第五章 语法制导的翻译

冯 洋

fengyang@nju.edu.cn

引言

- 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
  - **$E \rightarrow E_1+T$　　$E.code=E_1.code \| \| T.code \| \| '+'$**
    - 属性code代表中缀表达式的逆波兰表示(后缀表示法?)，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

语法制导定义和语法制导翻译

- 逆波兰表达式，英文为 Reverse Polish notation，跟波兰表达式（Polish notation）相对应
- 波兰的数理科学家 Jan Łukasiewicz 发明
  - 中缀表达式: (1 + 2) * (3 + 4)加减乘除等运算符写在中间
  - 波兰表达式: (* (+ 1 2) (+ 3 4))，将运算符写在前面，因而也称为前缀表达式。
  - 逆波兰表达式: ((1 2 +) (3 4 +) *)，将运算符写在后面，因而也称为后缀表达式

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
  - **$E \rightarrow E_1+T$　　$E.code=E_1.code \| T.code \| \text{‘+’}$**
    - 属性code代表中缀表达式的逆波兰表示(后缀表示法)，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，并在适当的时候执行这些语义动作
  - **$E \rightarrow E_1+T$　　{print ‘+’;}**

语法制导的定义（SDD）

- SDD是对上下文无关文法的推广
  - 将每个文法符号和一个语义属性集合相关联
  - 将每个产生式和一组语义规则相关联，这些规则用于计算该产生式中各文法符号的属性值

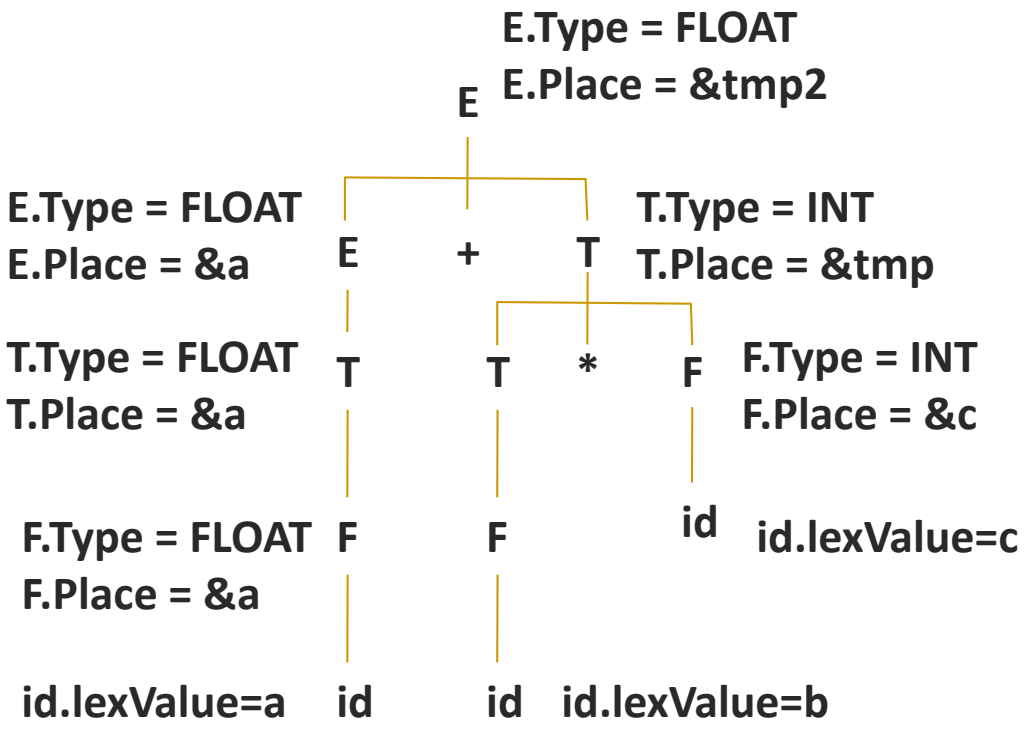
语法制导的定义（SDD）

- 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
  - 主要就是为CFG中的文法符号设置语义属性，具体来说，主要就是将一些语义属性附加到代表语言构造的文法符号上，从而把信息和语言构造联系起来
- 对于文法符号X和属性a，用X.a表示分析树中的某个标号为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

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在何处，我们就需要两个属性：type，place
- 产生式规则：$E \rightarrow E_1+T$
- 语义规则：(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 →id
  - F.type = lookupIDTable(id.lexValue)->type;
  - F.place = lookupIDTable(id.lexValue)->address;

分析树和属性值(2)



•a+b*c的语法分析树以及属性值

假设a, b, c是已经声明的全局变量，a的类型为FLOAT，b, c的类型为INT

中间未标明的T和F的type和address都是INT和&b;

继承属性和综合属性

- **综合属性**(synthesized attribute)：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**(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
- **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

继承属性和综合属性

- 综合属性（synthesized attribute）：在分析树节点N上的非终结符号A的综合属性是由N上的产生式所关联的语义规则来定义的。注意：这个产生式的头一定是A。节点N上的综合属性只能通过N的子节点或N本身的属性值来定义。【终结符可以有综合属性，为词法分析器提供的词法值】
  - 通过N的子结点或N本身的属性值来定义

SDD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

- 目标：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

习题

- 已知文法：
  - $E \rightarrow E - T \mid T$
  - $T \rightarrow num \mid num\ .\ num$

请给出一个SDD，来确定减法表达式的类型。

注：T有综合属性type（属性值分为integer和float两种），E有综合属性type。可以使用函数getType(type_1, type_2)来为减法表达式获取类型。

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

L属性的SDD

- 每个属性
  - 要么是综合属性
  - 要么是继承属性，且产生式$A \rightarrow X_1X_2\cdots X_n$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

语法分析树上的SDD求值（1）

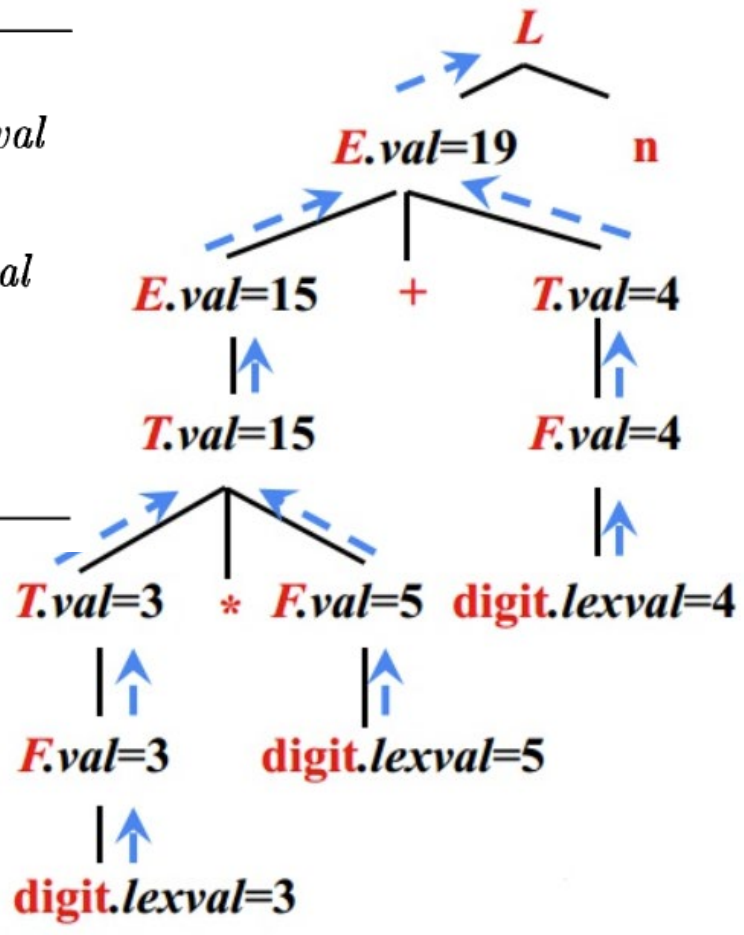
- 实践中很少先构造语法分析树再进行SDD求值
- 但在分析树上求值有助于翻译方案的可视化，便于理解
- 注释语法分析树
  - 包含了各个结点的各属性值的语法分析树
- 步骤：
  - 对于任意的输入串，首先构造出相应的分析树。
  - 给各个结点（根据其文法符号）加上相应的属性值
  - 按照语义规则计算这些属性值即可

语法分析树上的SDD求值（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
  - 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　　循环定义
  - $A \rightarrow B$　　A.s=B.i;　　B.i=A.s+1;

注释语法分析树的例子

	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

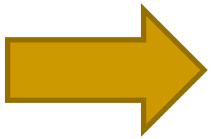


- 对于输入3*5+4n
- n表示结束

适用于自顶向下分析的SDD

- 左递归文法无法直接用自顶向下方法处理

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



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

适用于自顶向下分析的SDD

- 计算 3*5所用注释语法树
  - 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

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

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

适用于自顶向下分析的SDD

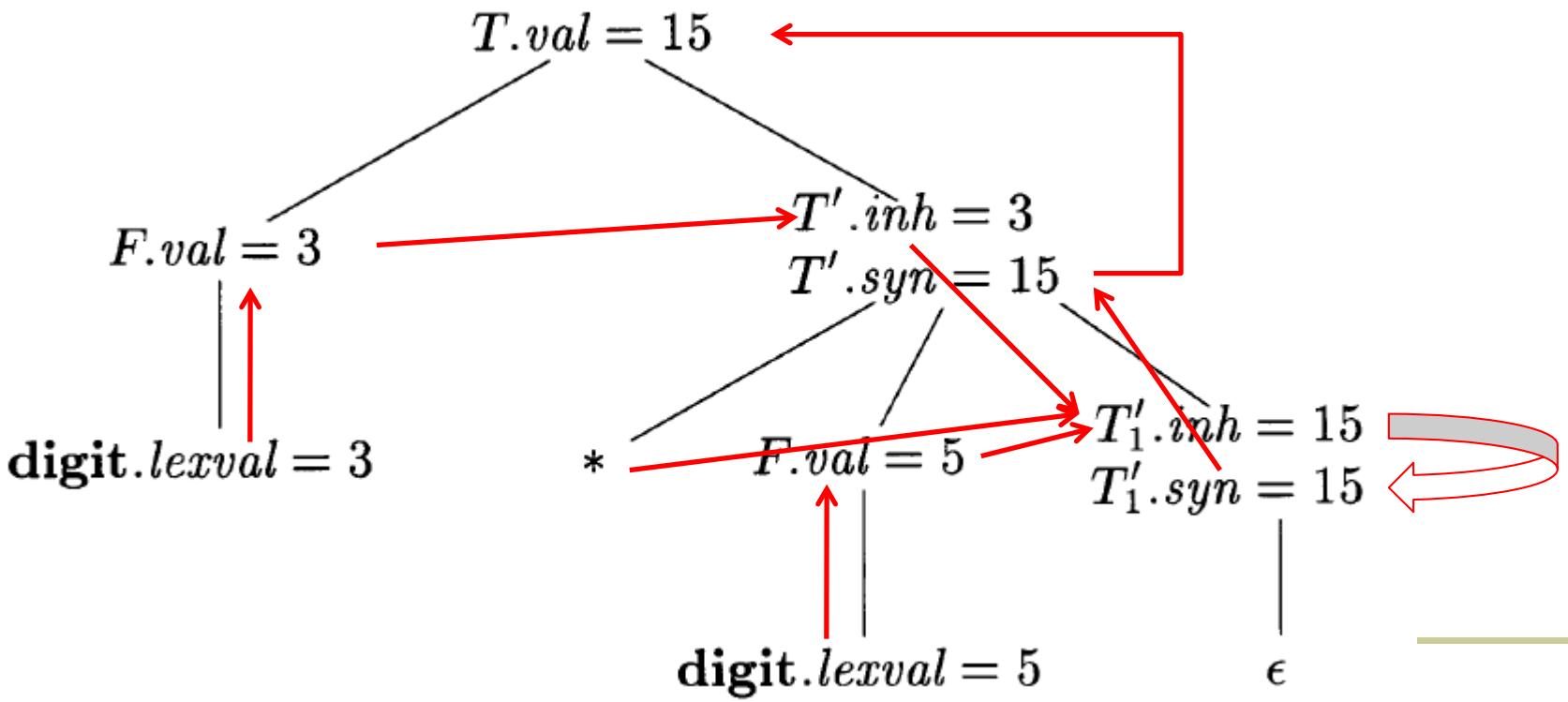
产生式	语义规则
1) $T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4) $F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

- 注意：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

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) $F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

注意观察inh属性是如何传递的



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

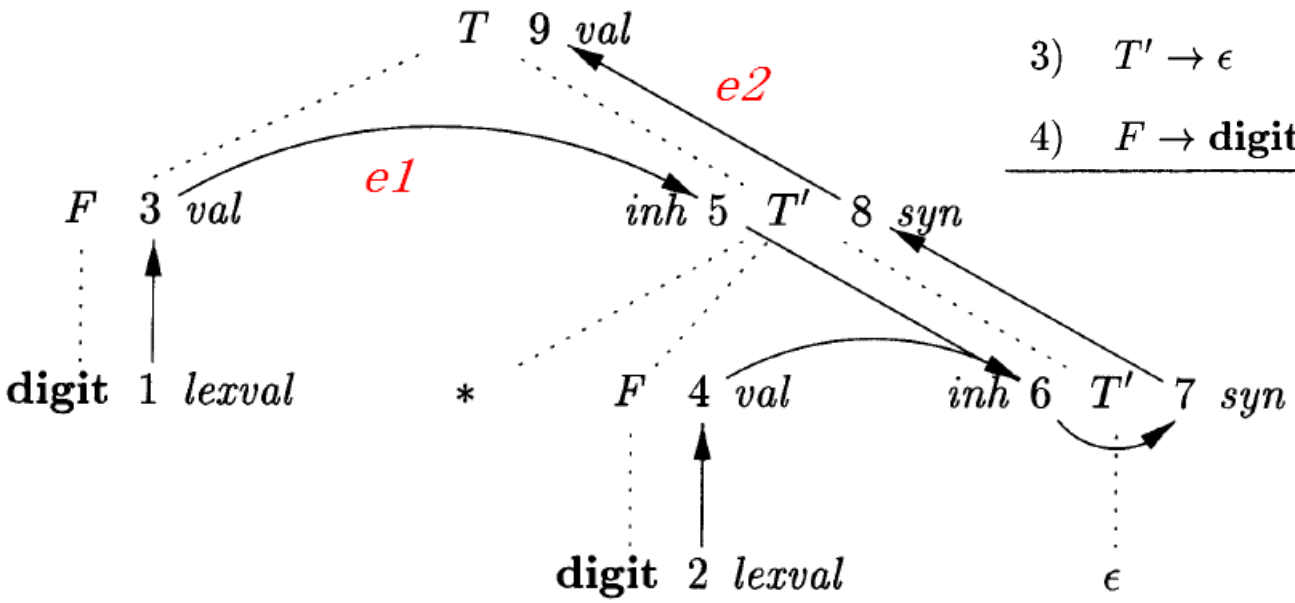
依赖图

- 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

- 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

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) $F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$



属性值的计算顺序

- 各个属性的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
- 怎样的SDD可以和以前所介绍的语法分析结合在一起？

属性值的计算顺序

- 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

- 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。可以和自顶向下、自底向上的语法分析过程一起计算
  - 自底向上
    - 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）
  - 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）

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;
}
```

- 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
  - 要么是综合属性
  - 要么是继承属性，且产生式$A \rightarrow X_1X_2\cdots X_n$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
- 特点
  - 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

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

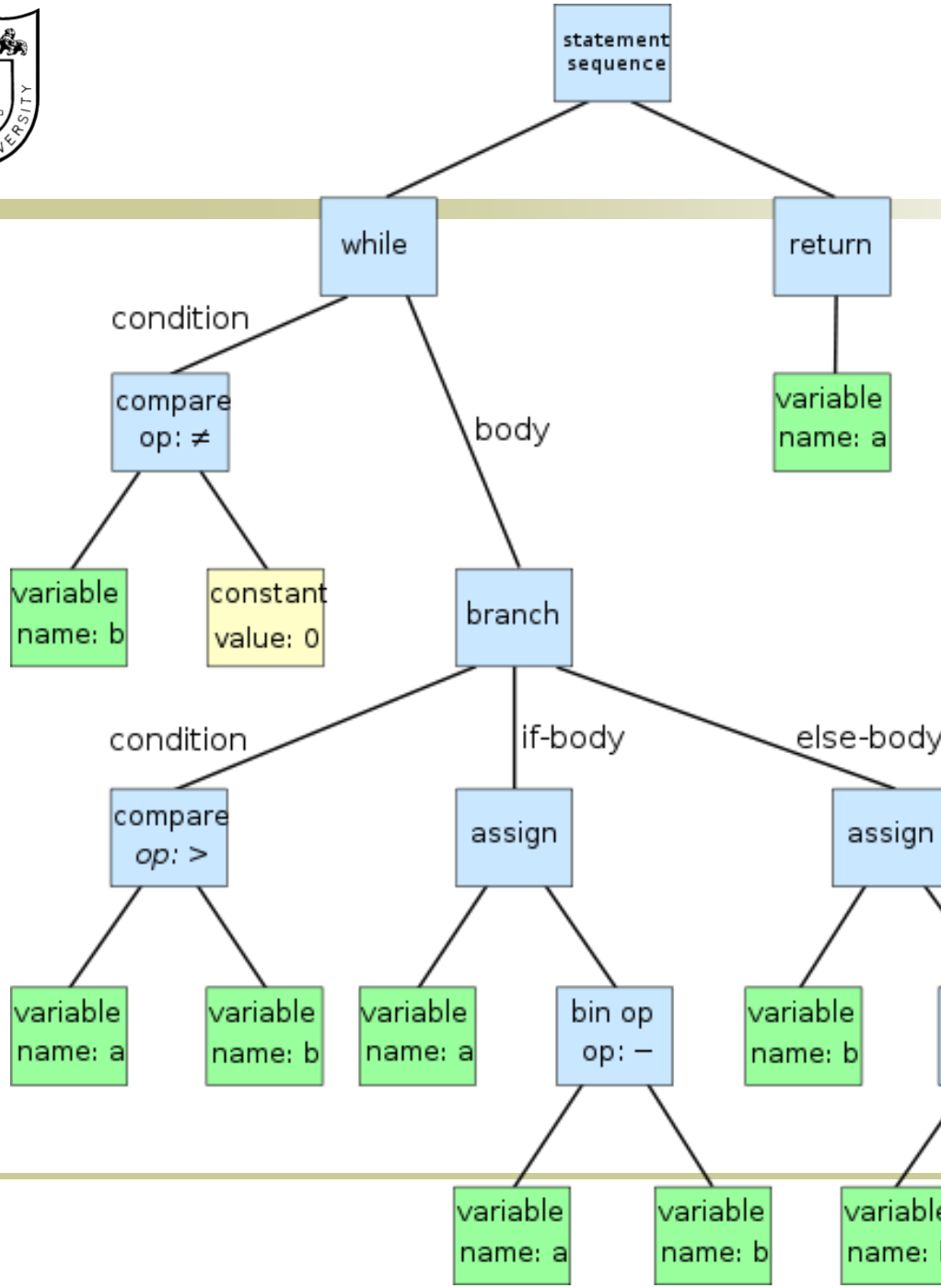
产生式	语义规则
1) $D \rightarrow T\ L$	$L.inh = T.type$
2) $T \rightarrow \mathbf{int}$	$T.type = \text{integer}$
3) $T \rightarrow \mathbf{float}$	$T.type = \text{float}$
4) $L \rightarrow L_1\ ,\ \mathbf{id}$	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5) $L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry,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

语法制导翻译的应用例子

- 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
    - 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）




```
while b ≠ 0
    if a > b
        a := a − b
    else
        b := b − a
```

构造简单表达式的语法树的SDD

- 属性E. node指向E对应的语法树的根结点

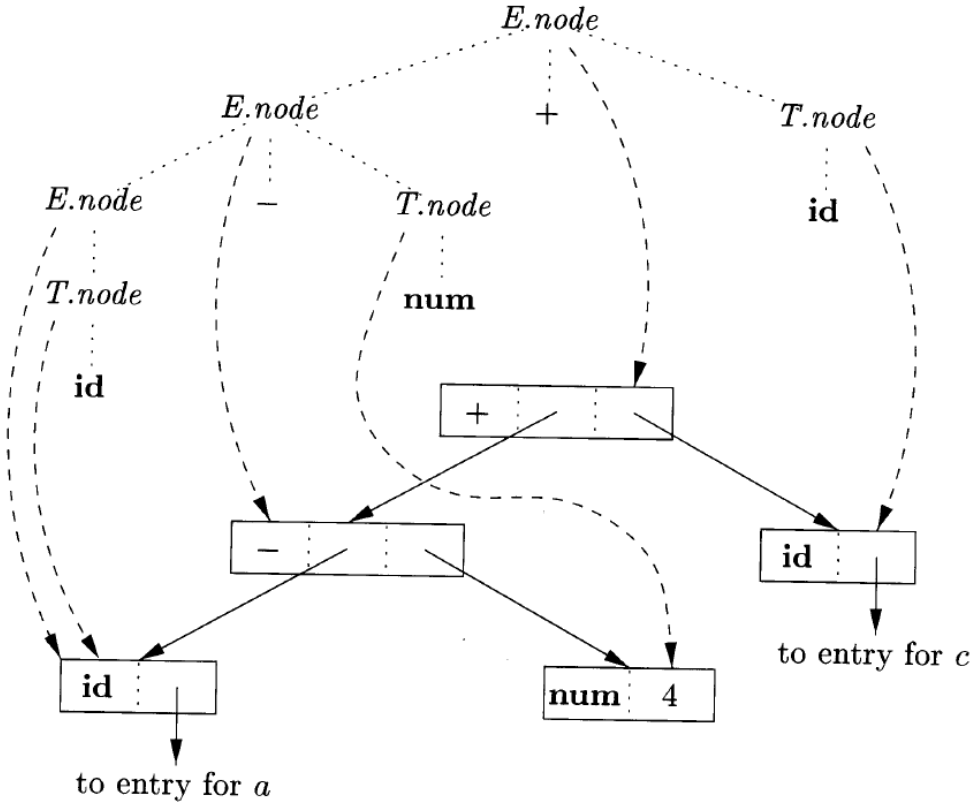
	产生式	语义规则
1)	$E \to E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (\ E\ )$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$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

表达式语法树的构造过程

- 输入：

  a-4+c

- 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);



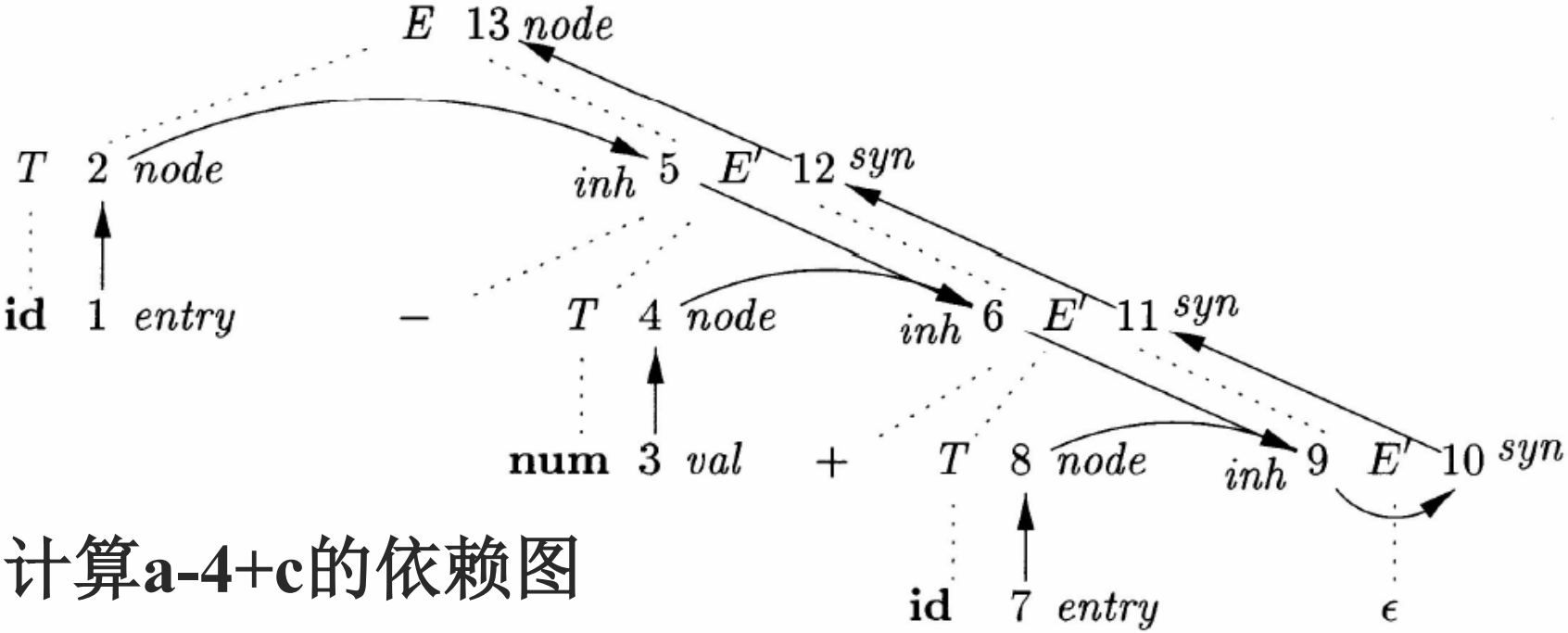
自顶向下方式处理的L属性定义（1）

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

- 在消除左递归时，按照规则得到此SDD

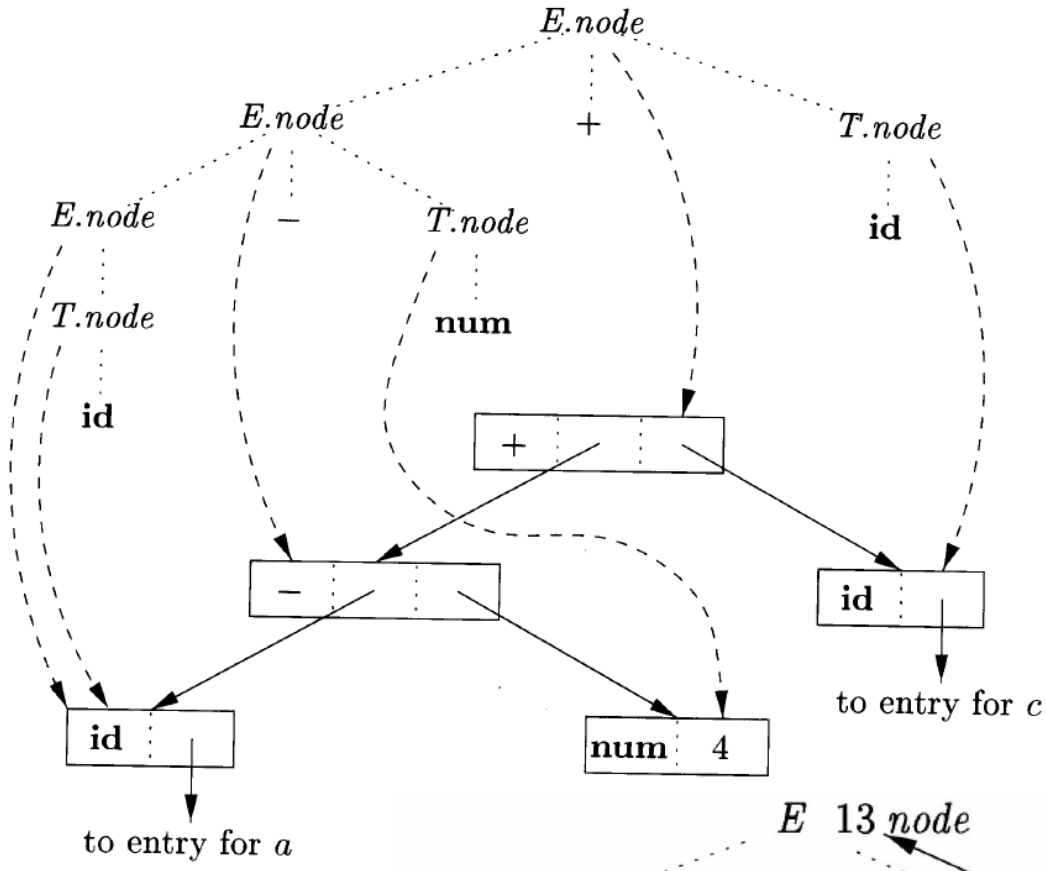
自顶向下方式处理的L属性定义（2）

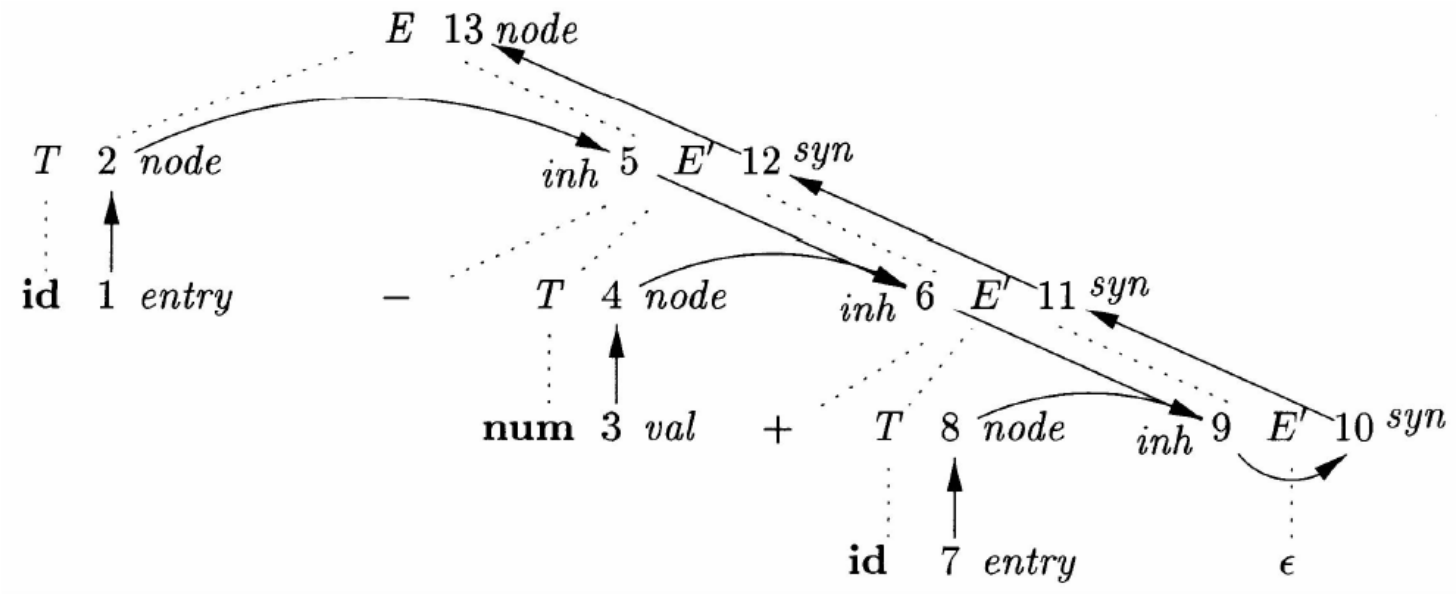
- 对于这个SDD，各属性值的计算过程实际上和原来S属性定义中的计算过程一致
- 继承属性可以把值从一个结构传递到另一个并列的结构；也可把值从父结构传递到子结构
- 抽象语法树和分析树不一致时，继承属性很有用



计算a-4+c的依赖图

- 不同的语法分析树与抽象语法树
  - **a-4+c**





类型结构

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

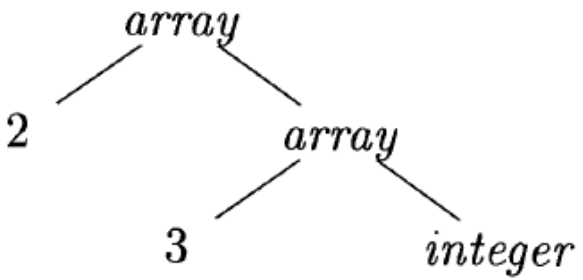
产生式	语义规则
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow \mathbf{int}$	$B.t = integer$
$B \rightarrow \mathbf{float}$	$B.t = float$
$C \rightarrow [\ \mathbf{num}\ ]\ C_1$	$C.t = array(\mathbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

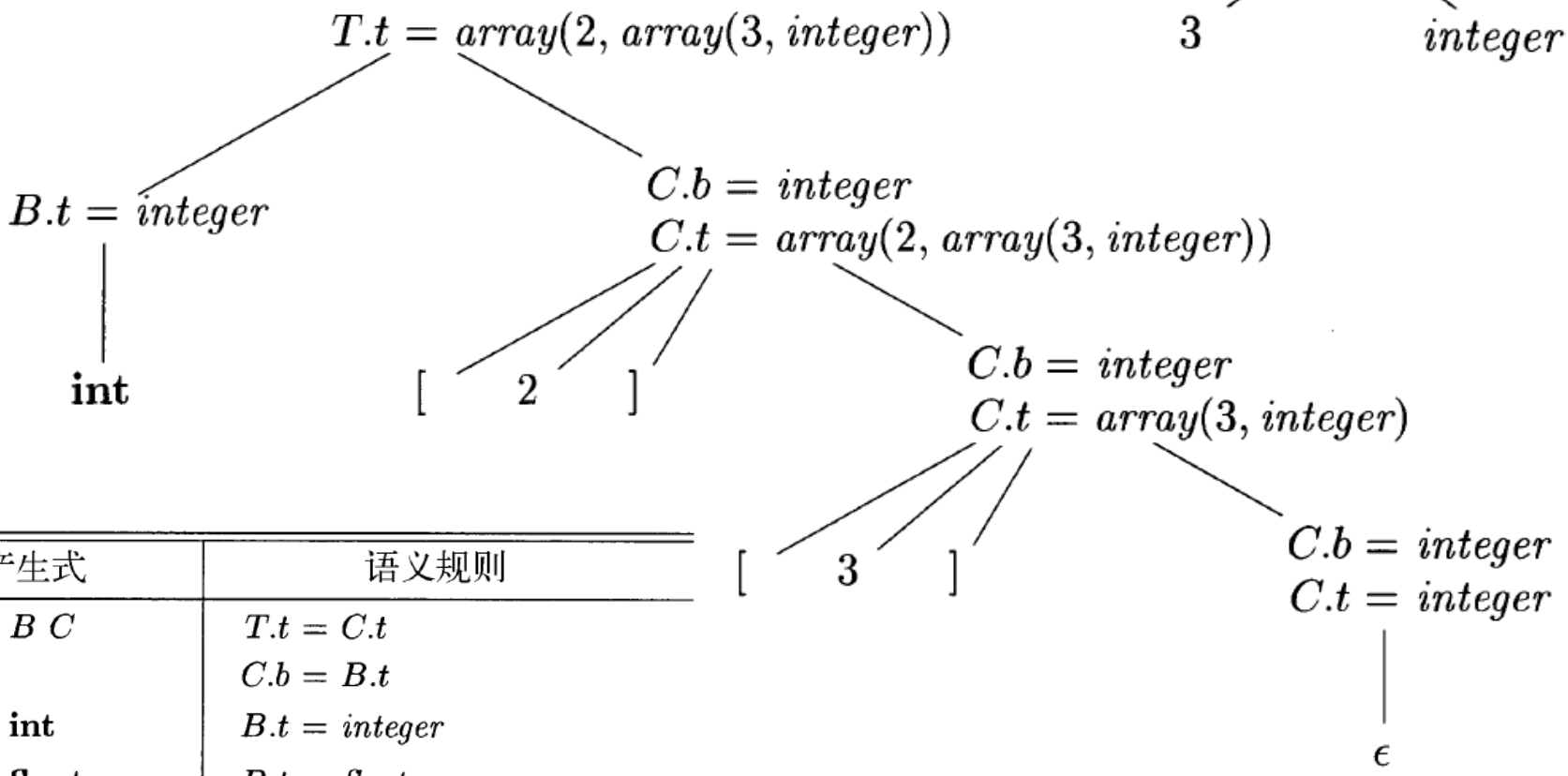
类型的含义

- 类型包括两个部分：T → B C
  - 基本类型　B
  - 分量 C
- 分量形如[3][4]
  - 表示3X4的二维数组
- 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

类型表达式的生成过程

- 输入：int [2][3]





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

习题

- 令S.val为下面文法由S生成的二进制数的值（例如，对于输入101.101,S.val=5.625）：

S → L.L

S → L

L → LB

L →B

B →0

B →1

按语法制导翻

产生式	语义规则
S`→S	print(S.val)
S→$L_1.L_2$	S.val:=L_1.val+L_2.val/$2^{L2.length}$
S→L	S.val:=L.val
L→L_1B	L.val:=L_1.val*2+B.val L.length:=L_1.length+1
L→B	L.val:=B.val L.length:=1
B→0	B.val:=0
B→1	B.val:=1

语法制导的翻译方案

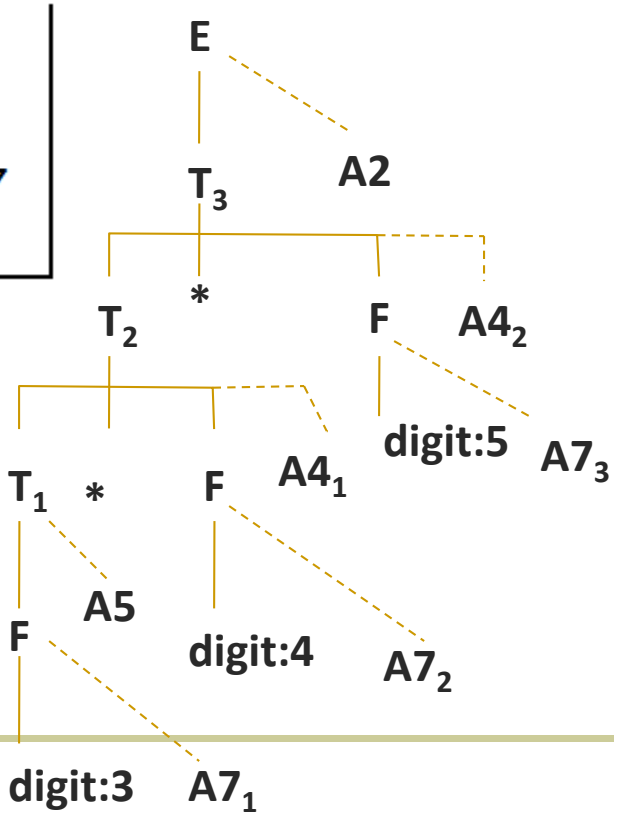
- 语法制导的翻译方案（SDT）是在产生式体中嵌入程序片断（语义动作）的上下文无关文法

- 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的结点
  - 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动作

在语法树上实现SDT

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

- 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



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
- 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

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

判断SDT可否用特定分析技术实现例子

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

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

后缀翻译方案

- 后缀SDT：所有动作都在产生式最右端的SDT

- 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

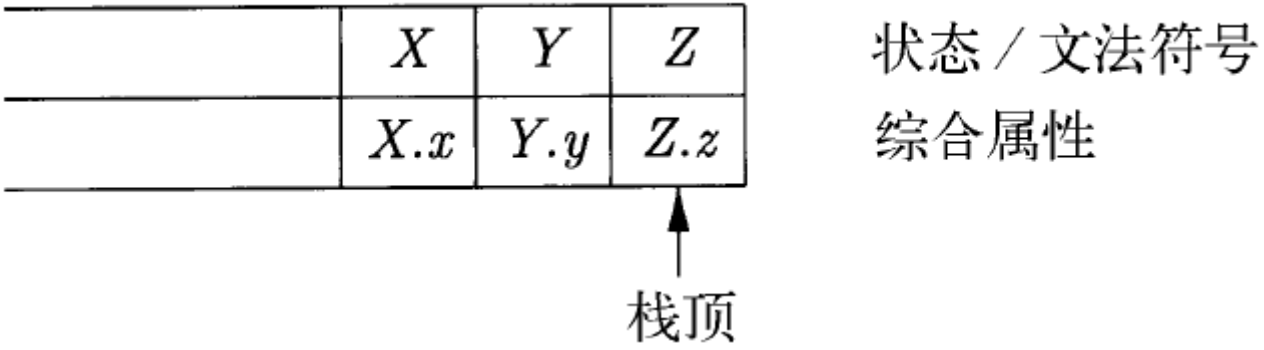
L	$\rightarrow$	E **n**	{ print($E.val$); }
E	$\rightarrow$	$E_1 + T$	{ $E.val = E_1.val + T.val$; }
E	$\rightarrow$	T	{ $E.val = T.val$; }
T	$\rightarrow$	$T_1 * F$	{ $T.val = T_1.val \times F.val$; }
T	$\rightarrow$	F	{ $T.val = F.val$; }
F	$\rightarrow$	(E)	{ $F.val = E.val$; }
F	$\rightarrow$	**digit**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

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

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



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

后缀SDT的栈实现

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

注意：stack[top-i]和文法符号的对应

- 这个SDT中没有局部变量，不会产生和局部变量有关的问题

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
  - 后缀SDT以及L属性对应的SDT可以在分析时完成

对于一般的SDT，可以先建立分析树（语义动作作为虚拟的结点），然后进行前序遍历并执行动作

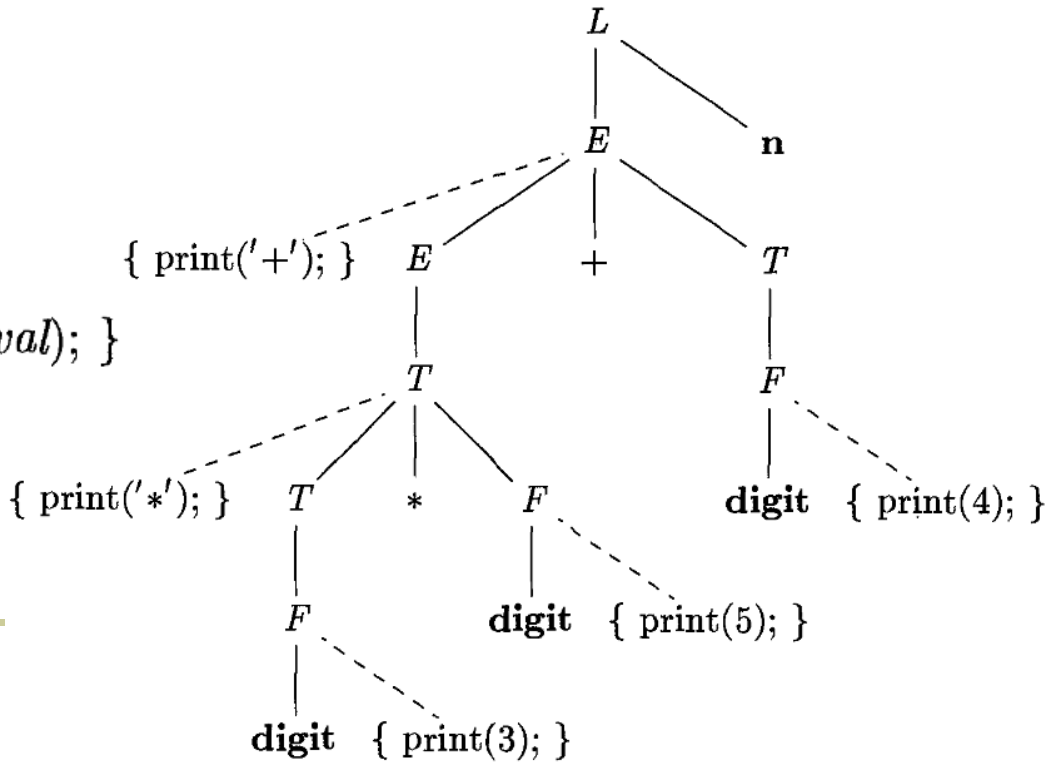
1) $L \rightarrow E\ \mathbf{n}$
2) $E \rightarrow \{\ \text{print}('+');\ \}\ E_1 + T$
3) $E \rightarrow T$
4) $T \rightarrow \{\ \text{print}('*');\ \}\ T_1 * F$
5) $T \rightarrow F$
6) $F \rightarrow (\ E\ )$
7) $F \rightarrow \mathbf{digit}\ \{\ \text{print}(\mathbf{digit}.lexval);\ \}$

移入/规约 冲突：

$M_2 \rightarrow \varepsilon$

$M_4 \rightarrow \varepsilon$

移入数字



消除左递归时SDT的转换

- 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
  - $R \rightarrow \varepsilon$

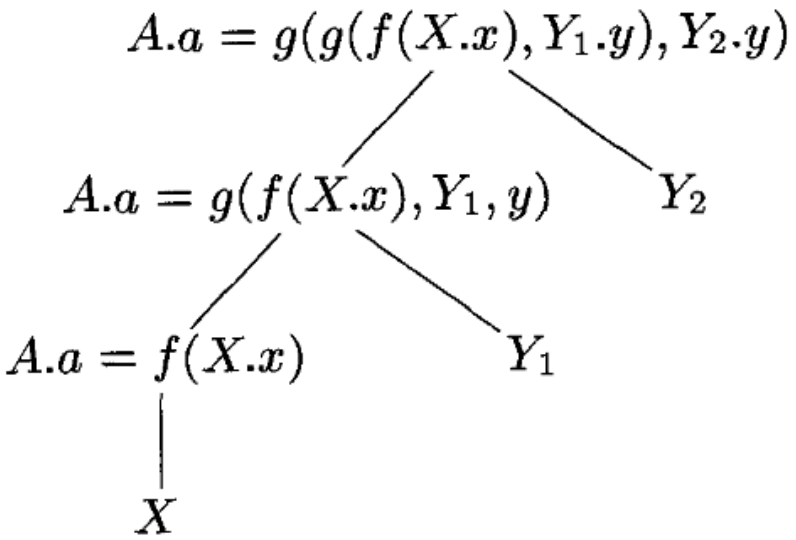
消除左递归时SDT的一般转换形式

- 假设只有单个递归产生式，且该左递归非终结符只有单个属性

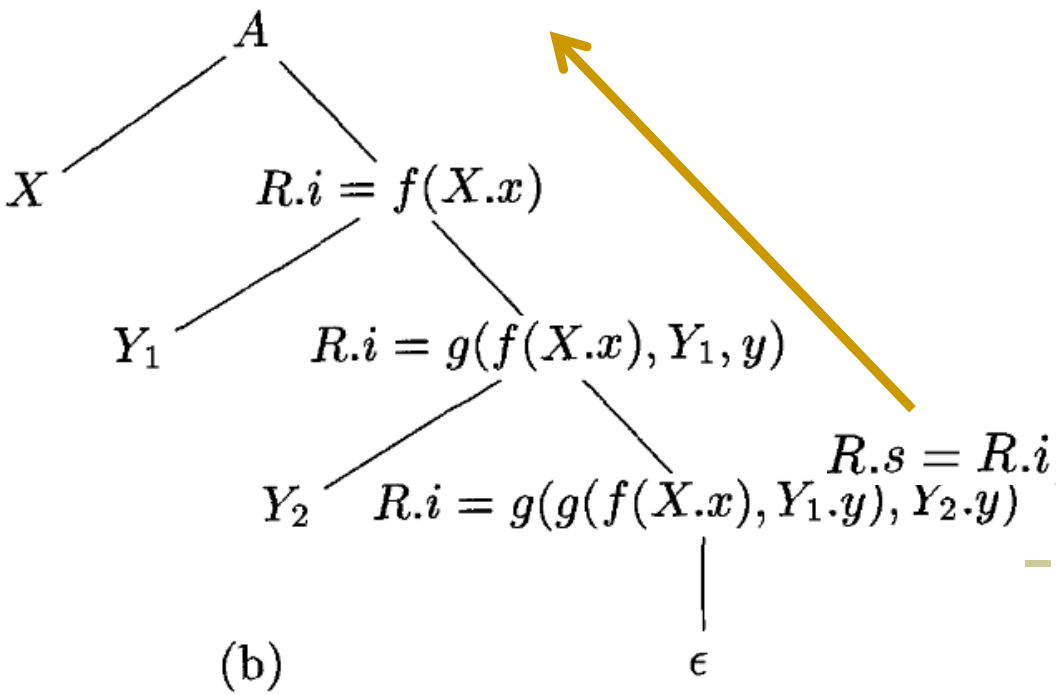
$$A \rightarrow A_1\ Y\ \{A.a = g(A_1.a, Y.y)\}$$
$$A \rightarrow X\ \{A.a = f(X.x)\}$$

- 消除左递归

$$A \rightarrow X\ R$$
$$R \rightarrow Y\ R \mid \epsilon$$



(a)



(b)

消除左递归时SDT的一般转换形式

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

左递归，语义属性计算过程的转化

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

L属性定义的SDT

- 将L属性的SDD转换为SDT
  - 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　继承属性需要在进入A的计算之前就得到
      - 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

L属性的SDT实例

- SDD

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

- 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

将SDD转换为SDT

考法：给出SDD，将动作放在合适的位置形成SDT

- 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
- 根据放置语义动作的规则得到如下SDT
  - (b)在C之前；(c)在S1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属性的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

递归下降法实现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();
                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 */
}
```

边扫描边生成属性（1）

- 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
- 可以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

边扫描边生成属性（2）

- 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
- 举例说明
  - 假设我们在扫描一个非终结符号对应的语法结构时，调用相应的函数，并生成主属性
  - $S \rightarrow$ while (C) S_1
    - S.code = Label || L1 ||C.code ||Label || L2 || S_1.code
  - 处理S时，先调用C，再调用S（对应于S_1）
  - 如果各个函数把主属性打印出来，我们处理while语句时，只需要先打印Label L1，再调用C（打印了C的代码），再打印Label L2，再调用S（打印S_1的代码）
  - 对于这个规则而言，只需要打印Label L1和Label L2，当然，我们要求C和S的语句在相应情况下跳转到L1和L2

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

- S → 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

- 前提是所有的非终结符号的SDT规则都这么做

边扫描边生成属性的例子

```
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 */
}
```

习题

$$
\begin{array}{lcl}
L & \rightarrow & E\ \mathbf{n} \\
E & \rightarrow & E_1 + T \\
E & \rightarrow & T \\
T & \rightarrow & T_1 * F \\
T & \rightarrow & F \\
F & \rightarrow & (\ E\ ) \\
F & \rightarrow & \mathbf{digit}
\end{array}
$$

1. 能否给出：
   - 求值的SDD
   - 将输入表达式翻译成前缀表达式的SDD
   - 将输入表达式翻译成后缀表达式的SDD？
   - 所给的SDD是S属性（或L属性）的吗？
2. 能否给出相应的SDT？
3. 所给出的SDT能够在LL或LR分析过程中实现吗？
4. 能否给出实现SDD的递归下降方法？