第5章　语法制导的翻译

本章继续2.3节的主题：使用上下文无关文法来引导对语言的翻译。本章讨论的翻译技术将在第6章中用于类型检查和中间代码生成。这些技术也可以用于实现那些完成特殊任务的小型语言。本章包含了一个有关排版的例子。

如2.3.2节所讨论的，我们把一些属性附加到代表语言构造的文法符号上，从而把信息和一个语言构造联系起来。语法制导定义通过与文法产生式相关的语义规则来描述属性的值。比如，一个从中缀表达式到后缀表达式的翻译器可能包含如下产生式和规则：

产生式	语义规则	
$E \to E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$	(5.1)

这个产生式有两个非终结符号 E 和 T，E_1 的下标用于区分 E 在产生式体中的出现和 E 在产生式头部的出现。E 和 T 都有一个字符串类型的属性 *code*。上面的语义规则指明字符串 $E.code$ 是通过将 $E_1.code$、$T.code$ 和字符 + 连接起来而得到的。虽然这个规则明确指出对 E 的翻译结果是根据 E_1、T 的翻译结果和“+”构造得到的，但直接通过字符串操作来实现这个翻译过程是很低效的。

根据2.3.5节的介绍可知，语法制导的翻译方案在产生式体中嵌入了称为语义动作的程序片段。比如

$$E \to E_1 + T \quad \{ \text{print } '+' \} \tag{5.2}$$

按照惯例，语义动作放在花括号之内。（对于作为文法符号出现的花括号，我们将用单引号把它们括起来，比如‘{’和‘}’。）一个语义动作在产生式体中的位置决定了这个动作的执行顺序。在产生式(5.2)中，语义动作出现在所有文法符号之后。一般情况下，语义动作可以出现在产生式体中的任何位置。

对于这两种标记方法，语法制导定义更加易读，因此更适合作为对翻译的规约。而翻译方案更加高效，因此更适合用于翻译的实现。

最通用的完成语法制导翻译的方法是先构造一棵语法分析树，然后通过访问这棵树的各个结点来计算结点的属性值。在很多情况下，翻译可以在扫描分析过程中完成，不需要构造出明确的语法分析树。因此，我们将研究一类称为“L属性翻译”（L代表从左到右）的语法制导翻译方案，这一类方案实际上包含了所有可以在语法分析过程中完成的翻译方案。我们还将研究一个较小的类别，称为“S属性翻译方案”（S代表综合），这类方案可以很容易地和自底向上语法分析过程联系起来。

5.1　语法制导定义

语法制导定义（Syntax-Directed Definition，SDD）是一个上下文无关文法和属性及规则的结合。属性和文法符号相关联，而规则和产生式相关联。如果 X 是一个符号而 a 是 X 的一个属性，那么我们用 $X.a$ 来表示 a 在某个标号为 X 的分析树结点上的值。如果我们使用记录或对象来实现这个语法分析树的结点，那么 X 的属性可以被实现为代表 X 的结点的记录的数据字段。属性可以有多种类型，比如数字、类型、表格引用或串。这些串甚至可能是很长的代码序列，比如编译器使用的中间语言的代码。

5.1.1 继承属性和综合属性

我们将处理非终结符号的两种属性：

1）综合属性(synthesized attribute)：在分析树结点 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。请注意，这个产生式的头一定是 A。结点 N 上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。

2）继承属性(inherited attribute)：在分析树结点 N 上的非终结符号 B 的继承属性是由 N 的父结点上的产生式所关联的语义规则来定义的。请注意，这个产生式的体中必然包含符号 B。结点 N 上的继承属性只能通过 N 的父结点、N 本身和 N 的兄弟结点上的属性值来定义。

另一种定义继承属性的方法

即使我们允许结点 N 上的一个继承属性 $B.c$ 通过 N 的子结点、N 本身、N 的父结点和兄弟结点上的属性值来定义，我们可以定义的翻译的种类并不会增加。这样的规则可以通过创建附加的 B 的属性，比如 $B.c_1$、$B.c_2$、… 来模拟。这些都是综合属性，用于把标号为 B 的结点的子结点上的属性拷贝过来。然后，我们使用属性 $B.c_1$、$B.c_2$、…来替换子结点属性，按照继承属性的方法计算得到 $B.c$。在实践中很少需要这种属性。

我们不允许结点 N 上的继承属性通过 N 的子结点上的属性值来定义，但是我们允许结点 N 上的一个综合属性通过结点 N 本身的继承属性来定义。

终结符号可以具有综合属性，但是不能有继承属性。终结符号的属性值是由词法分析器提供的词法值，在 SDD 中没有计算终结符号的属性值的语义规则。

例5.1 图 5-1 中的 SDD 基于我们熟悉的带有运算符 * 和 + 的算术表达式文法。它对一个以 **n** 作为结尾标记的表达式求值。在这个 SDD 中，每个非终结符号具有唯一的被称为 *val* 的综合属性。我们同时假设终结符号 **digit** 具有一个综合属性 **lexval**，它是由词法分析器返回的整数值。 □

产生式 1 $L \rightarrow E$ **n** 的规则将 $L.val$ 设置为 $E.val$。我们将看到，它就是整个表达式的值。

产生式 2 $E \rightarrow E_1 + T$ 也有一个规则。它计算出 E_1 和 T 的值的和，作为产生式头 E 的 *val* 属性的值。在任何标号为 E 的语法分析树结点 N 上，E 的 *val* 值是 N 的两个子结点(标号分别为 E 和 T)上的 *val* 值的和。

	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

图 5-1 一个简单的桌上计算器的语法制导定义

产生式 3 $E \rightarrow T$ 有唯一的规则，它定义了 E 的 *val* 值和对应于 T 的子结点的 *val* 值相同。产生式 4 和第二个产生式类似，它的规则将子结点的值相乘，而不是相加。产生式 5 和 6 的规则和第三个产生式的规则类似，它们拷贝子结点的值。产生式 7 给 $F.val$ 赋予一个 digit 的值，即由词法分析器返回的词法单元 **digit** 的数值。 □

一个只包含综合属性的 SDD 称为 *S 属性*(S-attribute)的 SDD，图 5-1 中的 SDD 就具有这个性质。在一个 S 属性的 SDD 中，每个规则都根据相应产生式的产生式体中的属性值来计算产生式头部非终结符号的一个属性。

为简单起见，本节中的语义规则没有副作用。在实践中，允许 SDD 具有一些副作用会带来一些方便。比如允许打印桌上计算器计算得到的结果，或者和一个符号表进行交互。等到在 5.2

节中讨论了属性的求值顺序之后，我们将允许语义规则计算任意的函数，这些函数可能会有副作用。

一个 S 属性的 SDD 可以和一个 LR 语法分析器一起自然地实现。实际上，图 5-1 中的 SDD 是图 4-58 中的 Yacc 程序的另一种表示，该程序演示了在 LR 语法分析过程中进行翻译的过程。两者的区别在于，Yacc 程序在产生式 1 的规则中通过副作用打印了 $E.val$ 的值，而不是定义属性 $L.val$。

一个没有副作用的 SDD 有时也称为*属性文法*(attribute grammar)。一个属性文法的规则仅仅通过其他属性值和常量值来定义一个属性值。

5.1.2 在语法分析树的结点上对 SDD 求值

在语法分析树上进行求值有助于将 SDD 所描述的翻译方案可视化，虽然翻译器实际上不需要构建语法分析树。因此，我们想象一下在应用一个 SDD 的规则之前首先构造出一棵语法分析树，然后再使用这些规则对这棵语法分析树上的各个结点上的所有属性进行求值。一个显示了它的各个属性的值的语法分析树称为*注释语法分析树*(annotated parse tree)。

我们如何构造一棵注释语法分析树呢？我们按照什么顺序来计算各个属性？在我们对一棵语法分析树的某个结点的一个属性进行求值之前，必须首先求出这个属性值所依赖的所有属性值。比如，如例 5.1 所示，所有的属性都是综合属性，那么在我们对一个结点上的 val 属性求值之前，必须求出该结点的所有子结点的属性 val 的值。

对于综合属性，我们可以按照任何自底向上的顺序计算它们的值，比如对语法分析树进行后序遍历的顺序。对于 S 属性定义的求值将在 5.2.3 节中讨论。

对于同时具有继承属性和综合属性的 SDD，不能保证有一个顺序来对各结点上的属性进行求值。比如，考虑非终结符号 A 和 B，它们分别具有综合属性 $A.s$ 和继承属性 $B.i$。同时它们的产生式和规则如下：

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

这些规则是循环定义的。不可能首先求出结点 N 上的 $A.s$ 或 N 的子结点上的 $B.i$ 中的一个的值，然后再求出另一个的值。一棵语法分析树的某个结点对上的 $A.s$ 和 $B.i$ 之间的循环依赖关系如图 5-2 所示。

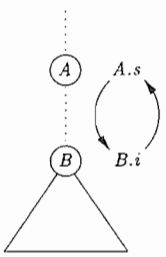

图 5-2 $A.s$ 和 $B.i$ 之间的循环依赖

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 的属性值之间具有循环依赖关系[⊖]。幸运的是，存在一个 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。我们将在 5.2 节中介绍这类 SDD。

例 5.2 图 5-3 显示了一个对应于输入串 $3 * 5 + 4\mathbf{n}$ 的注释语法分析树，该分析树是利用图 5-1 的文法和规则构造得到的。我们假定 $lexval$ 的值由词法分析器提供。对应于非终结符号的每个结点都有一个按自底向上顺序计算得到的 val 属性。在图中我们可以看到每个结点都关联了一个结果值。比如，在图中结点 $*$ 的父结点上，当计算得到它的第一和第三个子结点上的 $T.val = 3$ 和 $F.val = 5$ 之后，我们应用了相应的规则，指明 $T.val$ 就是这

⊖ 简单地讲，虽然这个问题是可判定的，但即使 $\mathcal{P}=\mathcal{NP}$ 成立，它也不可能使用多项式时间的算法来求解，因为它具有指数的时间复杂性。

两个值的乘积，即 15。 □

当一棵语法分析树的结构和源代码的抽象语法不“匹配”时，继承属性是很有用的。因为文法不是为了翻译而定义的，而是以语法分析为目的进行定义的，因此可能会产生这种不匹配的情况。下面的例子显示了如何使用继承属性来解决这个问题。

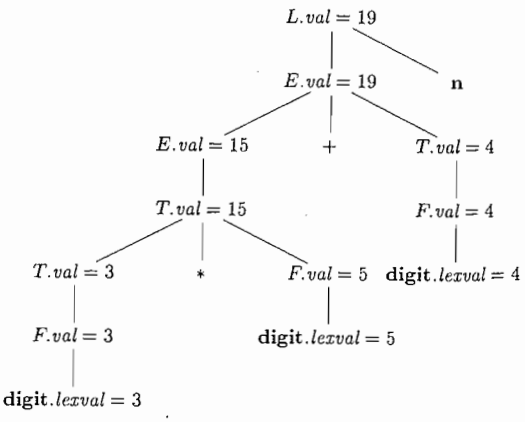

图 5-3 3 * 5 + 4**n** 的注释语法分析树

例 5.3 图 5-4 中的 SDD 计算诸如 3 * 5 和 3 * 5 * 7 这样的项。处理输入 3 * 5 的自顶向下语法分析过程首先使用了产生式 $T \rightarrow FT'$。这里，F 生成了数位 3，但是运算符 * 由 T' 生成。因此，左运算分量 3 和运算符 * 位于不同的子树中。我们将使用一个继承属性来把这个运算分量传递给运算符 *。

这个例子中的文法摘自常见的表达式文法的无左递归版本，我们在 4.4 节中使用这个文法作为说明自顶向下语法分析的例子。

非终结符号 T 和 F 各自有一个综合属性 *val*，终结符号 **digit** 有一个综合属性 *lexval*。非终结符号 T' 具有两个属性：继承属性 *inh* 和综合属性 *syn*。

这些语义规则基于如下思想：运算符 * 的左运算分量是通过继承得到的。更准确地说，产生式 $T' \rightarrow * TF'_1$ 的头 T' 继承了产生式体中 * 的左运算分量。给定一个项 $x * y * z$，对应于 $* y * z$ 的子树的根结点继承了 x 的值。对应于 $* z$ 的子树的根结点继承了 $x * y$ 的值。如果项中还有更多的因子，我们可以继续这样的处理过程。当所有的因子都处理完毕后，这个结果就通过综合属性向上传递到树的根部。

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*

图 5-4 一个基于适用于自顶向下语法分析的文法的 SDD

为了了解如何使用这些语义规则，考虑图 5-5 中对应于 3 * 5 的注释语法分析树。这棵语法分析树中最左边的标号为 **digit** 的叶子结点具有属性值 *lexval* = 3，其中的 3 是由词法分析器提供的。它的父结点对应于产生式 4，即 $F \rightarrow$ **digit**。和这个产生式相关的唯一语义规则定义 *F. val* = **digit**. *lexval*，等于 3。

在根结点的第二个子结点上，继承属性$T'.inh$根据和产生式 1 关联的语义规则$T'.inh = F.val$定义。因此，运算符 * 的左运算分量 3 从根结点的左子结点传递到右子结点。

对应于 T'的结点的产生式是 $T' \rightarrow * FT'_1$。(我们保留了注释语法分析树中的下标 1，以区分树中的两个 T'结点。)继承属性 $T'_1.inh$ 是由语义规则 $T'_1.inh = T'.inh \times F.val$ 定义的，这个规则和产生式 2 相关联。

已知 $T'.inh = 3$ 且 $F.val = 5$，我们得到 $T'_1.inh = 15$。在层次较低的 T'_1 结点上的产生式是 $T' \rightarrow \epsilon$。相应的语义规则 $T'.syn = T'.inh$ 定义了 $T'_1.syn = 15$。各个 T'结点上的属性 syn 将值 15 沿着树向上传递到 T 结点，使得 $T.val = 15$。□

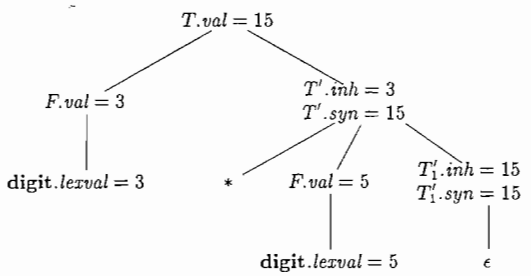

图 5-5 3 * 5 的注释语法分析树

5.1.3 5.1 节的练习

练习 5.1.1：对于图 5-1 中的 SDD，给出下列表达式对应的注释语法分析树：

1) (3 + 4) * (5 + 6)**n**

2) 1 * 2 * 3 * (4 + 5)**n**

3) (9 + 8 * (7 + 6) + 5) * 4**n**

练习 5.1.2：扩展图 5-4 中的 SDD，使它可以像图 5-1 所示的那样处理表达式。

练习 5.1.3：使用你在练习 5.1.2 中得到的 SDD，重复练习 5.1.1。

5.2 SDD 的求值顺序

依赖图(dependency graph)是一个有用的工具，它可以确定一棵给定的语法分析树中各个属性实例的求值顺序。注释语法分析树显示了各个属性的值，而依赖图可以帮助我们确定如何计算这些值。

在本节中，除了依赖图，我们还定义了两类重要的 SDD："S 属性"SDD 和更加通用的"L 属性"SDD。使用这两类 SDD 描述的翻译方案可以和我们已经研究过的语法分析方法很好地结合在一起。并且在实践中遇到的大部分翻译方案可以按照这两类 SDD 中的至少一类的要求写出来。

5.2.1 依赖图

依赖图描述了某个语法分析树中的属性实例之间的信息流。从一个属性实例到另一个实例的边表示计算第二个属性实例时需要第一个属性实例的值。图中的边表示语义规则所蕴涵的约束。更详细地说：

- 对于每个语法分析树的结点，比如一个标号为文法符号 X 的结点，和 X 关联的每个属性都在依赖图中有一个结点。
- 假设和产生式 p 关联的语义规则通过 $X.c$ 的值定义了综合属性 $A.b$ 的值(这个规则定义 $A.b$ 时可能还用到了 $X.c$ 之外的其他属性)。那么，相应的依赖图中有一条从 $X.c$ 到 $A.b$ 的边。更准确地讲，在每个标号为 A 且应用了产生式 p 的结点 N 上，创建一条从该产生式体中的符号 X 的实例所对应的 N 的子结点上的属性 c 到 N 上的属性 b 的边。[⊖]

⊖ 因为一个结点 N 可能有多个标号为 X 的子结点，我们再次假设使用下标来区分同一个符号在这个产生式的不同位置上的多次使用。

- 假设和产生式 p 关联的一个语义规则通过 $X.a$ 的值定义了继承属性 $B.c$ 的值。那么，在相应的依赖图中有一条从 $X.a$ 到 $B.c$ 的边。对于每个标号为 B、对应于产生式 p 中的这个 B 的结点 N，创建一条从结点 M 上的属性 a 到 N 上的属性 c 的边。这里的 M 对应于这个 X。请注意，M 可以是 N 的父结点或者兄弟结点。

例 5.4 考虑下面的产生式和规则：

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

在每个标号为 E，且其子结点对应于这个产生式体的结点 N 上，N 上的综合属性 val 使用两个子结点(标号分别为 E 和 T)上的 val 值计算得到。因此，对于每个使用了这个产生式的语法分析树，该树的依赖图中有一部分如图 5-6 所示。作为惯例，我们将把语法分析树的边显示为虚线，而依赖图的边显示为实线。 □

例 5.5 一个完整的依赖图的例子如图 5-7 所示。这个依赖图的结点用数字 1 ~ 9 表示，对应于图 5-5 中的注释语法分析树中的各个属性。

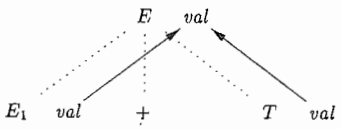

图 5-6 $E.val$ 由 $E_1.val$ 和 $T.val$ 综合得到

结点 1 和 2 表示和其标号为 **digit** 的两个叶子结点相关联的属性 $lexval$。结点 3 和 4 表示和其标号为 F 的两个结点相关联的属性 val。从结点 1 到结点 3 的边，以及从结点 2 到结点 4 的边是根据通过 SDD 中 **digit**.$lexval$ 定义 $F.val$ 的语义规则得到的。实际上，$F.val$ 等于 $digit.lexval$，但依赖图中的边表示的是依赖关系，而不是等于关系。

结点 5 和 6 表示和非终结符号 T' 的各次出现相关联的继承属性 $T'.inh$。从结点 3 到结点 5 的边是根据规则 $T'.inh = F.val$ 得到的，这个规则根据根的左子结点上的 $F.val$ 定义了右子结点上的 $T'.inh$。我们看到了从结点 5 到结点 6 的代表 $T'.inh$ 的边和从结点 4 到结点 5 的代表 $F.val$ 的边，因为这两个值相乘后得到了结点 6 上的属性 inh 的值。

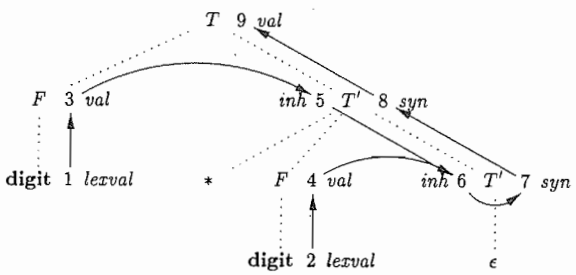

图 5-7 对应于图 5-5 中的注释语法分析树的依赖图

结点 7 和 8 表示了和 T'的各次出现相关联的综合属性 syn。从结点 6 到结点 7 的边是根据图 5-4 中的产生式 3 所关联的规则 $T'.syn = T'.inh$ 得到的。从结点 7 到结点 8 的边是根据产生式 2 所关联的语义规则得到的。

最后，结点 9 表示属性 $T.val$。从结点 8 到结点 9 的边是根据产生式 1 所关联的语义规则 $T.val = T'.syn$ 而得到的。 □

5.2.2 属性求值的顺序

依赖图刻画了对一棵语法分析树中不同结点上的属性求值时可能采取的顺序。如果依赖图中有一条从结点 M 到结点 N 的边，那么要先对 M 对应的属性求值，再对 N 对应的属性求值。因此，所有的可行求值顺序就是满足下列条件的结点顺序 N_1、N_2、…、N_k：如果有一条从结点 N_i 到 N_j 的依赖图的边，那么 $i < j$。这样的排序将一个有向图变成了一个线性排序，这个排序称为这个图的*拓扑排序*(topological sort)。

如果这个图中存在任意一个环，那么就不存在拓扑排序。也就是说，没有办法在这棵语法分析树上对相应的 SDD 求值。然而，如果图中没有环，那么总是至少存在一个拓扑排序。下面说明一下为什么会存在拓扑排序。因为没有环，所以我们一定能够找到一个没有边进入的结点。假设没有这样的结点，那么我们就可以不断地从一个前驱结点到达另一个前驱结点，直到我们回到某个已经访问过的结点，从而形成了一个环。令这个没有进入边的结点为拓扑排序的第一个结点，从依赖图中删除这个点，并对其余的结点重复上面的过程。（最终就可以得到一个拓扑排序——译者注。）

例 5.6 图 5-7 中的依赖图没有环。它的拓扑排序之一是这些结点的编码的顺序：1、2、…、9。请注意，这个图的每条边都是从编号较低的结点指向编号较高的结点，因此这个排序一定是拓扑排序。还有其他的拓扑排序，比如 1、3、5、2、4、6、7、8、9。 □

5.2.3 S 属性的定义

前面提到过，给定一个 SDD，很难判定是否存在一棵其依赖图包含环的语法分析树。在实践中，翻译过程可以使用某些特定类型的 SDD 来实现。这些类型的 SDD 一定有一个求值顺序，因为它们不允许产生带有环的依赖图。不仅如此，这一节中介绍的两类 SDD 可以和自顶向下及自底向上的语法分析过程一起高效地实现。

第一种 SDD 类型的定义如下：

- 如果一个 SDD 的每个属性都是综合属性，它就是 *S 属性的*。

例 5.7 图 5-1 中的 SDD 是一个 S 属性定义的例子。其中的每个属性（*L. val*、*E. val*、*T. val* 和 *F. val*）都是综合属性。 □

如果一个 SDD 是 S 属性的，我们可以按照语法分析树结点的任何自底向上顺序来计算它的各个属性值。对语法分析树进行后序遍历并对属性求值常常会非常简单，当遍历最后一次离开某个结点 N 时计算出 N 的各个属性值。也就是说，我们可以把下面定义的函数 *postorder* 应用到语法分析树的根上（见 2.3.4 节中的“前序遍历和后序遍历”部分）：

```
postorder(N)
{for(从左边开始,对 N 的每个子结点 C)postorder(C);
对 N 关联的各个属性求值;
}
```

S 属性的定义可以在自底向上语法分析的过程中实现，因为一个自底向上的语法分析过程对应于一次后序遍历。特别地，后序顺序精确地对应于一个 LR 分析器将一个产生式体归约成为它的头的过程。这个性质将在 5.4.2 节中用于 LR 语法分析过程中的综合属性求值工作，这些值将存放在分析栈中。这个过程不会显式地创建语法分析树的结点。

5.2.4 L 属性的定义

第二种 SDD 称为 L *属性定义*（L-attributed definition）。这类 SDD 的思想是在一个产生式体所关联的各个属性之间，依赖图的边总是从左到右，而不能从右到左（因此称为 L 属性的）。更准确地讲，每个属性必须要么是

- 一个综合属性，要么是
- 一个继承属性，但是它的规则具有如下限制。假设存在一个产生式 $A \rightarrow X_1 X_2 \ldots X_n$，并且有一个通过这个产生式所关联的规则计算得到的继承属性 $Xi.a$。那么这个规则只能使用：

1）和产生式头 A 关联的继承属性。

2）位于 X_i 左边的文法符号实例 X_1、X_2、…、X_{i-1} 相关的继承属性或者综合属性。

3）和这个 X_i 的实例本身相关的继承属性或综合属性，但是在由这个 X_i 的全部属性组成的依赖图中不存在环。

例 5.8 图 5-4 中的 SDD 是 L 属性的。要知道为什么，考虑对应于继承属性的语义规则。为方便起见，我们在这里再重复一下这些规则：

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

其中的第一个规则定义继承属性 $T'.inh$ 时只使用了 $F.val$，且 F 在相应产生式体中出现在 T' 的左部，因此满足 L 属性的要求。第二个规则定义 $T_1'.inh$ 时使用了和产生式头相关联的继承属性 $T'.inh$ 及 $F.val$，其中 F 在这个产生式体中出现在 T_1' 的左边。

从语法分析树的角度看，在每一种情况中，当这些规则被应用于某个结点时，它使用的信息“来自于上边或左边”的语法树结点，因此满足这一类 SDD 的要求。其余的属性是综合属性，因此这个 SDD 是 L 属性的。 □

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

第一个规则 $A.s = B.b$ 在 S 属性 SDD 或 L 属性 SDD 中都是一个合法的规则。它通过一个子结点（也就是产生式体中的一个符号）的属性定义了综合属性 $A.s$。

第二个规则定义了一个继承属性 $B.i$，因此整个 SDD 不可能是 S 属性的。不仅如此，虽然这个规则是合法的，这个 SDD 也不可能是 L 属性的，因为属性 $C.c$ 用来定义 $B.i$，并且 C 在产生式体中位于 B 的右边。虽然在 L 属性的 SDD 中可以使用语法分析树中的兄弟结点的属性，但这些结点必须位于被定义属性的符号的左边。 □

5.2.5 具有受控副作用的语义规则

在实践中，翻译过程会出现一些副作用：一个桌上计算器可能打印出一个结果；一个代码生成器可能把一个标识符的类型加入到符号表中。对于 SDD，我们在属性文法和翻译方案之间找到了一个平衡点。属性文法没有副作用，并支持任何与依赖图一致的求值顺序。翻译方案要求按从左到右的顺序求值，并允许语义动作包含任何程序片段。翻译方案将在 5.4 节中讨论。

我们将按照下面的方法之一来控制 SDD 中的副作用：

- 支持那些不会对属性求值产生约束的附带副作用。换句话说，如果按照依赖图的任何拓扑顺序进行属性求值时都可以产生“正确的”翻译结果，我们就允许副作用存在。这里的“正确”要视具体应用而定。
- 对允许的求值顺序添加约束，使得以任何允许的顺序求值都会产生相同的翻译结果。这些约束可以被看作隐含加入到依赖图中的边。

作为附带副作用的一个例子，让我们修改例 5.1 的桌上计算器，使它打印出计算结果。我们不使用规则 $L.val = E.val$，这个规则将结果保存到综合属性 $L.val$ 中。我们考虑：

产生式	语义规则
1) $L \rightarrow E\mathbf{n}$	$print(E.val)$

像 $print(E.val)$ 这样的语义规则的目的就是执行它们的副作用。它们将会被看作与相应产生式头相关的哑综合属性的定义。这个经过修改的 SDD 在任何拓扑顺序下都能产生相同的值，因为这个打印语句在结果被计算到 $E.val$ 中之后才会被执行。

例 5.10 图 5-8 中的 SDD 处理了简单的声明 D。该声明中包含一个基本类型 T，后跟一个标识符列表 L。T 的类型可以是 **int** 或 **float**。对于列表中的每个标识符，这个类型被录入到标识符的符号表条目中。我们假设录入一个标识符的类型不会影响其他标识符对应的符号表条目。这样，这些条目可以按照任何顺序进行更新。这个 SDD 不会检查一个标识符是否被声明了多次，我们也可以修改这个 SDD，使它能够对标识符声明次数进行检查。

非终结符号 D 表示了一个声明。根据产生式 1 可知，这个声明包含一个类型 T，后跟一个标识符的列表。T 有一个属性 $T.type$，它是声明 D 中的类型。非终结符号 L 也有一个属性，我们称它为 inh，以强调它是一个继承属性。$L.inh$ 的作用是将声明的类型沿着标识符列表向下传递，使得它可以被加入到相应的符号表条目中。

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

图 5-8 简单类型声明的语法制导定义

产生式 2 和产生式 3 都计算综合属性 $T.type$，为它赋予正确的值：integer 或 float。这个类型值在产生式 1 的规则中被传递给属性 $L.inh$。产生式 4 将 $L.inh$ 沿着语法分析树向下传递。也就是说，在一个分析树结点上，值 $L_1.inh$ 是通过拷贝该结点的父结点的 $L.inh$ 值而得到的，这个父结点对应于此产生式的头。

产生式 4 和产生式 5 还包含另一个规则。该规则用如下两个参数调用函数 addType：

- **id**. *entry*：在词法分析过程中得到的一个指向某个符号表对象的值。
- $L.inh$：被赋给列表中各个标识符的类型值。

我们假设函数 *addType* 正确地将 **id** 所代表的标识符的类型设置为类型值 $L.inh$。

输入串 **float** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$ 的依赖图如图 5-9 所示。数字 1 ~ 10 表示了这个依赖图中的结点。结点 1、2 和 3 表示了和各个标号为 **id** 的叶子结点相关的属性 *entry*。结点 6、8 和 10 是表示函数 *addType* 的应用于一个类型和这些 entry 值之一的哑属性。

结点 4 表示属性 $T.type$，它实际上是属性求值过程开始的地方。然后，这个类型被传递到结点 5、7 和 9。这些结点表示和非终结符号 L 的各次出现相关的 $L.inh$。 □

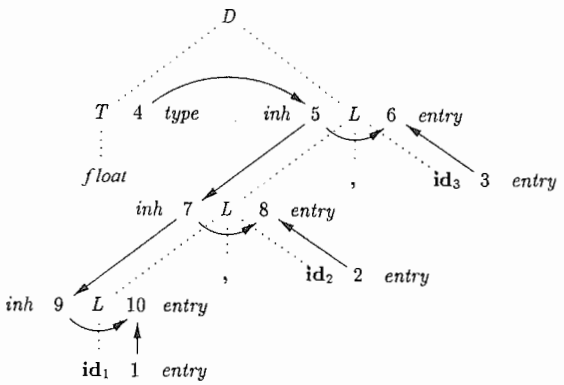

图 5-9 声明 **float** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$ 的依赖图

5.2.6 5.2 节的练习

练习 5.2.1：图 5-7 中的依赖图的全部拓扑排序有哪些？

练习 5.2.2：对于图 5-8 中的 SDD，给出下列表达式对应的注释语法分析树：

1) `int a, b, c`

2) `float w, x, y, z`

练习 5.2.3：假设我们有一个产生式 $A \rightarrow BCD$。A、B、C、D 这四个非终结符号都有两个属性：s 是一个综合属性，而 i 是一个继承属性。对于下面的每组规则，指出(i)这些规则是否满足 S 属性定义的要求。(ii)这些规则是否满足 L 属性定义的要求。(iii)是否存在和这些规则一致的求值过程？

1) $A.s = B.i + C.s$

2) $A.s = B.i + C.s$ 和 $D.i = A.i + B.s$

3) $A.s = B.s + D.s$

!4) $A.s = D.i$, $B.i = A.s + C.s$, $C.i = B.s$ 和 $D.i = B.i + C.i$

! 练习 5.2.4：这个文法生成了含“小数点”的二进制数：

$$S \to L \,.\, L \mid L$$
$$L \to L\ B \mid B$$
$$B \to 0 \mid 1$$

设计一个 L 属性的 SDD 来计算 *S.val*，即输入串的十进制数值。比如，串 101.11 应该被翻译为十进制数 5.635。提示：使用一个继承属性 *L.side* 来指明一个二进制位在小数点的哪一边。

!! 练习 5.2.5：为练习 5.2.4 中描述的文法和翻译设计一个 S 属性的 SDD。

!! 练习 5.2.6：使用一个自顶向下语法分析文法上的 L 属性 SDD 来实现算法 3.23。这个算法把一个正则表达式转换为一个不确定的有穷自动机。假设有一个表示任意字符的词法单元 **char**，并且 **char**. *lexval* 是它所表示的字符。你可以假设存在一个函数 *new*( )，该函数返回一个新的状态，也就是一个之前尚未被这个函数返回的状态。使用任何方便的表示方式来描述这个 NFA 的翻译。

5.3 语法制导翻译的应用

本章中的语法制导的翻译技术将在第 6 章中用于类型检查和中间代码生成。这里，我们将给出一些例子来解释有代表性的 SDD。

本节中的主要应用是抽象语法树的构造。因为有些编译器使用抽象语法树作为一种中间表示形式，所以一种常见的 SDD 形式将它的输入串转换为一棵树。为了完成到中间代码的翻译，编译器接下来可能使用一组规则来编译这棵语法树。这些规则实际上是一个建立于语法树之上的 SDD，而通常的 SDD 建立于语法分析树之上。（第 6 章将讨论应用一个 SDD 来生成中间代码的方法，这个方法不需要显式地生成树。）

我们考虑两个为表达式构造语法树的 SDD。第一个是一个 S 属性定义，它适合在自底向上语法分析过程中使用。第二个是一个 L 属性定义，它适合在自顶向下的语法分析过程中使用。

本节的最后一个例子是一个处理基本类型和数组类型的 L 属性定义。

5.3.1 抽象语法树的构造

2.8.2 节讨论过，一棵语法树中的每个结点代表一个程序构造，这个结点的子结点代表这个构造的有意义的组成部分。表示表达式 $E_1 + E_2$ 的语法树结点的标号为 +，且两个子结点分别代表子表达式 E_1 和 E_2。

我们将使用具有适当数量的字段的对象来实现一棵语法树的各个结点。每个对象将有一个 *op* 字段，也就是这个结点的标号。这些对象将具有如下所述的其他字段：

- 如果结点是一个叶子，那么对象将有一个附加的域来存放这个叶子结点的词法值。构造函数 *Leaf*(*op*, *val*) 创建一个叶子对象。我们也可以把结点看作记录，那么 *Leaf* 就会返回一个指向与叶子结点对应的新记录的指针。
- 如果结点是内部结点，那么它的附加字段的个数和该结点在语法树中的子结点个数相同。构造函数 *Node* 带有两个或多个参数：*Node*(*op*, c_1, c_2, …, c_k)，该函数创建一个对象，第一个字段的值为 *op*，其余 k 个字段的值为 $c_1, \cdots, c_k$。

例 5.11 图 5-10 中的 S 属性定义为一个简单的表达式文法构造出语法树。这个文法只包含二目运算符 + 和 −。通常，这两个运算符具有相同的优先级，并且都是左结合的。所有的非终结符号都有一个综合属性 *node*，该属性表示相应的抽象语法树结点。

每当使用第一个产生式 $E \rightarrow E_1 + T$ 时，它的语义规则就创建出一个结点。创建时使用“+”作为 *op*，使用 E_1. *node* 和 *T*. *node* 作为代表子表达式的两个子结点。第二个产生式也有类似的规则。

	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** $Leaf(\mathbf{id}, \mathbf{id}.entry)$
6)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf(\mathbf{num}, \mathbf{num}.val)$

图 5-10 为简单表达式构造语法树

产生式 3，即 $E \rightarrow T$，没有创建任何结点，因为 *E*. *node* 和 *T*. *node* 是一样的。类似地，产生式 4，即 $T \rightarrow (E)$，也没有创建任何结点。*T*. *node* 的值和 *E*. *node* 的值相同，因为括号仅仅用于分组。它们会影响语法分析树和抽象语法树的结构，但是一旦分组完成，就不需要在抽象语法树中保留这些括号了。

最后两个 *T*-产生式的右部是一个终结符号。我们使用构造函数 *Leaf* 来创建合适的结点。这些结点就成为 *T*. *node* 的值。

图 5-11 显示了为输入 $a - 4 + c$ 构造一棵抽象语法树的过程。这棵抽象语法树的结点被显示为记录。这些记录的第一个字段是 *op*。现在，抽象语法树的边用实线表示。基础的语法分析树使用点虚线表示边。实际上不需要生成语法分析树。第三种线是虚线，它表示 *E*. *node* 和 *T*. *node* 的值。每条线都指向适当的抽象语法树结点。

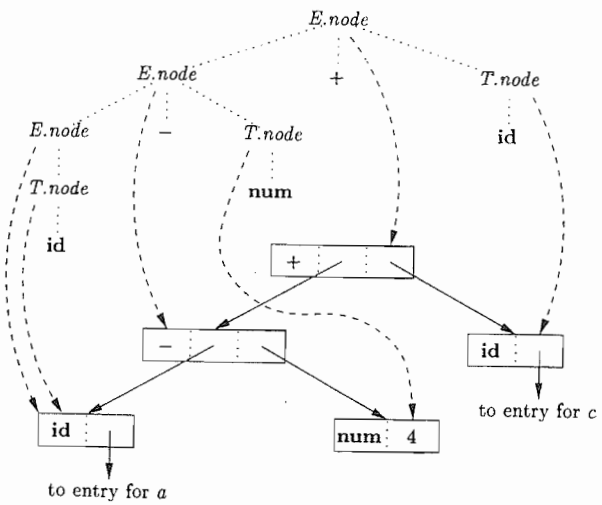

图 5-11 $a - 4 + c$ 的抽象语法树

在最底端，我们可以看到由 *Leaf* 构造得到的分别表示 *a*、4 和 *c* 的叶子结点。我们假设词法值 **id**. *entry* 指向符号表，并且词法值 **num**. *val* 是一个常量值。根据规则 5 和 6，这些叶子结点，或指向它们的指针，变成了图中的三个标号为 *T* 的语法分析树结点上的 *T*. *node* 的值。请注意，根据规则 3，指向 *a* 对应的叶子结点的指针同时也是语法分析树中最左边的 *E* 的 *E*. *node* 值。

我们根据规则 2 创建了一个结点，该结点的 *op* 字段等于减号，它的指针指向前两个叶子结

点。然后，规则 1 将对应于 - 的结点和第三个叶子组合起来，得到这个抽象语法树的根结点。

如果这些规则是在对语法分析树的后序遍历过程中求值的，或者是在自底向上分析过程中和归约动作一起进行求值的，那么当图 5-12 中显示的一系列步骤结束时，p_5 指向构造得到的抽象语法树的根结点。 □

```
1)  p1 = new Leaf(id, entry-a);
2)  p2 = new Leaf(num, 4);
3)  p3 = new Node('-', p1, p2);
4)  p4 = new Leaf(id, entry-c);
5)  p5 = new Node('+', p3, p4);
```

图 5-12 $a-4+c$ 的抽象语法树的构造步骤

如果使用一个为自顶向下语法分析而设计的文法，那么得到的抽象语法树仍然相同，其构造的步骤也相同，虽然语法分析树的结构和抽象语法树的结构有极大的不同。

例 5.12 图 5-13 中的 L 属性定义完成的翻译工作和图 5-10 中的 S 属性定义所完成工作的相同。文法符号 E、T、**id** 和 **num** 的属性和例 5-11 中讨论的相同。

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

图 5-13 在自顶向下语法分析过程中构造抽象语法树

这个例子中构造抽象语法树的规则和例 5.3 中桌上计算器的规则类似。在桌上计算器的例子中，项 $x*y$ 中的 x 和 $*y$ 位于语法分析树的不同部分，因此在计算 $x*y$ 时 x 是作为继承属性传递的。这里的思想是在构造 $x+y$ 的抽象语法树时将 x 作为一个继承属性传递，因为 x 和 $+y$ 出现在不同的子树中。非终结符号 E' 对应于例 5.3 中的非终结符号 T'。请比较一下图 5-14 中 $a-4+c$ 的依赖图和图 5-7 中 $3*4$ 的依赖图的相似之处。

非终结符号 E' 有一个继承属性 inh 和一个综合属性 syn。属性 $E'.inh$ 表示至今为止构造得到的部分抽象语法树。明确地说，它表示的是位于 E' 的子树左边的输入串前缀所对应的抽象语法树的根。在图 5-14 中依赖图的结点 5 处，$E'.inh$ 表示对应于 a 的抽象语法树的根，实际上就是对应于 a 的叶子结点。在结点 6 处，$E'.inh$ 表示对应于输入 $a-4$ 的部分抽象语法树的根。在结点 9 处，$E'.inh$ 表示 $a-4+c$ 的抽象语法树。

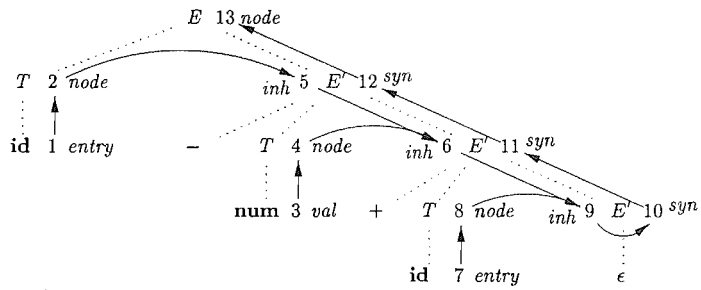

图 5-14 使用图 5-13 中的 SDD 时的 $a-4+c$ 的依赖图

因为没有更多的输入，所以在结点 9 处，$E'.inh$ 指向整个抽象语法树的根。属性 syn 把这个值沿着语法分析树向上传递，直到它成为 $E.node$ 的值。明确地讲，结点 10 上的属性值是通过产生式 $E' \rightarrow \epsilon$ 所关联的规则 $E'.syn = E'.inh$ 来定义的。在结点 11 处的属性值是通过图 5-13 中与产生式 2 相关的规则 $E'.syn = E'_1.syn$ 来定义的。类似的规则还定义了结点 12 和 13 处的值。 □

5.3.2 类型的结构

当语法分析树的结构和输入的抽象语法树的结构不同时，继承属性是很有用的。在这种情况下，继承属性可以用来将信息从语法分析树的一部分传递到另一部分。下一个例子显示了这种结构上的不匹配可能是由语言设计引起的，而不是由语法分析方法的约束引起的。

例 5.13 在 C 语言中，类型 **int** [2][3]可以读作："由两个数组组成的数组，子数组中有三个整数"。相应的类型表达式 $array(2, array(3, integer))$ 可以使用图 5-15 中的树来表示。运算符 $array$ 有两个参数，一个是数字，另一个是类型。如果使用树来表示类型，那么这个运算符返回一个标号为 $array$ 的结点，该结点具有两个子结点，分别表示数字和类型。

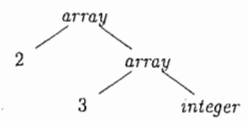

图 5-15 **int**[2][3] 的类型表达式

使用图 5-16 中的 SDD，非终结符号 T 生成的是一个基本类型或一个数组类型。非终结符号 B 生成基本类型 **int** 和 **float** 之一。当 T 推导出 BC 且 C 推导出 ϵ 时，T 生成一个基本类型。否则，C 就生成由一个整数序列组成的数组描述分量，其中的每个整数用方括号括起。

非终结符号 B 和 T 有一个表示类型的综合属性 t。非终结符号 C 有两个属性：一个继承属性 b 和一个综合属性 t。继承属性 b 将一个基本类型沿着树向下传播，而综合属性 t 则收集最终得到的结果。

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	$C.t = array(\text{num}.val, C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

图 5-16 T 生成一个基本类型或一个数组类型

输入串 **int**[2][3]的注释语法分析树如图 5-17 所示。图 5-15 中的相应类型表达式的构造过程如下：首先类型 $integer$ 从 B 开始，沿着 C 组成的链通过继承属性 b 向下传递。最后的数组类型是沿着 C 组成的链、通过属性 t 不断向上传递并综合而得到的。

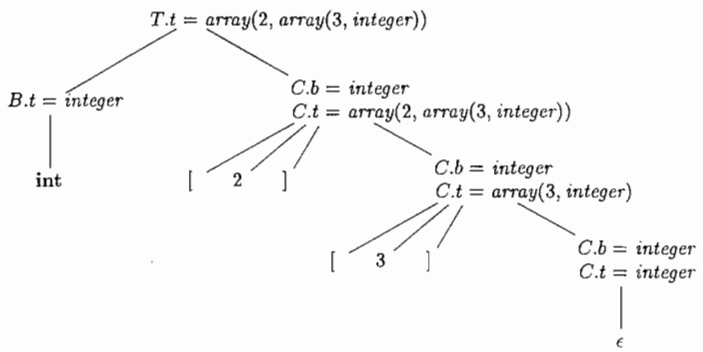

图 5-17 数组类型的语法制导的翻译

更详细地讲，在产生式 $T \rightarrow BC$ 对应的根结点上，非终结符号 C 使用继承属性 $C.b$ 从 B 那里继承类型。在最右边的 C 结点上的产生式是 $C \rightarrow \epsilon$，因此 $C.t$ 等于 $C.b$。产生式 $C \rightarrow [\ \textbf{num}\ ]\ C_1$ 的语义规则将运算符 $array$ 作用到运算分量 **num**.val 和 $C_1.t$ 上，得到 $C.t$ 的值。 □

5.3.3 5.3 节的练习

练习 5.3.1：下面是涉及运算符 + 和整数或浮点运算分量的表达式的文法。区分浮点数的方法是看它有无小数点。

$$E \rightarrow E + T \mid T$$
$$T \rightarrow \text{num} . \text{num} \mid \text{num}$$

1）给出一个 SDD 来确定每个项 T 和表达式 E 的类型。

2）扩展(a)中得到的 SDD，使得它可以把表达式转换成为后缀表达式。使用一个单目运算符 **intToFloat** 把一个整数转换为相等的浮点数。

！**练习 5.3.2**：给出一个 SDD，将一个带有 + 和 * 的中缀表达式翻译成没有冗余括号的表达式。比如，因为两个运算符都是左结合的，并且 * 的优先级高于 +，所以$((a*(b+c))*(d))$可翻译为 $a*(b+c)*d$。

！**练习 5.3.3**：给出一个 SDD 对 $x*(3*x+x*x)$这样的表达式求微分。表达式中涉及运算符 + 和 *、变量 x 和常量。假设不进行任何简化，也就是说，比如 $3*x$ 将被翻译为 $3*1+0*x$。

5.4 语法制导的翻译方案

语法制导的翻译方案是语法制导定义的一种补充。5.3 节中的所有语法制导定义的应用都可以使用语法制导的翻译方案来实现。

根据 2.3.5 节的介绍可知，*语法制导的翻译方案*(syntax-directed translation scheme，SDT)是在其产生式体中嵌入了程序片段的一个上下文无关文法。这些程序片段称为*语义动作*，它们可以出现在产生式体中的任何地方。按照惯例，我们在这些动作两边加上花括号。如果花括号要作为文法符号出现，则要给它们加上引号。

任何 SDT 都可以通过下面的方法实现：首先建立一棵语法分析树，然后按照从左到右的深度优先顺序来执行这些动作，也就是说在一个前序遍历过程中执行。5.4.3 节将给出一个这样的例子。

通常情况下，SDT 是在语法分析过程中实现的，不会真的构造一棵语法分析树。在本节中，我们主要关注如何使用 SDT 来实现两类重要的 SDD：

1）基本文法可以用 LR 技术分析，且 SDD 是 S 属性的。

2）基本文法可以用 LL 技术分析，且 SDD 是 L 属性的。

我们将会看到，在这两种情况下，一个 SDD 中的语义规则是如何被转换成为一个带有语义动作的 SDT 的。这些动作将在适当的时候执行。在语法分析过程中，产生式体中的一个动作在它左边的所有文法符号都被匹配之后立刻执行。

可以在语法分析过程中实现的 SDT 可以按照如下的方式识别：将每个内嵌的语义动作替换为一个独有的*标记非终结符号*(marker nonterminal)。每个标记非终结符号 M 只有一个产生式 $M \rightarrow \epsilon$。如果带有标记非终结符号的文法可以使用某个方法进行语法分析，那么这个 SDT 就可以在语法分析过程中实现。

5.4.1 后缀翻译方案

至今为止，最简单的实现 SDD 的情况是文法可以用自底向上方法来分析且该 SDD 是 S 属性定义。在这种情况下，我们可以构造出一个 SDT，其中的每个动作都放在产生式的最后，并且在按照这个产生式将产生式体归约为产生式头的时候执行这个动作。所有动作都在产生式最右端的 SDT 称为*后缀翻译方案*。

例 5.14 图 5-18 中的后缀 SDT 实现了图 5-1 中的桌上计算器的 SDD。其中只有一处改动：第

一个产生式的动作是打印出结果值。其余的语义动作和原来的语义规则对应的动作完全一样。因为此 SDD 的基本文法是 LR 的，并且这个 SDD 是 S 属性的，所以这些动作可以和语法分析器的归约步骤一起正确地执行。 □

5.4.2 后缀 SDT 的语法分析栈实现

后缀 SDT 可以在 LR 语法分析的过程中实现，当归约发生时执行相应的语义动作。各个文法符号的属性值可以放到栈中的某个位置，使得执行归约的时候可以找到它们。最好的方法是将属性和文法符号(或者表示文法符号的 LR 状态)一起放在栈中的记录里。

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**.$lexval$; }

图 5-18 实现桌上计算器的后缀 SDT

在图 5-19 中，语法分析栈包含的记录中有一个字段，该字段用于存放文法符号(或语法分析器的状态)，并且在这个字段之下有一个字段用于存放属性。三个文法符号 $X\ Y\ Z$ 位于栈的顶部，可能它们即将按照一个产生式，比如 $A \rightarrow X\ Y\ Z$，进行归约。这里，我们用 $X.x$ 表示 X 的一个属性，等等。一般来说，我们可以支持多个属性，方法是使记录变得足够大，或者在栈中的记录里放上指针。对于小型的属性，将记录变得足够大可能是比较简单的方法，即使有些时候有些字段不会被用到也没有太大关系。然而，如果一个或多个属性的大小没有限制，比如它们是字符串，那么最好把一个指针放到栈记录的属性值中，并把实际的值存放在栈之外的某个比较大的共享存储区域中。

如果所有属性都是综合属性，并且所有动作都位于产生式的末端，那么我们可以在把产生式体归约成产生式头的时候计算各个属性的值。如果我们使用 $A \rightarrow XYZ$ 这样的产生式进行归约，那么此时 X、Y 和 Z 的所有属性值都是可用的，并且都位于已知的位置上，如图 5-19 所示。在这个动作之后，A 和它的属性都位于栈的顶端，即现在存放 X 的记录的位置上。

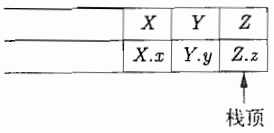

图 5-19 带有用于存放综合属性的字段的语法分析栈

例 5.15 让我们重写例 5.14 中桌上计算器 SDT 中的动作，使它们显式地操作语法分析栈。这样的栈操作通常是由语法分析器自动完成的。

假设语法分析栈存放在一个被称为 *stack* 的记录数组中，而 *top* 是指向栈顶的游标。这样，*stack*[*top*]指向这个栈的栈顶记录，*stack*[*top* − 1]指向栈顶记录的下一个记录，依此类推。我们还假设每个记录有一个被称为 *val* 的字段，该字段存放了这个记录所代表的文法符号的属性值。这样，我们可以使用 *stack*[*top* − 2]. *val* 来指向出现在栈中第三个位置上的属性 $E.val$。完整的 SDT 显示在图 5-20 中。

比如，在第二个产生式 $E \rightarrow E_1 + T$ 中，我们在栈顶之下两个位置上找到 E_1 的值，在栈顶找到 T 的值。求和的结果放在归约之后产生式头 E 将出现的位置上，也就是当前栈顶之下两个位置处。这是因为在归约之后，最上面的三个符号将被替换为一个符号。在计算完 $E.val$ 之后，我们将两个符号弹出栈，现在我们放置 $E.val$ 的记录将变成栈顶。

在第三个产生式 $E \rightarrow T$ 中不需要任何语义动作，因为栈的长度没有改变，栈顶的 $T.val$ 值直接变成了 $E.val$ 的值。产生式 $T \rightarrow F$ 和 $F \rightarrow$ **digit** 的情况与此类似。产生式 $F \rightarrow (E)$ 稍有不同。虽然值没有改变，但是在归约过程中消除了栈中的两个位置，因此这个值必须移动到归约之后的位置上。

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

图 5-20 在一个自底向上语法分析栈中实现桌上计算器

请注意，我们省略了针对栈中记录的第一个字段的操作步骤。这个字段保存了 LR 状态或文法符号。如果我们执行 LR 语法分析过程，语法分析表将给出每次归约之后的新状态，见算法 4.44。因此，我们可以直接把这个新状态放到新的栈顶记录中。 □

5.4.3 产生式内部带有语义动作的 SDT

动作可以放置在产生式体中的任何位置上。当一个动作左边的所有符号都被处理过后，该动作立刻执行。因此，如果我们有一个产生式 $B \rightarrow X\{a\}Y$，那么当我们识别到 X（如果 X 是终结符号）或者所有从 X 推导出的终结符号（如果 X 是非终结符号）之后，动作 a 就会执行。更准确地讲，

- 如果语法分析过程是自底向上的，那么我们在 X 的此次出现位于语法分析栈的栈顶时，我们立刻执行动作 a。
- 如果语法分析过程是自顶向下的，那么我们在试图展开 Y 的本次出现（如果 Y 是非终结符号）或者在输入中检测 Y（如果 Y 是终结符号）之前执行语义动作 a。

可以在语法分析过程中实现的 SDT 包括后缀 SDT 和即将在 5.5 节中讨论的一类 SDT，这类 SDT 实现了 L 属性定义。不是所有的 SDT 都可以在语法分析过程中实现，下面我们就给出一个例子。

例 5.16 作为一个有问题的 SDT 的极端例子，假设我们将桌上计算器的例子改成一个可以打印输入表达式的前缀表示方式的 SDT，而不再对表达式进行求值。新 SDT 的产生式和动作显示在图 5-21 中。

```
1)  L  →  E n
2)  E  →  { print('+'); }  E1 + T
3)  E  →  T
4)  T  →  { print('*'); }  T1 * F
5)  T  →  F
6)  F  →  ( E )
7)  F  →  digit  { print(digit.lexval); }
```

图 5-21 在语法分析过程中完成中缀到前缀翻译的有问题的 SDT

遗憾的是，不可能在自顶向下或自底向上的语法分析过程中实现这个 SDT，因为语法分析程序必须在它还不知道出现在输入中的运算符号是 * 还是 + 的时候，就执行打印这些符号的操作。

在产生式 2 和 4 中分别使用标记非终结符号 M_2 和 M_4 来替代相应的动作，一个移入 - 归约语法分析器（见 4.5.3 节）在处理输入 digit（比如 3）的时候会因为不能确定是使用 $M_2 \rightarrow \epsilon$ 归约，使用 $M_4 \rightarrow \epsilon$ 归约，还是移入输入数字而产生一个冲突。 □

任何 SDT 都可以按照下列方法实现：

1）忽略语义动作，对输入进行语法分析，并产生一棵语法分析树。

2）然后检查每个内部结点 N，假设它的产生式是 $A\rightarrow\alpha$。将 α 中的各个动作当作 N 的附加子结点加入，使得 N 的子结点从左到右和 α 中的符号及动作完全一致。

3）对这棵语法树进行前序遍历（见 2.3.4 节），并且当访问到一个以某个动作为标号的结点时立刻执行这个动作。

比如，图 5-22 显示了带有插入动作的表达式 3 * 5 + 4 的语法分析树。如果我们按照前序次序来访问结点，我们就得到了这个表达式的前缀形式：+ * 354。

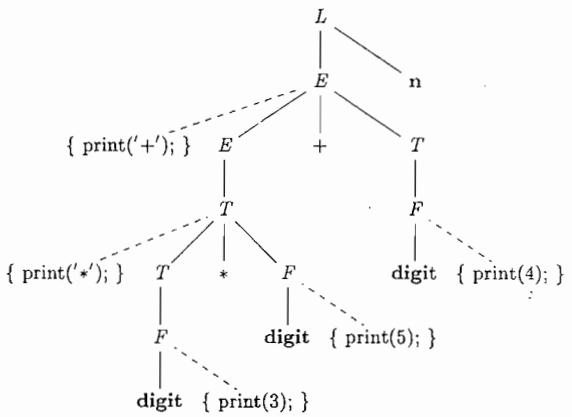

图 5-22 嵌入了动作的语法分析树

5.4.4 从 SDT 中消除左递归

因为带有左递归的文法不能按照自顶向下的方式确定地进行语法分析，所以在 4.3.3 节中介绍了左递归的消除。当文法是 SDT 的一部分时，我们还需要考虑如何处理其中的动作。

首先考虑简单的情况，即我们只需要关心一个 SDT 中的动作的执行顺序的情况。比如，如果每个动作只打印一个字符串，我们就只关心这些字符串的打印顺序。在这种情况下，可以应用下面的原则完成这个转化：

- 当转换文法的时候，将动作当成终结符号处理。

这个原则基于下面的思想：文法转换保持了由文法生成的符号串中终结符号的顺序。因此，这些动作在任何从左到右的语法分析过程中都按照相同的顺序执行，不管这个分析是自顶向下的还是自底向上的。

消除左递归的“技巧”是对两个产生式

$$A \rightarrow A\alpha \mid \beta$$

进行替换。这两个产生式生成的串包含一个 β 和任意数量的 α。它们将被替换为下面的产生式。新的产生式使用了一个新非终结符号 R（代表“其余部分”）来生成同样的串。

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned}$$

如果 β 不以 A 开头，那么 A 就不再有左递归的产生式。按照正则定义的表示法，在两组产生式中 A 都被定义为 β(α) *。在 4.3.3 节中可以看到如何处理 A 有多个递归或非递归产生式的情况。

例 5.17 考虑下面的 E 产生式。它们来自一个将中缀表达式翻译成后缀表达式的 SDT：

$$\begin{aligned} E &\rightarrow E_1 + T \quad \{ \text{print}('+'); \} \\ E &\rightarrow T \end{aligned}$$

如果我们对 E 应用标准的左递归消除转换，左递归产生式的余部为

$$\alpha \;=\; +\; T\;\{\;\text{print}('+');\;\}$$

而 β（即另一个产生式的体）是 T。如果我们引入 R 来表示 E 的余部，我们就得到如下的产生式集合：

$$\begin{array}{lcl} E & \rightarrow & T\;R \\ R & \rightarrow & +\;T\;\{\;\text{print}('+');\;\}\;R \\ R & \rightarrow & \epsilon \end{array}$$

□

当一个 SDD 的动作是计算属性的值，而不是仅仅是打印输出时，我们必须更加小心地考虑如何消除文法中的左递归。然而，如果这个 SDD 是 S 属性的，那么我们总是可以通过将计算属性值的动作放在新产生式中的适当位置上来构造出一个 SDT。

我们将给出一个通用的解决方案，以解决只有单个递归产生式、单个非递归产生式并且该左递归非终结符号只有单个属性的情况。将这个方案推广到多个递归/非递归产生式的情况并不困难，但是写起来非常麻烦。假设这两个产生式是：

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

这里 $A.a$ 是左递归非终结符号 A 的综合属性，而 X 和 Y 是单个文法符号，分别有综合属性 $X.x$ 和 $Y.y$。因为这个方案在递归的产生式中用任意的函数 g 来计算 $A.a$，而在第二个产生式中用任意函数 f 来计算 $A.a$ 的值，所以这两个符号可以代表由多个文法符号组成的串，每个符号都有自己的属性。在每种情况下，f 和 g 可以把它们能够访问的属性当作它们的参数，只要这个 *SDD* 是 *S* 属性的。

我们要把基础文法改成

$$\begin{array}{lcl} A & \rightarrow & X\;R \\ R & \rightarrow & Y\;R \mid \epsilon \end{array}$$

图 5-23 指出了在新文法上的 SDT 必须做的事情。在图 5-23a 中，我们看到的是原文法之上的后缀 SDT 的运行效果。我们将 f 应用一次，该次应用对应于产生式 $A \rightarrow X$ 的使用。然后我们应用函数 g，应用的次数和我们使用产生式 $A \rightarrow AY$ 的次数一样。因为 R 生成了 Y 的一个余部，它的翻译依赖于它左边的串，即一个形如 $XYY\cdots Y$ 的串。对产生式 $R \rightarrow YR$ 的每次使用都导致对 g 的一次应用。对于 R，我们使用一个继承属性 $R.i$ 来累计从 $A.a$ 的值开始不断应用 g 所得到的结果。

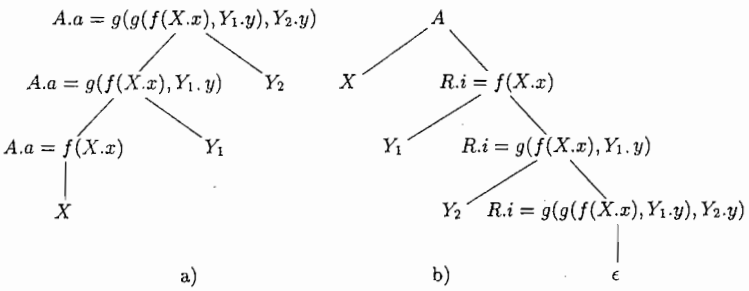

图 5-23 消除一个后缀 SDT 中的左递归

除此之外，R 还有一个没有在图 5-23 中显示的综合属性 $R.s$。当 R 不再生成文法符号 Y 时才开始计算这个属性的值，这个时间点是以产生式 $R \rightarrow \epsilon$ 的使用为标志的。然后 $R.s$ 沿着树向上拷贝，最后它就可以变成对应于整个表达式 $XYY\cdots Y$ 的 $A.a$ 的值。从 A 生成 XYY 的情况显示在图 5-23中，我们看到在图 5-23a 中的根结点上的 $A.a$ 的值使用了两次 g，而在图 5-23b 的底部的 $R.i$

也使用了两次 g，而正是这个结点上的 $R.s$ 的值被沿着树向上拷贝。

为了完成这个翻译，我们使用下列 SDT：

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

请注意，继承属性 $R.i$ 在产生式体中 R 的一次使用之前完成求值，而综合属性 $A.a$ 和 $R.s$ 在产生式的结尾完成求值。因此，计算这些属性时需要的任何值都已经在左边计算完成，变成了可用的值。

5.4.5 L 属性定义的 SDT

在 5.4.1 节，我们将 S 属性的 SDD 转换成为后缀 SDT，它的动作位于产生式的右端。只要基础文法是 LR 的，后缀 SDT 就可以按照自底向上的方式进行语法分析和翻译。

现在，我们考虑更加一般化的情况，即 L 属性的 SDD。我们假设基础文法将以自顶向下的方式进行语法分析，因为如果不是这样，那么翻译过程常常无法和一个 LL 或 LR 语法分析器一起完成。对于任何文法，我们只需要将动作附加到一棵语法分析树中，并在对这棵树进行前序遍历时执行这些动作，便可以实现下面的技术。

将一个 L 属性的 SDD 转换为一个 SDT 的规则如下：

1）把计算某个非终结符号 A 的继承属性的动作插入到产生式体中紧靠在 A 的本次出现之前的位置上。如果 A 的多个继承属性以无环的方式相互依赖，就需要对这些属性的求值动作进行排序，以便先计算需要的属性。

2）将计算一个产生式头的综合属性的动作放置在这个产生式体的最右端。

我们将使用两个例子来说明这些原则。第一个例子是关于排版的。它说明了如何将编译技术应用于其他的语言处理应用，编译技术的应用范围并不限于我们通常认为的程序设计语言。第二个例子是关于一个典型程序设计语言构造的中间代码生成的，这个构造是某种形式的 *while* 语句。

例 5.18 这个例子来自于数学公式排版语言。Eqn 是这种语言的早期例子，来自 Eqn 的思想仍然可以在 Tex 排版系统中找到，本书就是用 Tex 排版系统排版的。

我们将关注定义下标、下标的下标等排版能力，而忽略了上标、叠加的分数以及其他数学功能。在 Eqn 语言中，人们可以使用 `a sub i sub j` 来设定表达式 a_{i_j}。一个简单的 *boxes*（即由一个方框括起来的文本元素）的文法是：

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

对应于这四个产生式，一个方框可以是下列之一：

1）两个并列的方框，其中第一个方框 B_1 在另一个方框 B_2 的左边。

2）一个方框和一个下标方框。第二个方框的尺寸较小且位置较低，位于第一个方框的右边。

3）一个用括号括起来的方框，用于方框和下标的分组。Eqn 和 Tex 都使用花括号进行分组，但是我们将使用通常的圆括号来分组，以避免和 SDT 动作两边的括号混淆。

4）一个文本串，也就是任何字符串。

这个文法是二义性的，但是如果我们令下标和并列关系都是右结合的，并且令 **sub** 的优先级高于并列，那么我们仍然可以使用它来完成自底向上的语法分析。

表达式的排版过程就是由较小的方框构造出较大的方框的过程。在图 5-24 中，E_1 的方框和 $E_1.height$ 将被并列放置形成方框 E。而 E_1 的左边方框本身又是从 E 的方框和下标 1 的方框构造得到的。下标 1 的处理方法是将它的方框缩小大约 30%，并放在较低的位置上，然后把它

放在 E 的方框之后。虽然我们将把 *. height* 作为一个文本串进行处理，但它的方框中的长方形会说明它是如何从各个字母对应的方框构造得到的。

图 5-24 从较小的方框构造较大的方框

在这个例子中，我们只考虑这些方框的垂直方向的几何性质。水平方向的几何性质，即方框的宽度，也很有意思，当不同字符具有不同宽度时更是如此。可能看起来不是那么明显，但是图 5-24 中的各个字符确实具有不同的宽度。

和这些方框的垂直方向几何性质相关的值如下：

1）字体大小(point size)。它被用于在一个方框中设置文本。我们将假设不在下标中的字符被设置为 10 点，也就是一般书籍的字体大小。进一步，我们假设如果一个方框的字体大小是 p，那么它的下标方框的字体大小就是 $0.7p$。继承属性 $B.ps$ 表示块 B 的字体大小点数。这个属性必须是继承属性，因为一个给定的块的上下文决定了这个块在哪个下标层次，从而决定需要缩小多少。

2）每个方框有一个基线(baseline)，它是对应于文本行的底部的垂直位置，它不考虑像 g 这样的伸展到正常基线之下的字符。在图 5-24 中，点虚线就表示了方框 E、*. height* 以及整个表达式的基线。包含了下标 1 的方框的基线经过了调整，以便把这个下标放在较低位置。

3）每个方框有一个高度(height)，它是从方框顶部到方框基线的距离。综合属性 $B.ht$ 给出了方框 B 的高度。

4）每个方框有一个深度(depth)，它是从基线到达方框底部的距离。综合属性 $B.dp$ 给出了方框 B 的深度。

图 5-25 中的 SDD 给出了计算字体大小、高度和深度的规则。产生式 1 的功能是把初始值 10 赋给 $B.ps$。

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

图 5-25 方框排版的 SDD

产生式 2 处理并列的情况。字体大小被沿着语法分析树向下拷贝，也就是说，一个方框的两个子方框从这个较大的方框中继承了同样的字体大小点数。高度和深度是沿着语法分析树向上计算的，总是取两者的最大值。也就是说，大方框的高度是它的两个组成部分的高度的最大值，深度也按照类似的方法计算。

产生式 3 处理下标，它是最复杂的。在这个简化了的例子中，我们假设一个下标方框的字体大小是它的父方框的大小的 70%。实际情况会更加复杂，因为下标不可能无限缩小。在实践中，在几层下标之后，下标的大小就几乎不再缩小。另外我们还假设一个下标方框的基线向下移动了父方框的字体点数大小的 25%，同样，实际情况要更加复杂。

产生式 4 在使用括号的时候正确地拷贝各个属性。最后，产生式 5 处理表示文本方框的叶子结点。在这里，实际情况也是很复杂的，因此我们只显示了两个未定义的函数 *getHt* 和 *getDp*。它们检查各个字体的表格，以确定文本串中的全部字符的最大高度和最大深度。我们假设这个文本串中的字符是由终结符号 **text** 的属性 *lexval* 提供的。

最后一个任务是按照图 5-25 中处理 L 属性 SDD 的规则，将这个 SDD 转换为 SDT。正确的 SDT 显示在图 5-26 中。因为产生式的体比较长，为了增加可读性，我们把它们分割到多行中，并把动作对齐排列。因此，产生式体包含了到下一个产生式的头为止的多行内容。 □

	产生式		语义动作
1)	$S \rightarrow$	B	$\{\ B.ps = 10;\ \}$
2)	$B \rightarrow$		$\{B_1.ps = B.ps;\ \}$
		B_1	$\{B_2.ps = B.ps;\ \}$
		B_2	$\{\ B.ht = \max(B_1.ht, B_2.ht);$
			$\ B.dp = \max(B_1.dp, B_2.dp);\ \}$
3)	$B \rightarrow$		$\{B_1.ps = B.ps;\ \}$
		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)	$B \rightarrow$	(	$\{B_1.ps = B.ps;\ \}$
		B_1)	$\{\ B.ht = B_1.ht;$
			$\ B.dp = B_1.dp;\ \}$
5)	$B \rightarrow$	**text**	$\{\ B.ht = getHt(B.ps, \mathbf{text}.lexval);$
			$\ B.dp = getDp(B.ps, \mathbf{text}.lexval);\ \}$

图 5-26 方框排版的 SDT

我们的下一个例子是考虑一个简单的 while 语句，考虑如何为这种类型的语句生成中间代码。中间代码将被当作一个值为字符串的属性。稍后我们将探究一些高效的技术。这些技术在我们进行语法分析的时候顺序输出一个取值为字符串的属性的各个部分，从而避免了通过长字符串的拷贝来构造出更长的字符串。这个技术在例 5.17 中已经介绍过。在那个例子中，我们以“边扫描边生成”的方式生成了一个中缀表达式的后缀形式，而不是把表达式的后缀形式当作一个属性来计算。然而，在我们第一次表示中间代码生成时，我们通过字符串的连接来创建一个值为字符串的属性。

例 5.19 在这个例子中，我们只需要一个产生式：

$$S \rightarrow \mathbf{while}(C)S_1$$

这里，S 是生成各种语句的非终结符号，我们假设这些语句包括 if 语句、赋值语句和其他类型的

语句。在这个例子中，C 表示一个条件表达式——一个值为真或假的布尔表达式。

在这个关于语句控制流的例子中，我们只需要生成多个标号。我们假设其他的中间代码指令都由这个 SDT 的未显示部分生成。更明确地讲，我们生成显式的形如 **label** L 的指令，其中 L 是一个标识符。这个指令表明后一条指令的标号是 L。我们假设中间代码和 2.8.4 节中介绍的代码类似。

这个 while 语句的含义是首先对条件表达式 C 求值。如果它为真，控制就转向 S_1 的代码的开始处。如果 C 的值为假，那么控制就转向跟在这个 while 语句的代码之后的代码。我们必须设计 S_1 的代码，使得它在结束的时候能够跳转到这个 while 语句的代码的开始处。图 5-27 没有显示出跳转到对 C 求值的代码的开始处的指令。

我们使用下面的属性来生成正确的中间代码：

1）继承属性 *S. next* 是必须在 S 执行结束之后执行的代码的开始处的标号。

2）综合属性 *S. code* 是中间代码的序列，它实现了语句 S，并在最后有一条跳转到 *S. next* 的指令。

3）继承属性 *C. true* 是必须在 C 为真时执行的代码的开始处的标号。

4）继承属性 *C. false* 是必须在 C 为假时执行的代码的开始处的标号。

5）综合属性 *C. code* 是一个中间代码的序列，它实现了条件表达式 C，并根据 C 的值为真或假跳转到 *C. true* 或者 *C. false*。

计算 while 语句的这些属性的 SDD 显示在图 5-27 中。有几个要点需要解释一下：

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

图 5-27 while 语句的 SDD

- 函数 *new* 生成了新的标号。
- 变量 $L1$ 和 $L2$ 存放了在代码中需要的标号。$L1$ 表示这个 while 语句的代码的开始处，我们必须安排 S_1 在执行完毕之后跳转到这里。这就是我们把 S_1. *next* 设置为 $L1$ 的原因。$L2$ 是 S_1 的代码的开始处，它变成了 *C. true* 的值，因为在 C 为真时会跳转到那里。
- 请注意 *C. false* 被设置为 *S. next*，因为当条件为假时，就会执行 S 的代码之后的代码。
- 我们使用 $\|\|$ 作为连接各个中间代码片段的符号。因此，*S. code* 的值的以标号 $L1$ 开始，然后是条件表达式 C 的代码，然后是另一个标号 $L2$，然后是 S_1 的代码。

这个 SDD 是 L 属性的。当我们把它转换为 SDT 时，还需要考虑如何处理标号 $L1$ 和 $L2$，它们是变量而不是属性。如果我们把语义动作当作哑非终结符号来处理，那么这样的变量可以当作哑非终结符号的综合属性来处理。因为 $L1$ 和 $L2$ 不依赖于其他属性，它们可以被分配到产生式的第一个语义动作中。实现这个 L 属性定义的带有内嵌语义动作的 SDT 显示在图 5-28 中。 □

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

图 5-28 while 语句的 SDT

5.4.6 5.4 节的练习

练习 5.4.1：我们在 5.4.2 节中提到可能根据语法分析栈中的 LR 状态来推导出这个状态表示了什么文法符号。我们如何推导出这个信息？

练习 5.4.2：改写下面的 SDT：

$$A \rightarrow A\ \{a\}\ B \mid A\ B\ \{b\} \mid 0$$
$$B \rightarrow B\ \{c\}\ A \mid B\ A\ \{d\} \mid 1$$

使得基础文法变成非左递归的。这里，a、b、c 和 d 是语义动作，0 和 1 是终结符号。

！ **练习 5.4.3**：下面的 SDT 计算了一个由 0 和 1 组成的串的值。它把输入的符号串当作按照正二进制数来解释。

$$\begin{aligned} B &\rightarrow B_1\ 0\ \{B.val = 2 \times B_1.val\} \\ &\mid B_1\ 1\ \{B.val = 2 \times B_1.val + 1\} \\ &\mid 1\ \{B.val = 1\} \end{aligned}$$

改写这个 SDT，使得基础文法不再是左递归的，但仍然可以计算出整个输入串的相同的 $B.val$ 的值。

！ **练习 5.4.4**：为下面的产生式写出一个和例 5.10 类似的 L 属性 SDD。这里的每个产生式表示一个常见的 C 语言中那样的控制流结构。你可能需要生成一个三地址语句来跳转到某个标号 L，此时你可以生成语句 **goto** L。

1) $S \rightarrow$ **if** (C) S_1 **else** S_2

2) $S \rightarrow$ **do** S_1 **while** (C)

3) $S \rightarrow$ '{' L '}'; $L \rightarrow L\ S \mid \epsilon$

请注意，列表中的任何语句都可能包含一条从它的内部跳转到下一个语句的跳转指令，因此简单地为各个语句按顺序生成代码是不够的。

练习 5.4.5：按照例 5.19 的方法，把在练习 5.4.4 中得到的各个 SDD 转换成一个 SDT。

练习 5.4.6：修改图 5-25 中的 SDD，使它包含一个综合属性 $B.le$，即一个方框的长度。两个方框并列后得到的方框的长度是这两个方框的长度和。然后把你的新规则加入到图 5-26 中 SDT 的合适位置上。

练习 5.4.7：修改图 5-25 中的 SDD，使得它包含上标，用方框之间的运算符 **sup** 表示。如果方框 B_2 是方框 B_1 的一个上标，那么将 B_2 的基线放在 B_1 的基线上方，两条基线的距离是 0.6 乘以 B_1 的大小。把新的产生式和规则加入到图 5-26 的 SDT 中去。

5.5 实现 L 属性的 SDD

因为很多翻译应用可以用 L 属性定义来解决，所以我们将在这一节中详细地考虑它们的实现。下面的方法通过遍历语法分析树来完成翻译工作。

1) *建立语法分析树并注释*。这个方法对于任何非循环定义的 SDD 都有效。我们已经在 5.1.2 节中介绍了注释语法分析树。

2) *构造语法分析树，加入动作，并按照前序顺序执行这些动作*。这个方法可以处理任何 L 属性定义。我们在 5.4.5 节中讨论了如何把一个 L 属性 SDD 转变成为 SDT，还特别讨论了如何根据这样的 SDD 的语义规则把语义动作嵌入到产生式中。

在这一节，我们讨论下面的在语法分析过程中进行翻译的方法：

3) *使用一个递归下降的语法分析器*，它为每个非终结符号都建立一个函数。对应于非终结符号 A 的函数以参数的方式接收 A 的继承属性，并返回 A 的综合属性。

4）使用一个递归下降的语法分析器，以边扫描边生成的方式生成代码。

5）与 LL 语法分析器结合，实现一个 SDT。属性的值存放在语法分析栈中，而各个规则从栈中的已知位置获取需要的属性值。

6）与 LR 语法分析器结合，实现一个 SDT。这个方法会让人觉得惊讶，因为一个 L 属性 SDD 的 SDT 通常有一些动作位于产生式的中间，而在一个 LR 语法分析过程中，我们只有在构造出一个产生式体的全部符号之后才能肯定我们确实可以使用这个产生式。然而，我们将看到，如果基础文法是 LL 的，我们总是可以按照自底向上的方式来处理语法分析和翻译过程。

5.5.1 在递归下降语法分析过程中进行翻译

4.4.1 节讨论过，一个递归下降的语法分析器对每个非终结符号 A 都有一个函数 A。我们可以按照如下方法把这个语法分析器扩展为一个翻译器：

1）函数 A 的参数是非终结符号 A 的继承属性。

2）函数 A 的返回值是非终结符号 A 的综合属性的集合。

在函数 A 的函数体中，我们要进行语法分析并处理属性：

1）决定用哪一个产生式来展开 A。

2）当需要读入一个终结符号时，在输入中检查这些符号是否出现。我们假设分析过程不需要进行回溯，但是只要在出现语法错误时恢复输入位置，就可以把这个方法扩展到带回溯的递归下降语法分析技术，见 4.4.1 节中的讨论。

3）在局部变量中保存所有必要的属性值，这些值将用于计算产生式体中非终结符号的继承属性，或产生式头部的非终结符号的综合属性。

4）调用对应于被选定产生式体中的非终结符号的函数，向它们提供正确的参数。因为基础的 SDD 是 L 属性的，所以我们必然已经计算出了这些属性并且把它们存放到了局部变量中。

例 5.20 让我们考虑例 5.19 中 while 语句的 SDD 和 SDT。图 5-29 显示了函数 S 的相关部分的伪代码说明。

我们显示的这个函数 S 需要存储并返回很长的字符串。在实践中，更有效率的做法是让像 S 和 C 这样的函数返回一个指针，指向表示这些字符串的记录。那么，函数 S 中的返回语句将不会真的把各个组成部分连接起来，而是构造出一个记录或记录树。这个记录或记录树表示了将 *Scode*、*Ccode*、标号 *L1* 和 *L2* 以及文字串“`label`”的两次出现全部连接起来而得到的串。 □

```
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 */
}
```

图 5-29 用一个递归下降语法分析器实现 while 语句的翻译

例 5.21 现在我们将处理图 5-26 中用于方框排版的 SDT。我们首先处理语法分析问题，因为图 5-26 中的基础文法是二义性的。下面经过转换的文法使得并列运算和下标运算都是右结合的，而 **sub** 的优先级高于并列：

$$
\begin{array}{lcl}
S & \rightarrow & B \\
B & \rightarrow & T\ B_1 \mid T \\
T & \rightarrow & F\ \textbf{sub}\ T_1 \mid F \\
F & \rightarrow & (\ B\ ) \mid \textbf{text}
\end{array}
$$

引入两个非终结符号 T 和 F 的灵感来自于表达式中的项和因子。这里，由 F 生成的一个“因子”要么是一个括号中的方框，要么是一个文本串。由 T 生成的一个“项”是一个带有一系列下标的“因子”，而由 B 生成的一个方框是一个并列的“项”的序列。

T 和 F 的属性和 B 的属性一样，因为新的非终结符号也表示方框。引入它们的目的仅仅是为了帮助进行语法分析。因此，T 和 F 都有一个继承属性 ps 和综合属性 ht 及 dp。它们的语义动作可以从图 5-26 的 SDT 中修改得到。

这个文法还不可以直接进行自顶向下的语法分析，因为 B、T 的产生式都有相同的前缀。比如，考虑 T。一个自顶向下的语法分析器不能仅在输入中向前看一个符号就在 T 的两个产生式间做出决定。幸运的是，我们可以使用 4.3.4 节中讨论的提取左公因子的方法，使得这个文法可以进行自顶向下语法分析。处理 SDT 时，公共前缀的概念也被应用到语义动作中。T 的两个产生式都以非终结符号 F 开头，这个符号从 T 中继承了属性 ps。

图 5-30 中 $T(ps)$ 的伪代码中加入了 $F(ps)$ 的代码。对产生式 $T \rightarrow F$ **sub** $T_1 \mid F$ 应用提取左公因子的操作之后，只需要对 F 调用一次。这个伪代码显示了将该次调用替换为 F 的代码之后的结果。

```
(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);
}
```

图 5-30 递归下降的方框排板

B 的函数以 $T(10.0)$ 的方式调用函数 T，我们没有在这里显示这个调用。该次调用返回一个二元组，包括由非终结符号 T 生成的方框的高度和深度。在实践中，它将返回一个包含高度和深

度的记录。

函数 T 首先检查输入是否为左括号。如果是，它就必须处理产生式 $F \rightarrow (B)$。它保存了括号中 B 返回的任何值，但是如果 B 后面没有跟着一个右括号，那么就存在语法错误。处理这个语法错误的方式没有在这里显示。

否则，如果当前的输入是 **text**，那么函数 T 使用 *getHt* 和 *getDp* 来确定这个文本的高度和深度。

然后，函数 T 确定下一个方框是否为一个下标，如果是就调整 *point size*。我们使用和图 5-26 的产生式 $B \rightarrow B$ **sub** B 关联的语义动作来处理较大方框的高度和深度。否则，我们直接返回 F 所返回的值：$(h1, d1)$。 □

5.5.2 边扫描边生成代码

如例 5.20 所示，使用属性来表示代码并构造出很长的串不能满足我们的要求，原因是多方面的，比如拷贝和移动这些串字符时需要很长的时间。在通常情况下，比如在我们的代码生成例子中，我们可以通过执行一个 SDT 中的语义动作，逐步把各个代码片段添加到一个数组或输出文件中。要保证这项技术能够正确应用，下列要素必不可少：

1）存在一个（一个或多个非终结符号的）主属性。为方便起见，我们假设主属性都以字符串为值。在例 5.20 中，属性 *S.code* 和 *C.code* 是主属性，而其他属性不是主属性。

2）主属性是综合属性。

3）对主属性求值的规则保证：

① 主属性是将相关产生式体中的非终结符号的主属性值连接起来得到的。连接时也可能包括其他非主属性的元素，比如字符串 **label** 和标号 $L1$ 及 $L2$ 的值。

② 各个非终结符号的主属性值在连接运算中出现的顺序和这些非终结符号在产生式体中的出现顺序相同。

上面这些条件使得我们在构造主属性时只需要在适当的时候发出这个连接运算中的非主属性元素。我们可以依靠对一个产生式体中的非终结符号的对应函数的递归调用，以增量方式生成它们的主属性。

例 5.22 我们可以修改图 5-29 中的函数，使得它生成主属性 *S.code* 的各个元素，而不是把它们保存起来，再连接得到 *S.code* 的一个返回值。经过修改的函数 S 显示在图 5-31 中。

```
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 */
}
```

图 5-31 while 语句的 on-the-fly 的递归下降代码生成

在图 5-31 中，S 和 C 现在不返回任何值，因为它们唯一的综合属性是通过打印生成的。而且这些打印语句的位置很重要。打印输出的顺序是：首先是 `label` $L1$，然后是 C 的代码（它和图 5-29 中的 *Ccode* 的值相同），然后是 `label` $L2$，最后是对 S 的递归调用所生成的代码（它和图 5-29 中的 *Scode* 的值相同）。这样，对 S 的一次调用所打印的代码和图 5-29 中返回的 *Scode* 的值相同。

□

主属性的类型

我们的简单假设要求主属性具有字符串属性，这个限制实际上太严格了。真实要求是所有主属性的类型的值必须能够通过连接各个元素而构造得到。比如，任何类型的对象列表也可以作为主属性的类型，只要这些列表的表示方法允许我们把元素高效地加入到列表的尾部。因此，如果主属性的目的是表示一个中间代码语句的序列，我们就可以在一个对象数组的尾部不断写入语句，最终生成中间代码。当然，这个列表还需要满足 5.5.2 节中给出的其他要求。比如，一个主属性值必须由其他主属性值按照非终结符号的顺序连接得到。

我们附带地对基础 SDT 进行相同的修改：将一个主属性的构造转变为发出这个属性的元素的语义动作。在图 5-32 中，我们可以看到图 5-28 的 SDT 被修改成边扫描边生成代码的 SDT。

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

图 5-32 边扫描边生成 while 语句的代码的 SDT

5.5.3 L 属性的 SDD 和 LL 语法分析

假设一个 L 属性 SDD 的基础文法是一个 LL 文法，并且我们已经按照 5.4.5 节中描述的方法把它转换成一个 SDT，其语义动作被嵌入到各个产生式中。然后，我们就可以在 LL 语法分析过程中完成翻译过程，其中的语法分析栈需要进行扩展，以存放语义动作和属性求值所需的某些数据项。一般来说，这些数据项是属性值的拷贝。

除了那些代表终结符号和非终结符号的记录之外，语法分析栈中还将保存动作记录（action-record）和综合记录（synthesize-record），其中动作记录表示即将被执行的语义动作，而综合记录保存非终结符号的综合属性值。我们使用下列两个原则来管理栈中的属性：

- 非终结符号 A 的继承属性放在表示这个非终结符号的栈记录中。对这些属性求值的代码通常使用紧靠在 A 的栈记录之上的动作记录来表示。实际上，从 L 属性的 SDD 到 SDT 的转换方法保证了动作记录将紧靠在 A 的上面。
- 非终结符号 A 的综合属性放在一个单独的综合记录中，它在栈中紧靠在 A 的记录之下。

这个策略在语法分析栈中放置了多种类型的记录，这些不同的记录类型将被当作“栈记录”的子类进行正确管理。在实践中，我们可能把几个记录组合成一个记录，但是如果要解释这个方法的基本思想，最好还是把用于不同目的的数据分别存放在不同的记录中。

动作记录包含指向将被执行的动作代码的指针。动作也可能出现在综合记录中，这些动作通常把其他记录中的综合属性拷贝到栈中更低的位置上。在这个综合属性所在的记录被弹出栈之后，语法分析程序需要在这个较低的位置上找到该属性的值。

我们简单地看一下 LL 语法分析技术，以了解为什么需要建立属性的临时拷贝。根据 4.4.4

节的介绍可知，一个通过分析表驱动的 LL 语法分析器模拟了一个最左推导过程。如果 w 是至今为止已经匹配完成的输入，那么栈中就包含了一个文法符号序列 α，使得 $S \underset{lm}{\Rightarrow} w\alpha$，其中 S 是开始符号。当语法分析器按照一个产生式 $A \rightarrow B\ C$ 展开的时候，它把栈顶的 A 替换为 $B\ C$。

假设非终结符号 C 有一个继承属性 $C.i$。对于产生式 $A \rightarrow B\ C$，继承属性 $C.i$ 可能不仅仅依赖于 A 的继承属性，还可能依赖于 B 的所有属性。因此，我们可能需要在计算 $C.i$ 之前完成对 B 的处理。因此，我们需要计算 $C.i$ 所需的所有属性值的临时拷贝存放到计算 $C.i$ 的动作记录中。否则，当语法分析器把栈顶的 A 替换为 BC 的时候，A 的继承属性就和它的栈记录一起消失了。

因为基础 SDD 是 L 属性的，我们可以肯定当 A 位于栈顶时，A 的继承属性的值是可用的。因此当需要把这些值拷贝到对 C 的继承属性求值的动作记录中时，这些值也是可用的。不仅如此，用于存放 A 的综合属性的空间也不成问题，因为这个空间位于 A 的综合记录中，而这个记录在语法分析器使用 $A \rightarrow B\ C$ 进行展开时还保持在分析栈中（位于 B 和 C 之下）。

当处理 B 时，如果需要，我们可以（通过栈中紧靠在 B 之上的一个记录）执行一个动作，将它的继承属性拷贝给 C 使用。在处理完 B 之后，如果需要，B 的综合记录也可以拷贝它的综合属性供 C 使用。类似地，也可能需要一些临时变量来计算 A 的综合属性的值。这些值可以在先后处理 B 和 C 的时候被拷贝到 A 的综合记录中。所有这些属性的拷贝工作能够正确进行的原理是：

- 所有拷贝都发生在对某个非终结符号的一次展开时创建的不同记录之间。因此，这些记录中的每一个都知道其他各个记录在栈中离它有多远，因此可以安全地把值写到它下面的记录中。

下一个例子说明了通过不断地拷贝属性值，在 LL 语法分析过程中实现继承属性的方法。有可能存在一些捷径或者优化方法，对于那些只把一个属性值拷贝到另一个属性值的拷贝规则而言更是如此。我们要到例 5.24 中再说明这个问题，该例子还演示了对综合记录的处理方法。

例 5.23 这个例子实现了图 5-32 中的 SDT，该 SDT 边扫描边为 while 语句生成代码。这个 SDT 中除了表示标号的哑属性之外，没有综合属性。

图 5-33a 显示了我们即将使用 while 产生式来展开 S 的情况。这里假设我们已经知道输入的向前看符号就是 **while**。栈顶的记录对应于 S，它只包含继承属性 $S.next$。我们假设这个属性的值为 x。因为我们现在以自顶向下方式进行语法分析，所以按照惯例把栈顶显示在左边。

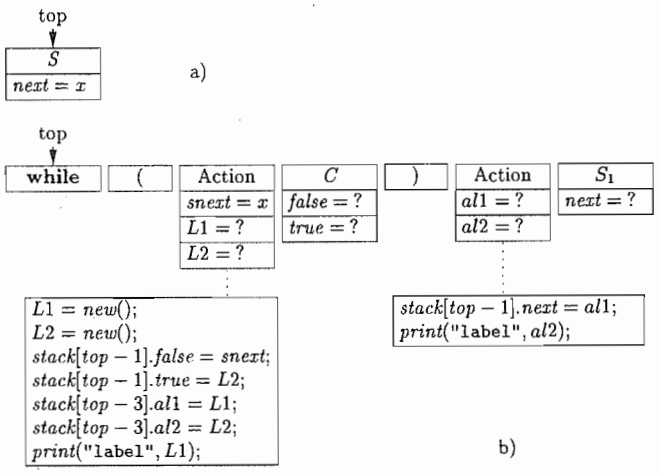

图 5-33　根据 while 语句的产生式扩展 S

图 5-33b 显示了我们展开 S 之后的情况。在非终结符号 C 和 S_1 之前存在动作记录，它们对应于图 5-32 中的基础 SDT 的语义动作。C 的记录包含了存放继承属性 *true* 和 *false* 的字段，而 S_1 的记录包含了存放属性 *next* 的字段。所有的 S 记录都必须包含这个字段。我们将这些字段的值显示为?，因为我们现在还不知道它们的值。

接下来，语法分析器识别了输入中的 **while** 和(，并将它们的记录弹出栈。现在，第一个动作位于栈顶，因此必须执行这个动作。这个动作记录有一个字段 *snext*，该字段存放了继承属性 *S. next* 的一个拷贝。当 S 被弹出栈的时候，*S. next* 的值被拷贝到字段 *snext* 中。在求 C 的继承属性值的时候将用到这个字段。第一个动作的代码生成了 $L1$ 和 $L2$ 的新值，我们分别将这两个值假设为 y 和 z。下一步是令 *C. true* 的值等于 z。我们把这个赋值语句写作 $stack[top-1].true = L2$ 是因为只有当这个动作记录位于栈顶时这个语句才会被执行，因此 $top-1$ 指向它下面的记录，即 C 的记录。

第一个动作记录将 $L1$ 拷贝到第二个动作记录的 $al1$ 字段中，在该处它将用于 $S_1.next$ 的求值。它也会将 $L2$ 拷贝到第二个动作记录中的 $al2$ 字段中，第二个动作需要这个值来正确打印输出。最后，第一个动作记录将 `label y` 打印到输出设备。

完成了第一个动作并将它的记录弹出栈之后的情形显示在图 5-34 中。在 C 的记录中的继承属性值都已经正确填写好，同时第二个动作记录中的临时变量 $al1$ 和 $al2$ 也已经填写好。此时 C 被展开，我们假设实现条件表达式 C 的包含了正确跳转到 x 和 z 的指令的代码已经生成。当 C 的记录被弹出栈时，)的记录变成了栈顶，使得语法分析器检查输入中的)。

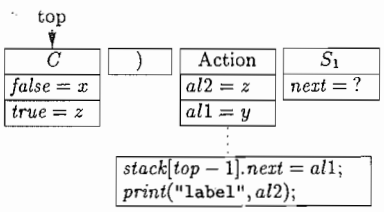

图 5-34 C 之上的动作被执行之后

当 S_1 之上的动作位于栈顶时，它的代码设置 $S_1.next$，并打印出 `label z`。上述工作完成之后，S_1 的记录成为栈顶。随着 S_1 被展开，假设它正确地生成了 S_1 的代码。不管 S_1 是什么类型的语句，生成的代码正确地实现了这个语句，随后跳转到 y。 □

例 5.24 现在让我们考虑同样的 while 语句，但是翻译方法把输出 *S. code* 作为一个综合属性，而不是通过边扫描边处理的方式生成。记住下面的不变式，或者说归纳假设，有助于理解接下来的解释。我们假设这些假设适用于每个非终结符号：

- 每个具有代码的非终结符号都把它的(字符串形式的)代码存放在栈中该符号的记录下方的综合记录中。

假设这个结论为真，我们处理 while 产生式时，将使它在处理完成后仍然成立，成为一个不变式。

图 5-35a 显示了使用 while 语句的产生式展开 S 之前的情形。我们在栈顶看到的是 S 的记录。和例 5.23 中一样，它有一个存放继承属性 *S. next* 的字段。紧靠在这个记录之下是 S 的本次出现的综合记录，它有一个存放 *S. code* 的字段。每个 S 的综合记录都包含这个字段。我们还显示了其他一些用于局部存储和动作的字段，因为图 5-28 中 while 产生式的 SDT 实际上是一个更大的 SDT 的一部分。

我们对 S 的展开是基于图 5-28 中的 SDT 的，展开的情形显示在图 5-35b 中。作为一种捷径，我们假设在展开过程中继承属性 *S. next* 被直接赋给 *C. false*，而不是先放到第一个动作中，然后再拷贝到 C 的记录中。

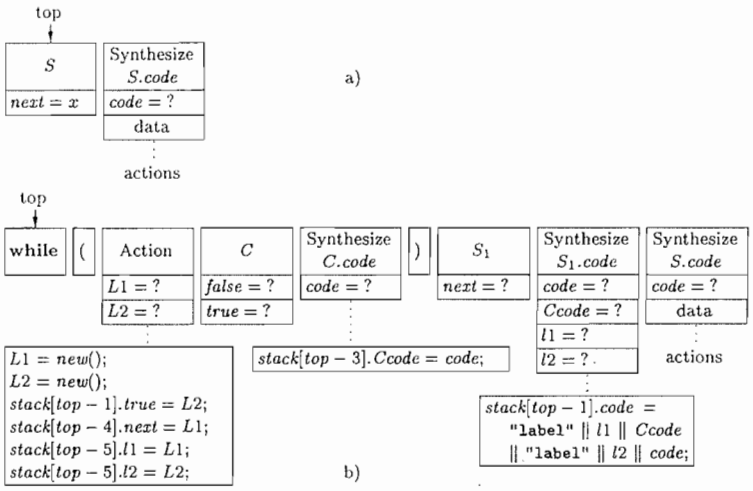

图 5-35　栈中构造的具有综合属性的 S 的扩展

我们看一下各个记录在变成栈顶的时候会做哪些事情。首先，**while** 记录使得词法单元 **while** 和输入匹配。这是一定会匹配的，否则我们就不会用这个产生式来展开 S。在 **while** 和(被弹出栈之后，执行动作记录中的代码。它生成了 $L1$ 和 $L2$ 的值，我们通过捷径直接把它们拷贝到需要它们的继承属性中，即 $S_1.next$ 和 $C.true$ 中。这个动作的最后两个步骤把 $L1$ 和 $L2$ 拷贝到被称为“Synthesize $S_1.code$”的记录中。

S_1 的综合记录有两个任务：它不仅仅要保存综合属性 $S_1.code$，它还要作为一个动作记录对整个产生式 $S\rightarrow$**while** (C) S_1 的属性求值。特别是，当它到达栈顶时，它将计算综合属性 $S.code$，并将这个值放到产生式头 S 的综合记录中。

当 C 成为栈顶的时候，它的两个继承属性都已经计算完成。根据上面给出的归纳假设，我们假设它正确地生成了代码，该代码执行了它的条件判断并跳转到正确的标号。我们同时假设在展开 C 时执行的动作正确地把这个代码放在了栈中下面的记录中，作为综合属性 $C.code$ 的值。

在 C 被弹出栈后，$C.code$ 的综合记录成为栈顶。它的代码要在 $S_1.code$ 的综合记录中使用，因为我们要在那里把所有的代码元素连接起来得到 $S.code$。因此，$C.code$ 的综合记录中有一个语义动作把 $C.code$ 拷贝到 $S_1.code$ 的综合记录中。完成上述工作之后，词法单元)的记录到达栈顶，使得语法分析器检查输入中的)。假设这个测试成功，S_1 的记录变成栈顶。根据我们的归纳假设，这个非终结符号被展开。这次展开的最终效果是它的代码被正确构造出来，并被放到 S_1 的综合记录中存放 $code$ 的字段中。

现在，S_1 的综合记录的所有数据字段都已经填充完毕，因此当它变成栈顶时，该记录中的动作就可以被执行。这个动作使得标号和来自 $C.code$ 和 $S_1.code$ 的代码按照正确的顺序被连接到一起。得到的串放在栈中下面的记录中，也就是 S 的综合记录中。我们现在已经正确地计算出了 $S.code$，并且当 S 的综合记录变成栈顶时，该代码可以被放置到栈中更低层的另一个记录中，在那里它最终会被组装到一个更大的代码串中，用于实现了包含这个 S 的更大的程序元素。 □

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 $A \rightarrow BC$,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 $A \rightarrow BC$ 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。

5.5.4 L 属性的 SDD 的自底向上语法分析

我们可以使用自底向上的方法来完成任何可以用自顶向下方式完成的翻译过程。更准确地说,给定一个以 LL 文法为基础的 L 属性 SDD,我们可以修改这个文法,并在 LR 语法分析过程中计算这个新文法之上的 SDD。这个"技巧"包括三个部分:

1) 以按照 5.4.5 节中的方法构造得到的 SDT 为起点。这样的 SDT 在各个非终结符号之前放置语义动作来计算它的继承属性,并且在产生式后端放置一个动作来计算综合属性。

2) 对每个内嵌的语义动作,向这个文法中引入一个标记非终结符号来替换它。每个这样的位置都有一个不同的标记,并且对于任意一个标记 M 都有一个产生式 $M \rightarrow \epsilon$。

3) 如果标记非终结符号 M 在某个产生式 $A \rightarrow \alpha\{a\}\beta$ 中替换了语义动作 a,对 a 进行修改得到 a',并且将 a' 关联到 $M \rightarrow \epsilon$ 上。这个动作 a'

① 将动作 a 需要的 A 或 α 中符号的任何属性作为 M 的继承属性进行拷贝。

② 按照 a 中的方法计算各个属性,但是将计算得到的这些属性作为 M 的综合属性。

这个变换看起来是非法的,因为通常和产生式 $M \rightarrow \epsilon$ 相关的动作将不得不访问某些没有出现在这个产生式中的文法符号的属性。然而,我们将在 LR 语法分析栈上实现各个语义动作。因此必要的属性总是可用的,它们位于栈顶之下的已知位置上。

例 5.25 假设一个 LL 文法中存在一个产生式 $A \rightarrow B\ C$,而继承属性 $B.i$ 是根据继承属性 $A.i$ 按照某个公式 $B.i = f(A.i)$ 计算得到的。也就是说,我们关心的 SDT 片段是

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

我们引入标记 M,M 有继承属性 $M.i$ 和综合属性 $M.s$。前者是 $A.i$ 的一个拷贝,而后者将成为 $B.i$。这个 SDT 将被写作

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

请注意,M 的规则中不可以使用 $A.i$,但是实际上我们将设法安排分析栈,使得如果即将进行一个到 A 的归约,那么 A 的每个继承属性都将出现在栈中执行这个归约的位置下方,从该处就可以读到这些继承属性。因此,当我们将 ϵ 归约为 M 时,我们直接在它的下方找到 $A.i$,在那里

读取到它的值。另外，$M.s$ 的值和 M 一起存放在栈中，它实际上是 $B.i$，以后在进行到 B 的归约时可以在下方找到这个值。 □

为什么标记能够正确工作?

标记是只能推导出 ϵ 的非终结符号，每个标记在所有产生式体中只出现一次。我们将正式证明如果一个文法是 LL 的，那么标记非终结符号可以被插入到产生式体中的任何位置，并且结果文法是 LR 的。如果文法是 LL 的，那么我们只需要看输入符号串 w 的第一个符号(如果 w 为空则是下一个符号)，就可以确定 w 是否可以从 A 开始，经过一个以产生式 $A\rightarrow\alpha$ 开头的推导序列得到。因此，如果我们用自底向上的方式对 w 进行语法分析，那么只要 w 的开头出现在输入中，我们就可以确定 w 的一个前缀首先必须被归约成为 α，然后再归约到 S。特别是，如果我们在 α 的任何位置插入标记，相应的 LR 状态将隐含地表明这个标记必定存在，并将在输入的正确位置上把 ϵ 归约为标记。

例 5.26 本例中我们把图 5-28 的 SDT 修改成基于经过修改的 LR 文法的 SDT，新的 SDT 可以和 LR 语法分析器一起完成翻译。我们在 C 之前引入标记 M，在 S_1 之前引入标记 N，因此基础文法变成

$$\begin{array}{lcl} S & \rightarrow & \textbf{while}\ (\ M\ C\ )\ N\ S_1 \\ M & \rightarrow & \epsilon \\ N & \rightarrow & \epsilon \end{array}$$

在我们讨论标记 M 及 N 的关联动作之前，先给出有关属性存放位置的“归纳假设”。

1) 在 while 产生式的整个产生式体之下(就是说在栈中的 **while** 之下)将是继承属性 $S.next$。我们可能不知道这个栈记录与哪个非终结符号或语法分析器状态相关，但是我们肯定该记录有一个字段存放了 $S.next$。这个字段位于该记录中的固定位置上，并且在我们知道 S 推导出什么短语之前就已经计算得到了 $S.next$。

2) 继承属性 $C.true$ 和 $C.false$ 将紧靠在 C 的栈记录的下方。因为假设这个文法是 LL 的，输入中出现的 **while** 告诉我们 while 产生式是唯一可能被识别的产生式，因此我们可以肯定 M 将出现在栈中紧靠 C 的下方，而 M 的记录将保存 C 的这些继承属性。

3) 类似地，继承属性 $S_1.next$ 必定出现在栈中紧靠 S_1 的下方，因此我们把该属性放在 N 的记录中。

4) 综合属性 $C.code$ 将出现在 C 的记录中。我们期望在实践中这个记录中出现的是一个指向这个字符串(对象)的指针，而该字符串本身位于栈外。当有一个属性的值是很长的字符串时，我们总是这样处理。

5) 类似地，综合属性 $S_1.code$ 将出现在 S_1 的记录中。

现在我们跟踪一个 while 语句的语法分析过程。假设一个保存 $S.next$ 的记录出现在栈顶，并且下一个输入是终结符号 **while**。我们把这个终结符号移入栈中。此时识别出的产生式肯定是 while 产生式，因此 LR 语法分析器可以移入“(”并确定下一步把 ϵ 归约为 M。此时的栈显示在图 5-36 中。我们同时还在该图中显示了和 M 的归约相关联的动作。我们创建出 $L1$ 和 $L2$ 的值，它们被存放在 M 的记录的域中。同处这个记录还有 $C.true$ 和 $C.false$ 的域。这些属性必定在这个记录的第二和第三个域中。这是为了和可能在不同上下文中出现于 C 之下，且需要为 C 提供这些属性的其他栈记录保持一致。这个动作最后把两个值赋给 $C.true$ 和 $C.false$。其中的第一个值来自于刚刚生成的 $L2$，另一个则从栈下方存放 $S.next$ 的地方获取。

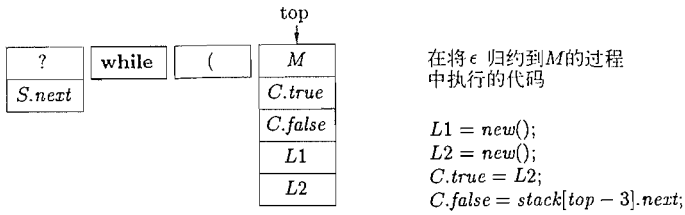

图 5-36 在将 ϵ 归约为 M 之后的 LR 语法分析栈

我们假设后面的输入被正确地归约为 C。因此，综合属性 $C.code$ 存放在 C 的记录中。这一次对栈的改变显示在图 5-37 中。该图还显示了接下来将被放到栈中的多个记录，它们将被放到 C 的记录之上。

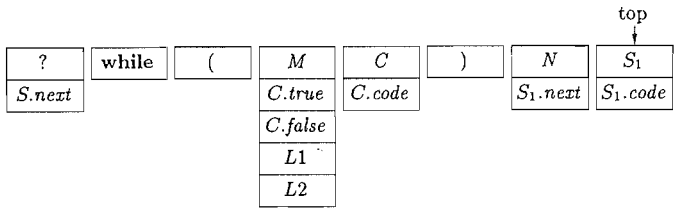

图 5-37 即将把 while 产生式的体归约为 S 之前的栈

继续识别 while 语句，语法分析器下一步将在输入中发现“)”，把它放在该符号自己的记录中，并压入栈中。因为文法是 LL 的，因此语法分析器在该点上已经知道它在处理一个 while 语句。语法分析器将把 ϵ 归约为 N。和 N 相关联的唯一数据是继承属性 $S_1.next$。请注意，需要将这个属性存放在此记录中的原因是这个记录将恰好位于 S_1 的记录之下。计算 $S_1.next$ 的值的代码是

$$S_1.next = stack[top - 3].L1;$$

这个动作从 N 之下三个记录的地方获取了 $L1$ 的值。当这个代码执行的时候，N 的记录位于栈顶。

接下来，语法分析器将其余输入的某个前缀归约成为 S。我们一直把它称为 S_1，以便和产生式头的 S 区分开。$S_1.code$ 的值计算完成并放在 S_1 的栈记录中。这个步骤对应于图 5-37 所示的情形。

此时，语法分析器将把从 **while** 到 S_1 的全部内容归约为 S。在这一次归约中，执行的代码是：

$$\begin{aligned} &tempCode = \textbf{label} \parallel stack[top - 4].L1 \parallel stack[top - 3].code \parallel \\ &\quad \textbf{label} \parallel stack[top - 4].L2 \parallel stack[top].code; \\ &top = top - 6; \\ &stack[top].code = tempCode; \end{aligned}$$

也就是说，我们在变量 $tempCode$ 中构造出 $S.code$ 的值。该代码也是由两个标号 $L1$ 和 $L2$、C 的代码和 S_1 的代码组成。这个栈执行了一些弹出操作，因此 S 出现在 **while** 原来出现的地方。S 的代码值存放在该记录的 $code$ 字段中。它在那里被解释为综合属性 $S.code$。请注意，我们在这次讨论中没有显示对 LR 状态的操作，实际上这些状态必须出现在栈中，其所在的字段就是存放文法符号的字段。 □

5.5.5 5.5 节的练习

练习 5.5.1：按照 5.5.1 节的风格，将练习 5.4.4 中得到的每个 SDD 实现为递归下降的语法

分析器。

练习 5.5.2：按照 5.5.2 节的风格，将练习 5.4.4 中得到的每个 SDD 实现为递归下降的语法分析器。

练习 5.5.3：按照 5.5.3 节的风格，将练习 5.4.4 中得到的每个 SDD 和一个 LL 语法分析器一起实现。它们应该边扫描输入边生成代码。

练习 5.5.4：按照 5.5.3 节的风格，将练习 5.4.4 中得到的每个 SDD 和一个 LL 语法分析器一起实现，但是代码(或者指向代码的指针)存放在栈中。

练习 5.5.5：按照 5.5.4 节的风格，将练习 5.4.4 中得到的每个 SDD 和一个 LR 语法分析器一起实现。

练习 5.5.6：按照 5.5.1 节的风格实现练习 5.2.4 中得到的 SDD。按照 5.5.2 节的风格得到的实现和这个实现相比有什么不同吗?

5.6 第 5 章总结

- *继承属性和综合属性*：语法制导的定义可以使用的两种属性。一棵语法分析树结点上的综合属性根据该结点的子结点的属性计算得到。一个结点上的继承属性根据它的父结点和/或兄弟结点的属性计算得到。
- *依赖图*：给定一棵语法分析树和一个 SDD，我们在各个语法分析树结点所关联的属性实例之间画上边，以指明位于边的头部的属性值要根据位于边的尾部的属性值计算得到。
- *循环定义*：在一个有问题的 SDD 中，我们发现存在一些语法分析树，无法找到一个顺序来计算所有结点上的所有属性的值。这些语法分析树关联的依赖图中存在环。确定一个 SDD 是否存在这种带环的依赖图是非常困难的。
- *S 属性定义*：在一个 S 属性的 SDD 中，所有的属性都是综合的。
- *L 属性定义*：在一个 L 属性的 SDD 中，属性可能是继承的，也可能是综合的。然而，一个语法分析树结点上的继承属性只能依赖于它的父结点的继承属性和位于它左边的兄弟结点的(任意)属性。
- *抽象语法树*：一棵抽象语法树中的每个结点代表一个构造；某个结点的子结点表示该结点所对应的构造的有意义的组成部分。
- *实现 S 属性的 SDD*：一个 S 属性定义可以通过一个所有动作都在产生式尾部的 SDT(后缀 SDT)来实现。这些动作通过产生式体中的各个符号的综合属性来计算产生式头的综合属性。如果基础文法是 LR 的，那么这个 SDT 可以在一个 LR 语法分析器的栈上实现。
- *从 SDT 中消除左递归*：如果一个 SDT 只有副作用(即不计算属性值)，那么消除文法左递归的标准方法允许我们把语义动作当作终结符号移动到新文法中去。在计算属性时，如果这个 SDT 是后缀 SDT，那么我们仍然能够消除左递归。
- *用递归下降语法分析实现 L 属性的 SDD*：如果我们有一个 L 属性定义，且其基础文法可以用自顶向下的方法进行语法分析，我们就可以构造出一个不带回溯的递归下降语法分析器来实现这个翻译。继承属性变成了非终结符号对应的函数的参数，而综合属性由该函数返回。
- *实现 LL 文法之上的 L 属性的 SDD*：每个以 LL 文法为基础文法的 L 属性定义可以在语法分析过程中实现。用于存放一个非终结符号的综合属性的记录被放在栈中这个非终结符号之下，而一个非终结符号的继承属性和这个非终结符号存放在一起。栈中还放置了动作记录，以便在适当的时候计算属性值。

- 以自底向上的方式实现一个在 LL 文法之上的 L 属性 SDD：一个以 LL 文法为基础文法的 L 属性定义可以转换成一个以 LR 文法为基础文法的翻译方案，且这个翻译可以和自底向上的语法分析过程一起执行。文法的转换过程中引入了“标记”非终结符号。这些符号出现在自底向上语法分析栈中，并保存了栈中位于它上方的非终结符号的继承属性。在栈中，综合属性和它的非终结符号放在一起。

5.7 第 5 章参考文献

语法制导定义是归纳定义的一种形式，它在语法结构上进行归纳。作为归纳定义，它们很早以前就已经在数学中非正式地使用了。它们在程序设计语言中的应用是和 Algol 60 中对文法的使用一起出现的。

调用语义动作的语法分析器的基本思想可以在 Samelson 和 Bauer[8]以及 Brooker 和 Morris[1]的工作中找到。Irons[2]使用综合属性构造出了一个最早的语法制导编译器。L 属性定义的分类来自于[6]。

继承属性、依赖图以及对 SDD 的循环依赖的测试(也就是说，是否存在一棵语法分析树使得不存在计算各个属性值的可行顺序)来自于 Knuth[5]。Jazayeri、Ogden 和 Rounds[3]说明了检测循环所需要的时间和 SDD 的大小呈指数关系。

语法分析器的生成器，比如 Yacc[4](也可见第 4 章中的文献目录)，支持语法分析过程中的属性求值。

Paakki 的研究成果[7]是阅读关于语法制导定义和翻译的大量文献的好起点。

1. Brooker, R. A. and D. Morris, “A general translation program for phrase structure languages,” *J. ACM* **9**:1 (1962), pp. 1–10.

2. Irons, E. T., “A syntax directed compiler for Algol 60,” *Comm. ACM* **4**:1 (1961), pp. 51–55.

3. Jazayeri, M., W. F. Ogden, and W. C. Rounds, “The intrinsic exponential complexity of the circularity problem for attribute grammars,” *Comm. ACM* **18**:12 (1975), pp. 697–706.

4. Johnson, S. C., “Yacc — Yet Another Compiler Compiler,” Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975. Available at `http://dinosaur.compilertools.net/yacc/` .

5. Knuth, D.E., “Semantics of context-free languages,” *Mathematical Systems Theory* **2**:2 (1968), pp. 127–145. See also *Mathematical Systems Theory* **5**:1 (1971), pp. 95–96.

6. Lewis, P. M. II, D. J. Rosenkrantz, and R. E. Stearns, “Attributed translations,” *J. Computer and System Sciences* **9**:3 (1974), pp. 279–307.

7. Paakki, J., “Attribute grammar paradigms — a high-level methodology in language implementation,” *Computing Surveys* **27**:2 (1995) pp. 196–255.

8. Samelson, K. and F. L. Bauer, “Sequential formula translation,” *Comm. ACM* **3**:2 (1960), pp. 76–83.