(L1); ← S.next
        return("label" || L1 || Ccode || "label" || L2 || Scode);
    }
    else /* 其他语句类型 */
}
```

# 边扫描边生成属性 (1)

- 当属性值的体积很大，对其进行运算会效率很低
  - code可能是一个上百K的串，对其进行并置会很低效
- 可逐步生成属性的各个部分，并增量式地添加到最终的属性值中
- 三个条件
  - 存在一个主属性，且其为综合属性
  - 在产生式中，主属性是通过产生式体中各非终结符号的主属性连接而得到，同时还会连接一些其它元素
  - 各个非终结符号的主属性的连接顺序与它们在产生式体中的顺序相同

# 边扫描边生成属性 (2)

- 基本思想
  - 在适当的时候“发出”元素，并拼接到适当的地方
- 举例说明
  - 假设扫描一个非终结符号对应的语法结构，调用其相应的函数，并生成主属性
    - $S \rightarrow \text{while}\ (C)\ S_1$
    - $\{\ S.code = \text{label} \mid\mid L1 \mid\mid C.code \mid\mid \text{label} \mid\mid L2 \mid\mid S_1.code\ \}$
  - 如果各函数都把其主属性打印出来，则对while语句，只需先打印label $L1$，再调用$C$ (打印$C$的代码)，再打印label $L2$，再调用$S$ (打印$S_1$的代码)
  - 需要在适当的时候打印label $L1$和label $L2$

# 边扫描边生成属性的例子 (1)

- 新的SDT

$S \rightarrow$ while ( { $L1$ = new(); $L2$ = new( ); $C.false = S.next$;
$C.true = L2$; print("label", $L1$); }
$C$ ) { $S_1.next = L1$; print("label", $L2$); }
$S_1$

- 前提是所有非终结符号的SDT规则都这么做

对照原SDT

| | |
|---|---|
| $S \rightarrow$ **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** $\|\|$ $L2$ $\|\|$ $S_1.code$; } |

# 边扫描边生成属性的例子 (2)

原来的

```
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 */
}
```

边扫描边生成