第6章 语义分析

本章要点

- 属性和属性文法
- 属性计算算法
- 符号表
- 数据类型和类型检查
- TINY语言的语义分析

本章要研究的编译程序阶段是计算编译过程所需的附加信息。这个阶段称作语义分析，因为它包括计算上下文无关文法和标准分析算法以外的信息，因此，它不被看成是语法⊖。信息的计算也与被翻译过程的最终含义或语义密切相关。因为编译器完成的分析是静态 (它在执行之前发生)定义的，这样，语义分析也可称作静态语义分析 (static semantic analysis)。在一个典型的静态类型的语言 (如C语言)中，语义分析包括构造符号表、记录声明中建立的名字的含义、在表达式和语句中进行类型推断和类型检查以及在语言的类型规则作用域内判断它们的正确性。

语义分析可以分为两类。第 1类是程序的分析，要求根据编程语言的规则建立其正确性，并保证其正确执行。对于不同的语言来说，语言定义所要求的这一类分析的总量变化很大。在LISP和Smalltalk这类动态制导的语言中，可能完全没有静态语义分析；而在 Ada这类语言中就有很强的需求，程序必须提交执行。其他的语言介于这两种极端情况之间 (例如Pascal语言，不像Ada和C对静态语义分析的要求那样严格，也不像LISP那样完全没有要求)。

语义分析的第2类是由编译程序执行的分析，用以提高翻译程序执行的效率。这一类分析通常包括对“最优化”或代码改进技术的讨论。第 8章“代码生成”中将研究一些这样的方法，而本章则集中讨论语言定义对正确性要求的一般分析。读者应该注意到，这里研究的技术对两类情况都适用。这两类分析也不是相互排斥的，因为与没有正确性要求的语言相比，如静态类型检查这样的正确性要求能使编译程序产生更加有效的代码。另外，值得注意的是，这里讨论的正确性要求永远不能建立程序的完全正确性，正确性仅仅是部分的。但这样的要求仍然是有用的，可以给编程人员提供一些信息，提高程序的安全性和有效性。

静态语义分析包括执行分析的描述 (description)和使用合适的算法对分析的实现(implementation)。在这里，它和词法及语法分析相类似。例如，在语法分析中，我们使用Backus-Naus范式(BNF)中的上下文无关文法描述语法结构，并用各种自顶向下和自底向上的分析算法实现语法结构。在语义分析中，情形不是那么清晰，其部分原因是没有用标准的方法(如BNF)来说明语言的静态语义；另一个原因是对于各种语言，静态语义分析的种类和总量的变化范围很大。编译程序编写者过去常用的且实现得很好的一种描述语义分析方法是：确定语言实体的属性(attribute)或特性，它们必须进行计算并写成属性等式(attribute equation)或语义规则(semantic rule)，并描述这些属性的计算如何与语言的文法规则相关。这样的一组属性和等式称作属性文法(attribute grammar)。属性文法对遵循语法制导语义(syntax-directed semantic)原

⊖ 这一点在第3章的3.6.3节进行了较为详细的讨论。

理的语言最有用，它表明程序的语义内容与它的语法密切相关。所有的现代语言都有这个特性。然而，编译程序的编写者通常必须根据语言手册手工构造属性文法，因为语言设计者很少为之提供。更糟糕的是，由于坚持语言清晰的语法结构，属性文法的构造会有不必要的复杂性。语义计算表达式的一种更好标准是抽象语法，就像抽象语法树表示的那样。但是抽象语法树的说明通常也由语言设计者留给了编译程序编写者。

语义分析实现的算法也不像语法分析算法那样能清晰地表达。这部分原因也是因为在考虑语义分析说明时，出现了刚刚提及的同样的问题。还有另外一个问题，它是由编译过程中分析的时间选择引起的。如果语义分析可以推迟到所有的语法分析 (以及抽象语法树的构造)完成之后进行，那么实现语义分析的任务就相当容易，其本质上由指定对语法树遍历的一个顺序组成，同时在遍历中每次遇到节点时进行计算。这就意味着编译程序必须是多遍的。另一方面，如果必须要求编译程序在一遍中完成所有的操作 (包括代码生成)，那么语义分析的实现就更加会变成寻找计算语义信息的正确顺序和方法的特别的过程 (假定这样的顺序实际存在)。当然，现代的惯例越来越允许编译程序编写者使用多遍扫描简化语义分析和代码生成的过程。

尽管有点扰乱语义分析的状态，研究属性文法和规范发布仍是特别有用的，因为这能从写出更加清晰、简练、不易出错的语义分析代码中得到补偿，同时代码也更加易懂。

因此，本章从研究属性和属性文法开始。接下来是通过属性文法说明实现计算的技术，包括推断与树的遍历相连的计算顺序。随后的两节集中于语义分析的两个主要方面：符号表和类型检查。最后一节讲述前一章介绍的TINY编程语言的语义分析程序。

与第5章不同，本章没有包含对语义分析程序生成器 (semantic analyzer generator)或构造语义分析程序的通用工具的描述。尽管已经构造了许多这样的工具，但没有一个能得到广泛的使用且能用于Lex或Yacc。在本章最后的“注意与参考”一节中，我们提及了几个这样的工具，并为感兴趣的读者提供了参考文献。

6.1 属性和属性文法

属性(attribute)是编程语言结构的任意特性。属性在其包含的信息和复杂性等方面变化很大，特别是当它们能确定时翻译/执行过程的时间。属性的典型例子有：

- 变量的数据类型。
- 表达式的值。
- 存储器中变量的位置。
- 程序的目标代码。
- 数的有效位数。

可以在复杂的处理(甚至编译程序的构造)之前确定属性。例如，一个数的有效位数可以根据语言的定义确定 (或者至少给出一个最小值)。属性也可以在程序执行期间才确定，如 (非常数)表达式的值，或者动态分配的数据结构的位置。属性的计算及将计算值与正在讨论的语言结构联系的过程称作属性的联编(binding)。联编属性发生时编译/执行过程的时间称作联编时间(binding time)。不同的属性变化，甚至不同语言的相同属性都可能有完全不同的联编时间。在执行之前联编的属性称作静态的(static)，而只在执行期间联编的属性是动态的(dynamic)。对于编译程序编写者而言，当然对那些在翻译时联编的动态属性感兴趣。

考虑先前给出的属性表的示例。我们讨论表中每个属性在编译时的联编时间和重要性。

- 在如C或Pascal这样的静态类型的语言中，变量或表达式的数据类型是一个重要的编译时属性。类型检查器(type checker)是一个语义分析程序，它计算定义数据类型的所有语言

实体的数据类型属性，并验证这些类型符合语言的类型规则。在如 C或Pascal这样的语言中，类型检查是语义分析的一个重要部分。而在LISP这样的语言中，数据类型是动态的，LISP编译程序必须生成代码来计算类型，并在程序执行期间完成类型检查。

- 表达式的值通常是动态的，编译程序要在执行时生成代码来计算这些值。然而事实上，一些表达式可能是常量 (例如3+4*5)，语义分析程序可以选择在编译时求出它们的值 (这个过程称作常量合并(constant folding))。
- 变量的分配可以是静态的也可以是动态的，这依赖于语言和变量自身的特性。例如，在FORTRAN77中所有的变量都是静态分配的，而在LISP中所有的变量是动态分配的。C和Pascal语言混合了静态和动态的两种变量分配。因为编译程序与变量分配联系的计算依赖于运行时环境，有时还依赖于具体的目标机器，所以这些计算会一直推迟到代码生成(第7章将更详细地探讨这一问题)。
- 程序的目标代码无疑是一个静态属性。编译程序的代码生成器专门用于这个属性的计算。
- 数的有效位数在编译期间是一个不被明确探讨的属性。编译程序编写者实现值的表示是不言而喻的，这通常被视为运行时环境的一部分，这将在第 7章中讨论。然而，甚至扫描器也需要知道允许的有效位数并判断是否正确转换了常量。

正如从这些例子中看到的，属性计算变化极大。当它们在编译程序中显式出现时，可能在编译过程的任意时刻发生：即使语义分析阶段与属性计算的联系最紧密，扫描器和语法分析程序也都需要对它们有用的属性信息，在语法分析的同时也需要进行一些语义分析。在本章中，我们集中讨论在代码生成前及语法分析后的典型计算 (语法分析期间的语义分析信息见 6.2.5节)。直接应用于代码生成的属性分析在第8章中讨论。

6.1.1 属性文法

在语法制导语义(syntax-directed semantics)中，属性直接与语言的文法符号相联系 (终结符号或非终结符号)[⊖] 。如果X是一个文法符号，a 是X的一个属性，那么我们把与X关联的a的值记作$X.a$。这个记号让人回忆起Pascal语言的记录字段表示符或 (等价于)C语言中的结构成员操作符。实际上，实现属性计算的一种典型方法是使用记录字段 (或结构成员)将属性值放到语法树的节点中去。下一节将更详细地讨论这一点。

若有一个属性的集合$a_1, \ldots, a_k$，语法制导语义的原理应用于每个文法规则$X_0 \rightarrow X_1 X_2 \ldots Xn$ (这里X_0是一个非终结符号，其他的X_i都是任意符号)，每个文法符号X_i的属性$X_i.a_j$的值与规则中其他符号的属性值有关。如果同一个符号X_i在文法规则中出现不止一次，那么每次必须用合适的下标与在其他地方出现的符号区分开来。每个关系用属性等式 (attribute equation)或语义规则(semantics rule)[⊜] 表示，形式如下

$$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k,\ X_1.a_1, \ldots, X_1.a_k, \ldots, X_n.a_1, \ldots, X_n.a_k)$$

这里的f_{ij}是一个数学函数。属性$a_1, \ldots, a_k$的属性文法(attribute grammar)是对语言的所有文法规则的所有这类等式的集合。

在这个一般性情况中，属性文法显得相当复杂。在实际情况下，函数f_{ij}通常非常简单。属性也很少依赖于大量的其他属性，因此可以将相互依赖的属性分割为较小的独立属性集，并对

⊖ 语法制导语义可以简单地称作语义制导语法 (semantics-directed syntax)，因为在大多数语言中语法是用已经在头脑中建立的(最终)语义设计的。

⊜ 后面我们将看到语义规则比属性等式更加通用一些。这里读者可以先把它们看成是等效的。

每个属性集单独写出一个属性文法。

一般地，将属性文法写成表格形式，每个文法规则用属性等式的集合或者相应规则的语义规则列出，如下所示[⊖]：

文法规则	语义规则
规则1	相关的属性等式
.	.
.	.
.	.
规则*n*	相关的属性等式

接下来继续看几个例子。

例6.1 考虑下面简单的无符号数文法：

$$number \rightarrow number\ digit \mid digit$$
$$digit \rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$$

一个数最重要的属性是它的值，我们将其命名为 *val*。每个数字都有一个值，可以用它表示的实际数直接计算。因此，例如，文法规则 $digit \rightarrow \mathbf{0}$ 表明在这个情况下 *digit* 的值为0。这可以用属性等式 $digit.val = 0$ 表示，我们将这个等式和规则 $digit \rightarrow \mathbf{0}$ 联系在一起。此外，每个数都有一个基于它所包含的数字的值。如果一个数使用了下面的规则推导

$$number \rightarrow digit$$

那么这个数就只包含了一个数字，其值就是这个数字的值。用属性等式表示为

$$number.val = digit.val$$

如果一个数包含的数字多于1个，可以使用下列文法规则推导

$$number \rightarrow number\ digit$$

我们必须表示出这个文法规则左边符号的值和右边符号的值之间的关系。请读者注意，在这个文法规则中对 *number* 的两次出现必须进行区分，因为右边的 *number* 和左边的 *number* 的值不相同。我们使用下标进行区分，将这个文法写成如下形式：

$$number_1 \rightarrow number_2\ digit$$

现在考虑一个数，如**34**。这个数的(最左)推导如下：*number* *number digit* *digit digit* **3** *digit* **34**。考虑在这个推导的第一步文法规则 $number_1 \rightarrow number_2\ digit$ 的使用。非终结符号 $number_2$ 对应于数字**3**，*digit* 对应于数字**4**。每个符号的值分别是3和4。为了获得 $number_1$ 的值(它为34)，必须用 $number_2$ 的值乘以10，再加上 *digit* 的值：34 = 3*10+4。换句话说，是把3左移一个十进制位再加上4。相应的属性等式是

$$number_1.val = number_2.val * 10 + digit.val$$

完整的 *val* 属性文法在表6-1中给出。

使用字符串的语法树可以形象化地表示特殊字符串的属性等式的意义。例如，图 6-1给出了数**345**的语法树。在这个图中相应合适的属性等式的计算显示在每个内部节点的下面。在语

⊖ 这些表中通常用表头“语义规则”代替“属性等式”，以便后面对语义规则进行更通用的解释。

法树中计算时观察属性等式对于计算属性值的算法是重要的，就像在下一节将看到的那样⊖。

在表6-1和图6-1中，通过使用不同的字体，我们强调了数字和值的语法表示或者数字语义内容之间的不同。例如在文法规则 *digit*→**0** 中，数字**0**是一个记号或特性，而 *digit.val* = 0的含义是数字的数值为0。

表6-1 例6.1的属性文法

文法规则	语义规则
$Number_1 \rightarrow number_2\ digit$	$number_1.val = number_2.val * 10 + digit.val$
$Number \rightarrow digit$	$number.val = digit.val$
$digit \rightarrow$ **0**	$digit.val = 0$
$digit \rightarrow$ **1**	$digit.val = 1$
$digit \rightarrow$ **2**	$digit.val = 2$
$digit \rightarrow$ **3**	$digit.val = 3$
$digit \rightarrow$ **4**	$digit.val = 4$
$digit \rightarrow$ **5**	$digit.val = 5$
$digit \rightarrow$ **6**	$digit.val = 6$
$digit \rightarrow$ **7**	$digit.val = 7$
$digit \rightarrow$ **8**	$digit.val = 8$
$digit \rightarrow$ **9**	$digit.val = 9$

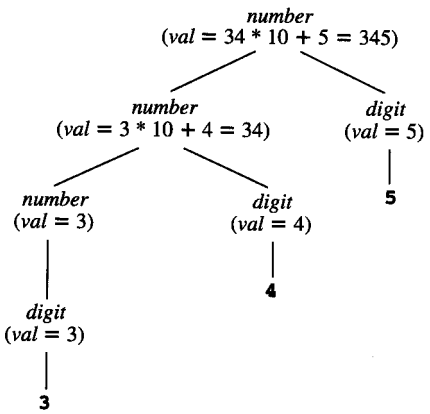


图6-1 显示例6.1中属性计算的语法树

例6.2 考虑下列简单的整数算术表达式文法：

$$exp \rightarrow exp\ \mathbf{+}\ term \mid exp\ \mathbf{-}\ term \mid term$$
$$term \rightarrow term\ \mathbf{*}\ factor \mid factor$$
$$factor \rightarrow (exp) \mid \mathbf{number}$$

这个文法对第5章广泛讨论的简单表达式文法稍微作了一些改动。 *exp*(或*term*或*factor*)的基本属性是它的数字值，写作*val*。*val* 属性的属性等式在表6-2中给出。

⊖ 事实上，扫描器通常认为数是标记，它们的数值也很容易计算。在此期间，扫描器很可能隐含地使用这里定义的属性等式。

表6-2 例6.2的属性文法

文法规则	语义规则
$exp_1 \rightarrow exp_2$ **+** $term$	$exp_1.val = exp_2.val + term.val$
$exp_1 \rightarrow exp_2$ **-** $term$	$exp_1.val = exp_2.val - term.val$
$exp \rightarrow term$	$exp.val = term.val$
$term_1 \rightarrow term_2$ ***** $factor$	$term_1.val = term_2.val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow$ **(**exp**)**	$factor.val = exp.val$
$factor \rightarrow$ ***number***	$factor.val =$ ***number***$.val$

这些等式表示了表达式的语法和它所进行的算术运算的语义之间的关系。注意，在文法规则

$$exp_1 \rightarrow exp_2 + term$$

中语义符号 +(记号)和等式

$$exp_1.val = exp_2.val + term.val$$

中执行的算术加运算符 + 的不同。还要注意***number***.*val*不会在等式的左边。就像在下一节将看到的，这意味着必须在任意一个使用这个属性文法 (例如扫描器)的语义分析之前计算***number***.*val*。换句话说，如果想在属性文法中明确这个值，就必须在属性文法中加进文法规则和属性等式(例如，例6.1中的等式)。

如例6.1一样，也可以通过在语法树的节点上附加等式来表示属性文法包含的计算。例如，给定表达式**(34-3)*42**，可以用在其语法树上值的语义来表达，如图 6-2所示。

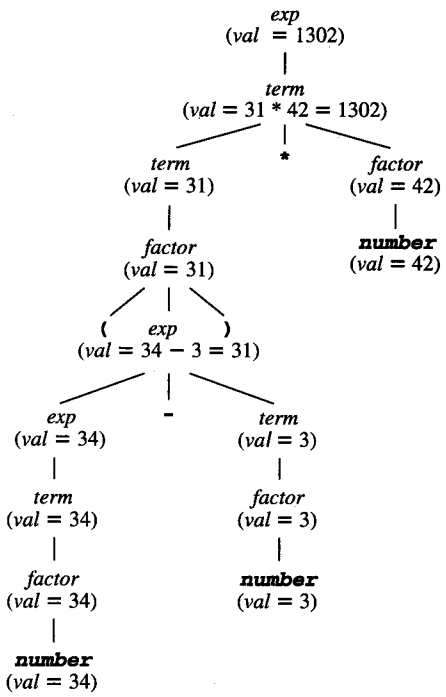


图6-2 **(34-3)*42** 的语法树，显示例6.2中属性文法的val属性计算

例6.3 考虑下列类似C语法中变量声明的简单文法：

decl → *type var-list*
type → **int** | **float**
var-list → ***id***, *var-list* | ***id***

我们要为在声明中标识符给出的变量定义一个数据类型属性，并写出一个等式来表示数据类型属性是如何与声明的类型相关的。通过构造 *dtype*属性的一个属性文法可以实现这一点 (使用名字*dtype*和非终结符*type*的属性进行区分)。*dtype*的属性文法在表6-3中给出。在图中关于属性等式我们做了以下标记。

首先，从{*integer* , *real*}集合中得出*dtype*的值，相应的记号为**int**和**float**。非终结符*type*有一个它表示的记号给定的 *dtype*。通过*decl*文法规则的等式，这个 *dtype*对应于全体*var-list*的*dtype*。通过*var-list*的等式，表中的每个 ***id***都有相同的*dtype*。注意，没有等式包含非终结符*decl* 的*dtype*。实际上*decl*并不需要*dtype*，一个属性的值没有必要为所有的文法符号指定。

同前面一样，可以在一个语法树上显示属性等式。图 6-3给出了一个例子。

表6-3 **例**6.3**的属性文法**

文法规则	语义规则
decl → *type var-list*	*var-list.dtype* = *type.dtype*
type → **int**	*type.dtype* = *integer*
type → **float**	*type.dtype* = *real*
$var\text{-}list_1$ → ***id***, $var\text{-}list_2$	***id***.*dtype* = $var\text{-}list_1$.*dtype*
	$var\text{-}list_2$.*dtype* = $var\text{-}list_1$.*dtype*
var-list → ***id***	***id***.*dtype* = *var-list.dtype*

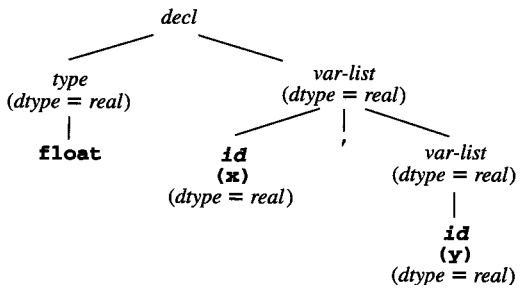


图6-3 字符串**float x,y**的语法树，显示表6-3中属性文法指定的dtype属性

到目前为止，所有的例子都只有一个属性，但属性文法可能会包含几个独立的属性。下一个例子是一个有几个相互联系属性的简单情形。

例6.4 考虑对例6.1中数文法进行修改，数可以是八进制或十进制的。假设这通过一个字符的后缀**o** (八进制)或**d** (十进制)来表示。这样就有下面的文法：

based-num → *num basechar*
basechar → **o** | **d**
num → *num digit* | *digit*
digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

在这种情况中，*num*和*digit*均需要一个新的属性*base*用来计算属性*val*。*base*和*val*的属性文法在表6-4中给出。

表6-4 例6.4的属性文法

文法规则	语义规则
based-num →	*Based-num.val* = *num.val*
num basechar	*num.base* = *basechar.base*
basechar → **o**	*basechar.base* = 8
basechar → **d**	*basechar.base* = 10
$num_1 \rightarrow num_2\ digit$	$num_1.val$ =
	if *digit.val* = *error* **or** $num_2.val$ = *error*
	then *error*
	else $num_2.val * num_1.base + digit.val$
	$num_2.base = num_1.base$
	$digit.base = num_1.base$
num → *digit*	*num.val* = *digit.val*
	digit.base = *num.base*
digit → **0**	*digit.val* = 0
digit → **1**	*digit.val* = 1
...	...
digit → **7**	*digit.val* = 7
digit → **8**	*digit.val* =
	if *digit.base* = 8 **then** *error* **else** 8
digit → **9**	*digit.val* =
	if *digit.base* = 8 **then** *error* **else** 9

在这个属性文法中必须注意两个新的特性。首先，这个 BNF文法在后缀为**o**时自己不能排除错误的(非八进制)数字**8**和**9**的组合。例如，按照上面的 BNF文法，字符串**189o**在语法上是

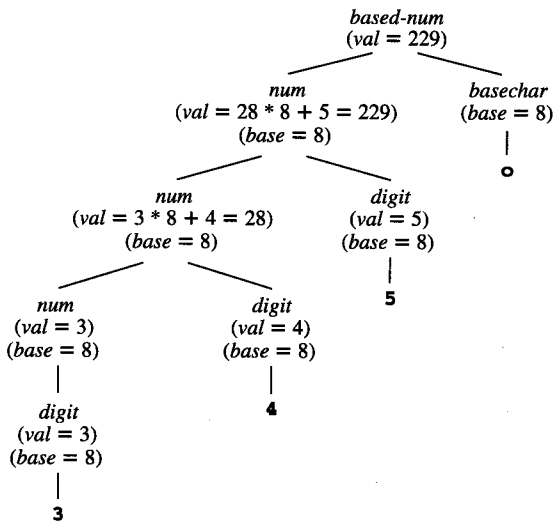


图6-4 例6.4中属性计算的语法树

正确的，但不能赋予任何值。因此，对于这些情况需要一个新的*error*值。另外，这个属性文法必须能表示这样的事实，一个带有后缀**o**的数中包括数字**8**或**9**时将导致值为*error*。最简单的方法是在合适的属性等式函数中使用**if-then-else**表达式。例如，等式

$$num_1.val = \textbf{if } digit.val = error \textbf{ or } num_2.val = error \textbf{ then } error \textbf{ else } num_2.val * num_1.base + digit.val$$

对应于文法规则$num_1 \rightarrow num_2\ digit$，表示如果$num_2.val$或$digit.val$为*error*，那么$num_1.val$也必须为*error*，除此之外只有一种情况：$num_1.val$的值由公式$num_2.val * num_1.base + digit.val$给出。

总结这个例子，再在一个语法树上表示属性计算。图 6-4给出了数**345o**的语法树，同时根据表6-4的属性文法计算出属性值。

6.1.2 属性文法的简化和扩充

if-then-else表达式的使用扩充了表达式的种类，它们可以通过有用的途径出现在属性等式中，在属性等式中，允许出现的表达式的集合称作属性文法的元语言 (metalanguage)。通常我们希望元语言的内涵尽可能清晰，不致于引起其自身语义的混淆。我们还希望元语言接近于一种实际使用的编程语言，因为就像我们即将看到的一样，在语义分析程序中需要把属性等式转换成执行代码。在本书中，我们使用的元语言局限于算术式、逻辑式以及一些其他种类的表达式，再加上**if-then-else**表达式，偶尔还有**case**或**switch**表达式。

定义属性等式的另一个有用的特征是在元语言中加入了函数的使用，函数的定义可以在别处给出。例如，在数的文法中，我们对*digit* 的每个选择都写出了属性等式。可以采用一个更简洁的惯例来代替，为*digit* 写一个文法规则*digit* →**D** (这里**D**是数字中的一个)，相应的属性等式是

$$digit.val = numval\ (\textbf{D})$$

这里*numval*是函数，必须在别处说明其作为属性文法的补充的定义。例如，我们可以用 C代码给出*numval*的定义：

```
int numval ( char D)
{ return (int)D-(int)'0';}
```

在说明属性文法时更简化的有用方法是使用原始文法二义性的但简单的形式。事实上，因为假定已经构造了分析程序，所有二义性在那个阶段都已经处理过了，属性文法可以自由地基于二义性概念，而无须在结果属性中说明任何二义性，例如，例 6.2的算术表达式文法有以下简单的但二义性的形式：

$$exp \rightarrow exp\ \textbf{+}\ exp \mid exp\ \textbf{-}\ exp \mid exp\ \textbf{*}\ exp \mid \textbf{(}\ exp\ \textbf{)} \mid \textbf{\textit{number}}$$

使用这个文法，属性*val*可以通过表6-5中的表定义(与表6-2相比较)。

表6-5 使用二义性文法定义表达式的*val*属性

文法规则	语义规则
$exp_1 \rightarrow exp_2\ \textbf{+}\ exp_3$	$exp_1.val = exp_2.val + exp_3.val$
$exp_1 \rightarrow exp_2\ \textbf{-}\ exp_3$	$exp_1.val = exp_2.val - exp_3.val$
$exp_1 \rightarrow exp_2\ \textbf{*}\ exp_3$	$exp_1.val = exp_2.val * exp_3.val$
$exp_1 \rightarrow \textbf{(}\ exp_2\ \textbf{)}$	$exp_1.val = exp_2.val$
$exp \rightarrow \textbf{\textit{number}}$	$exp.val = \textbf{\textit{number}}.val$

在显示属性值时，也可以通过使用抽象语法树代替分析树进行简化。抽象语法树通常必须用足够的结构来表达属性文法定义的语义。例如表达式 **(34-3)*42**，其分析树和*val*属性在图6-2中已给出，通过图6-5的抽象语法树也能完全表达它的语义。

如下一个例子所示，语法树自身也可以用属性文法指定，这并不奇怪。

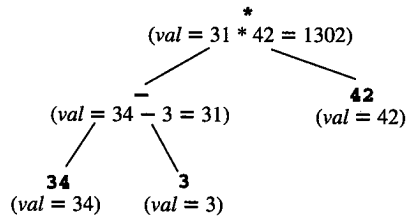


图6-5 **(34-3)*42** 的抽象语法树，显示表6-2或表6-5中属性文法的val属性计算

例6.5 给定例6.2的简单整数表达式文法，通过表6-6给出的属性文法可以定义表达式的抽象语法树。在这个属性文法中，我们使用了两个辅助函数 *mkOpNode*和*mkNumNode*。*mkOpNode*函数有3个参数(一个操作符记号和两个语法树)并构成一个新的树节点，其操作符记号是第1个参数，子节点是第2个和第3个参数。*mkNumNode*函数有一个参数(数字值)并构成一个叶子节点，它表示具有这个值的一个数。在表6-6中我们把数字值写作***number***.*lexval*，表示它是由扫描器构成的。事实上，根据不同的实现，这可以是实际的数字值，或用它的字符串表示 (比较表6-6中的等式和附录B中TINY语法树递归下降构造)。

表6-6 简单整型算术表达式的抽象语法树的属性文法

文法规则	语义规则
$exp_1 \rightarrow exp_2$ **+** *term*	exp_1.*tree* = *mkOpNode* (**+**, exp_2.*tree*, *term*. *tree*)
$exp_1 \rightarrow exp_2$ **-** *term*	exp_1.*tree* = *mkOpNode* (**-**, exp_2.*tree*,*term*. *tree*)
exp → *term*	*exp*.*tree* = *term*.*tree*
$term_1 \rightarrow term_2$ ***** *factor*	$term_1$.*tree* = *mkOpNode* (*****, $term_2$.*tree*,*factor* .*tree*)
term → *factor*	*term*.*tree* = *factor*. *tree*
factor → **(** *exp* **)**	*factor*.*tree* = *exP*. *tree*
factor → ***number***	*factor*.*tree* = *mkNumNode* (***number***.*lexval*)

如何确保一个特定的属性文法是一致的和完整的，是使用属性文法的属性说明的一个主要问题，也就是说，它能唯一定义给定的属性。简单的答案是到目前为止还不能做到。这个问题与确定一个文法是否是二义的类似。实际上，这是用来确定一个文法充分性的分析算法，类似的情形发生在属性文法的情况中。因此，下一节将研究属性计算的算法规则方法，确定一个属性文法是否足够能定义属性值。

6.2 属性计算算法

这一节将以属性文法为基础，研究编译程序中如何计算和使用由属性文法的等式定义的属

性。基本上，这等于将属性等式转化为计算规则。因此，属性等式

$$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k,\ X_1.a_1, \ldots, X_1.a_k,\ \ldots,\ X_n.a_1, \ldots, X_n.a_k)$$

被看作把右边的函数表达式的值赋给属性$X_i.a_j$。为达到这一点，在右边出现的所有属性值必须已经存在。在属性文法的说明中，这个要求可能被忽略，可以将等式写成任意顺序而不会影响正确性。实现对应于属性文法的算法规则的问题，主要在于为赋值和属性分配寻找一个顺序，并确保当每次进行计算时使用的所有属性值都是有效的。属性等式本身指示了属性计算时的顺序约束，首先是使用指示图表示这些约束来明确顺序的约束。这些指示图叫做相关图。

6.2.1 相关图和赋值顺序

给定一个属性文法，每个文法规则选择有一个相关依赖图 (associated dependency graph)。文法规则中的每个符号在这个图中都有用每个属性$X_i.a_j$标记的节点，对每个属性等式

$$X_i.a_j = f_{ij}(\ldots, X_m.a_k, \ldots)$$

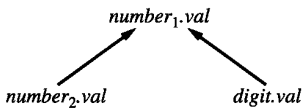
相关于文法规则从在右边的每个节点$X_m.a_k$到节点$X_i.a_j$有一条边(表示$X_i.a_j$对$X_m.a_k$的依赖)。依据上下文无关文法，在语言产生时给定一个合法的字符串，这个字符串的依赖图 (dependency graph)就是字符串语法树中选择表示每个(非叶子)节点文法规则依赖图的联合。

在绘制每个文法规则或字符串的依赖图时，与每个符号 X相关的节点均画在一组中，这样依赖就可以看作是语法树的构造。

例6.6　考虑例6.1的文法，属性文法在表6-1中给出。这里只有一个属性 *val*，因此每个符号在每个依赖图中只有一个节点对应于其 *val* 属性。选择$number_1 \rightarrow number_2.digit$ 的文法规则有单个的相关属性等式

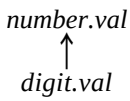
$$number_1.val = number_2.val * 10 + digit.val$$

这个文法规则选择的依赖图是



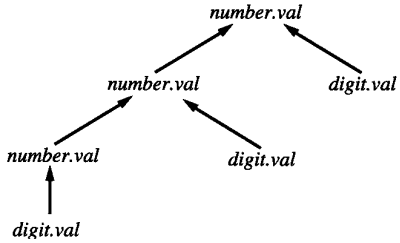
(在以后的依赖图中，因为图形化表示可以清楚地区分不同节点的不同出现，所以对重复的符号将省略下标)。

类似地，属性等式*number.val = digit.val*中文法规则*number* → *digit*的相关图是



对于剩下的形如*digit*→**D**的文法规则，因为*digit.val* 可以从规则的右边直接计算，其相关图是无足轻重的(它们没有边)。

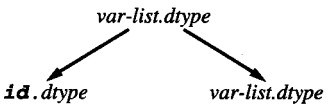
最后，对应于语法树(见图6-1)，字符串**345**的相关图如下：



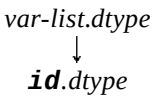
例6.7 考虑例6.3的文法，属性*dtype*的属性文法在表6-3中给出。在这个例子中，文法规则*var-list*$_1$ →**id** , *var-list*$_2$ 有两个相关的属性等式

$$\textbf{id}\ dtype = var\text{-}list_1.dtype$$
$$var\text{-}list_2.dtype = var\text{-}list_1.dtype$$

依赖图是

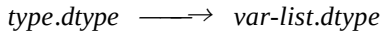


类似地，文法规则*var-list* →**id** 的相关图是

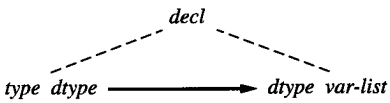


type →**int** 和*type* →**float** 两个规则的相关图无关紧要。

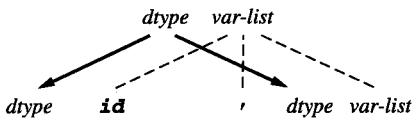
最后，与等式*var-list.dtype* = *type.dtype*相关的*decl* →*type var-list* 规则的相关图是



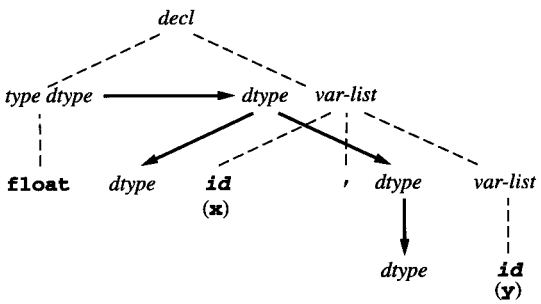
在这种情况中，因为*decl*不直接包含在相关图中，所以这个相关图相关的文法规则并不完全清晰。正因为这一点(稍后将讨论其他一些原因)，我们通常重叠在相应的文法规则的语法树片段上绘制相关图。这样，上面的相关图就可以画作



这就使和相关有关的文法规则更加清晰。还要注意，在画语法树节点时我们禁止使用属性的圆点符号，而是通过写出与其相连的下一个节点来表示属性。这样，在这个例子中第一个相关图也可以画作



最后，字符串**float x,y**的相关图是

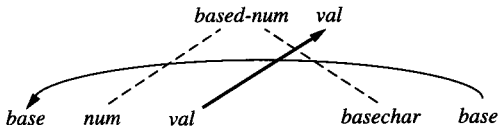


例6.8　考虑例6.4基于数的文法，属性*base*和*val*的属性文法在表6-4中给出。画出下面4条文法规则

based-num → *num basechar*
num → *num digit*
num → *digit*
digit → **9**

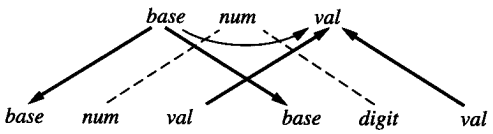
和字符串**345o**的相关图，其语法树在图6-4中给出。

首先从文法规则*based-num* → *num basechar*的相关图开始：



这个图表示了*based-num.val* = *num.val*和*num.base* = *basechar.base*两个相关的等式的相关。

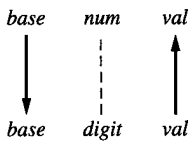
接下来再画出与文法规则*num* → *num digit*相对应的相关图：



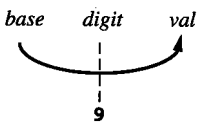
这个图表示3个属性等式的相关

$num_1.val$ =
　　if *digit.val* = *error* **or** $num_2.val$ = *error*
　　then *error*
　　else $num_2.val$ * $num_1.base$ + *digit.val*
$num_2.base = num_1.base$
digit.base = $num_1.base$

文法规则*num* → *digit*的相关图也与此类似：



最后，画出文法规则*digit* → **9**的相关图：



这个图表示由等式*digit.val* = **if** *digit.base* = 8 **then** *error* **else** 9创建的相关，也就是*digit.val*与于*digit.base*(这是**if** 表达式测试的一部分)相关。现在剩下画出字符串**345o**的相关图，如图6-6所示。

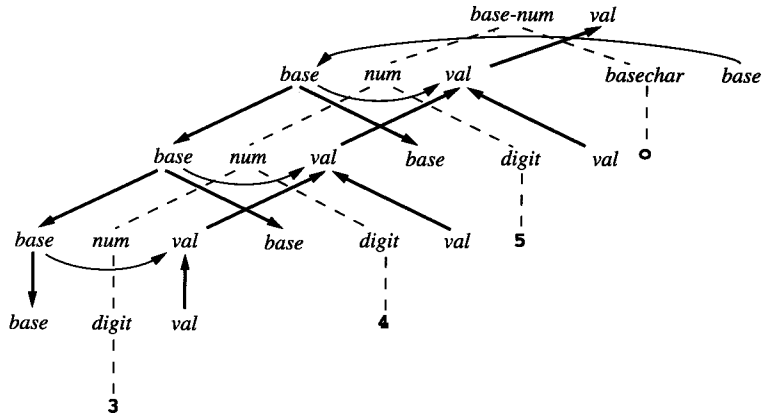


图6-6 字符串345o 的相关图(例6.8)

假设现在要确定一个算法，使用属性等式作为计算规则来计算一个属性文法的属性。给定要转换的一个特定的记号字符串，字符串语法树的相关图根据计算字符串属性的算法给出了一系列顺序约束。实际上，任一个算法在试图计算任何后继节点的属性之前，必须计算相关图中每个节点的属性。遵循这个限制的相关图遍历顺序称作拓扑排序 (topological sort)，而且众所周知，存在拓扑排序的充分必要条件是相关图必须是非循环 (acyclic)的。这样的图形称作确定非循环图(directed acyclic graphs，DAGs)。

例6.9 图6-6的相关图是一个DAG。在图6-7中，给节点编上号(为便于观察删除了下面的语法树)。根据节点的编号给出了一个拓扑排序的顺序。另一个拓扑排序的顺序是

12 6 9 1 2 11 3 8 4 5 7 10 13 14

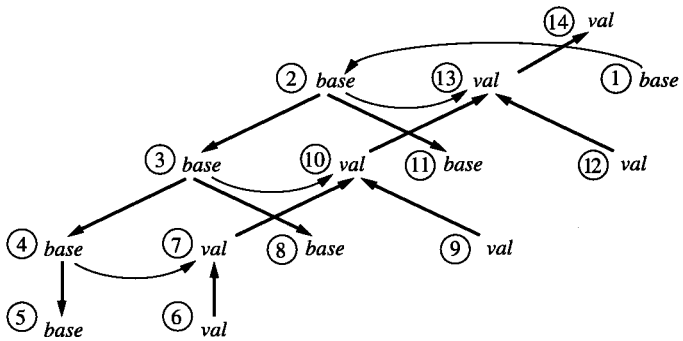


图6-7 字符串345o 的相关图(例6.9)

在图的根节点上如何找到属性值 (图的根节点是没有前驱的节点)是在使用相关图的拓扑排序计算属性值时产生的一个问题。在图 6-7中，节点1、6、9、12都是图的根节点 ⊖ 。这些节点的属性值不依赖于任何其他属性，因此必须使用直接可用的信息计算。这个信息通常在相应的语法树节点的子孙记号表中。例如，在图 6-7中，节点6的*val*依赖于记号**3**，它是对应于*val*的*digit*节点的子节点(见图6-6)。因此，节点6的属性值是**3**。所有这些根节点的值都需要在计算其他任何属性值之前计算。这些计算通常由扫描器或语法分析程序来完成。

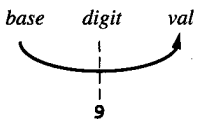
⊖ 相关图的根节点不要和语法树的根节点混淆。

在编译以及随后的为了确定属性赋值的顺序而进行的相关图的拓扑排序过程中，基于一个算法在相关图的构造上进行属性分析是可能的。因为在编译时相关图的构造是基于特定的语法树的，这个方法有时称作分析树方法(parse tree method)。在任何非循环(noncircular)的属性文法中能够对属性赋值，也就是说，对每一个可能的相关图属性文法是非循环的。

这种方法有几个问题。首先，在编译时构造相关图增加了额外的复杂性。其次，这种方法在编译时确定相关图是否非循环时，它通常不恰当地等待发现一个环，直到到达编译时间，因为在原始的属性文法中，环几乎都是表示错误。换句话说，属性文法必须预先进行无环测试。有一个算法进行这个工作(参见“注意与参考”一节)，但这是一个时间指数级增长的算法。当然，这个算法只需要在编译器构造时运行一次，因此这不能成为反对这个算法的压倒性的证据(至少对编译器的构造而言)。这种方法的复杂性就是更强的证据。

上述的属性赋值的另一种方法(几乎每一个编译器都采用)即编译器编写者在编译器构造时分析属性文法，并固定一个属性赋值顺序。虽然这种方法仍然使用语法树作为属性赋值的指导，但因为它依赖于对属性等式或语义规则的分析，所以它仍被称作基于规则的方法 (rule-based method)。在编译器构造时固定属性赋值顺序的这一类属性文法没有所有的非循环属性文法通用，但在实际中，所有合理的属性文法都有这个特性。它们有时称作强非循环 (strongly noncircular)属性文法。在下面的例子后面，我们将对这类属性文法讨论基于规则的算法。

例6.10 再次考虑例6.8的相关图和例6.9中讨论的相关图的拓扑排序(见图6-7)。虽然图6-7中的节点6、9和12是DAG的根节点，并且因此都能够出现在拓扑排序的开始，但在基于规则的方法中这是不可能的。其原因是如果相应的记号是**8**或**9**，任何*val* 可能依赖于与它相关的*digit* 节点的*base*。例如，对*digit* → **9**相关图是



因此，在图6-7中，节点6可能依赖于节点5，节点9可能依赖于节点8，而节点12可能依赖于节点11。在基于规则的方法中，这些节点将在任何可能潜在依赖的节点之后被强制赋值。因此，一种赋值顺序，首先赋值节点12(见例6.9)，这对图6-7中特定的树是正确的，但对基于规则的算法这是错误的顺序，因为它将破坏其他语法树的顺序。

6.2.2 合成和继承属性

基于规则的属性赋值依赖于分析树或语法树明确或不明确的遍历。不同种类的遍历处理的属性相关，在项目和能力的种类上都不同。为了研究这些不同之处，首先必须根据它们具有的相关的种类对属性分类。处理的最简单的依赖是综合属性，定义如下。

定义 一个属性是合成的(synthesized)，如果在语法树中它所有的相关都从子节点指向父节点。等价地，一个属性a是合成的，如果给定一个文法规则$A \rightarrow X_1 X_2 \ldots X_n$，左边仅有一个$a$的相关属性等式有以下形式

$$A.a = f(X_1.a_1, \ldots, X_1.a_k, \ldots, X_n.a_1, \ldots, X_n.a_k)$$

一个属性文法中所有的属性都是合成的，就称作S属性文法(S-attributed grammar)。

我们已经见过了一些合成属性和S属性文法的例子。在例6.1中数的*val* 属性是合成的(参见

例6.6中的相关图），例6.2中简单整数算术表达式的*val*属性也是一样。

给定由分析程序构造的分析树或语法树， S属性文法的属性值可以通过对树进行简单的自底向上或后序遍历来计算。这可以用以下递归的属性赋值程序来表示：

```
procedure PostEval (T: treenode);
begin
  for each child C of T do
    PostEval (C);
  compute all synthesized attributes of T;
end;
```

例6.11　用合成属性*val* 考虑例6.2中简单算术表达式的属性文法。给定下列语法树 (就像图6-5)的结构

```
typedef enum { Plus, Minus, Times } OpKind;
typedef enum { OpKind, ConstKind } ExpKind;
typedef struct streenode
     { ExpKind kind;
      OpKind op;
      struct streenode *lchild, *rchild;
       int val;
     } STreeNode;
typedef STreeNode *SyntaxTree;
```

PostEval程序可以转换成程序清单6-1中的C程序代码，进行从左向右的遍历。

程序清单6-1　**例**6.11**中后序属性赋值的**C**程序代码**

```
void postEval(SyntaxTree t)
{ int temp;
  if (t->kind = OpKind)
  { postEval(t->lchild);
    postEval(t->rchild);
    switch (t->op)
    { case Plus:
        t->val = t->lchild->val + t->rchild->val;
        break;
      case Minus:
        t->val = t->lchild->val - t->rchild->val;
        break;
      case Minus:
        t->val = t->lchild->val * t->rchild->val;
        break;
    } /* end switch */
  } /* end if */
} /* end postEval */
```

当然，不是所有的属性都是合成的。

定义　一个属性如果不是合成的，则称作继承(inherited)属性。

我们已经见过继承属性的例子，包括例 6.3中的*dtype*属性和例6.4中的*base*属性。无论在分析树中(使用这个名字的理由)从祖先到子孙的继承属性还是从同属的继承属性都有依赖。图 6-8 a和 b说明了继承属性依赖的两种基本种类。这两种类型的依赖在例 6.7中的*dtype*属性中都出现过。这两种分类都为继承属性的原因在于计算继承属性的算法，同属继承通常是通过把属性值经过祖先在同属中传递实现的。实际上，如果语法树的边只从祖先指向子孙 (这样子孙不能直接访问祖先或同属)这也是必要的。另一方面，在语法树中如果一些结构经过同属指针实现，那么同属继承可以直接沿着同属链进行，如图 6-8c的描述。

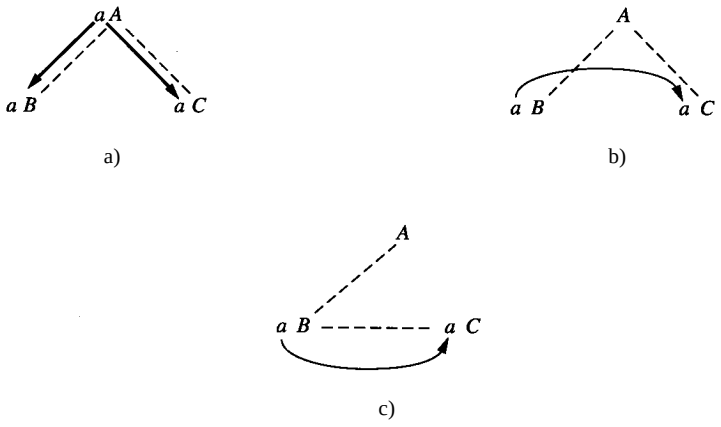


图6-8　不同种类的继承依赖

a) 从祖先到子孙的继承　b) 同属之间的继承　c) 通过同属指针的同属继承

现在回到继承属性赋值的算法方法。继承属性的计算可以通过对分析树或语法树的前序遍历或前序/中序遍历的组合来进行。这可以用下面的伪代码来示意表示：

```
procedure PreEval (T: treenode);
begin
   for each child C of T do
       compute all inherited attributes of C;
       PreEval ( C );
end;
```

与合成属性不同，子孙继承属性计算的顺序是重要的，因为在子孙的属性中继承属性可能有依赖关系。因此在前面的伪代码中 T的每个子孙C的访问顺序必须满足这些依赖的任何要求。在下面两个例子中，我们将使用前面例子中的*dtype*和*base*继承属性来说明这一点。

例6.12　考虑例6.3的文法，它有继承属性*dtype*，其相关图例6.7中给出(见表6-3的属性文法)。首先假设从文法明确构造了分析树，为便于参考重述如下：

decl → *type var-list*
type →**`int`** | **`float`**
var-list →***`id`*****`,`** *var-list* | ***`id`***

所有需要的节点*dtype*属性的递归程序的伪代码如下：

```
procedure EvalType (T: treenode);
```

```
begin
  case nodekind of T of
  decl :
      EvalType (type child of T );
      Assign dtype of type child of T to var-list child of T;
      EvalType (var-list child of T );
  type :
      if child of T = int then T.dtype := integer
      else T.dtype := real;
  var-list :
      assign T.dtype to first child of T;
      if third child of T is not nil then
        assign T.dtype to third child;
        EvalType (third child of T );
  end case:
end EvalType;
```

注意，前序和中序操作是如何根据被处理的不同类型的节点混合的。例如，节点 *decl*在子节点上递归调用*EvalType*之前，要求首先计算其第1个子孙的*dtype*，然后分配到其第2个子孙；这是一个中序处理。另一方面， *var-list*节点在进行任何递归调用之前分配 *dtype*给其子孙；这是一个前序处理。

在图6-9中，我们与*dtype*属性的相关图一起显示了字符串`float x,y`的分析树，并且按照上面的伪代码计算*dtype*的顺序给节点编了号。

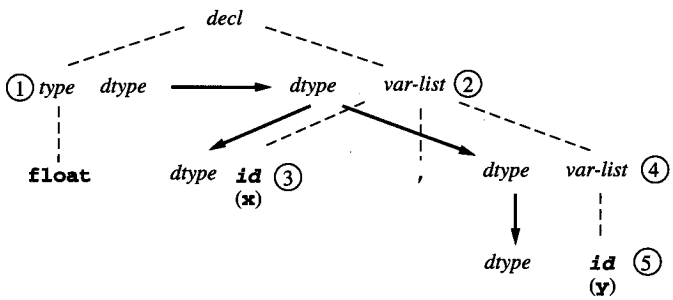
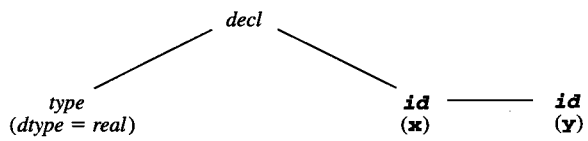


图6-9 显示例6.12遍历顺序的分析树

为了给出这个例子一个完全具体的形式，我们把前面的伪代码转换成实际的 C语言代码。并且，为了替代使用明确的分析树，假设已经构造了一个语法树， *var-list*用`id`节点的同属列表表示。这样，像`float x,y`这样的声明字符串语法树如下(与图6-9比较)



*decl*节点的子孙赋值顺序为从左至右 (首先是节点*type*，然后是节点`x`，最后是节点`y`)。注意，

在这个树中已经包括了*dtype*节点和*type*节点，假定在语法分析时已经预先计算了。

语法树的结构用下面的C语言声明给出：

```
typedef enum { decl, type, id } nodekind;
typedef enum { integer, real } typekind;
typedef struct treeNode
  { nodekind kind;
    struct treeNode
      *lchild, *rchild, *sibling;
    typekind dtype;
    /* for type and id nodes */
    char * name;
    /* for id nodes only */
  } * SyntaxTree;
```

相应地*EvalType*程序的C代码如下：

```
void evalType(SyntaxTree t)
{ switch (t->kind)
  { case decl:
      t->rchild->dtype = t->lchild->dtype;
      evalType(t->rchild);
      break;
    case id:
      if (t->sibling != NULL);
      { t->sibling->dtype = t->dtype;
        evalType(t->sibling);
      }
      break;
  } /* end switch */
} /* end evalType */
```

这段代码可以用下列非递归程序简化，其操作全部在根节点(**decl**)层次上进行：

```
void evalType(SyntaxTree t)
{ if (t->kind == decl)
  { SyntaxTree p = t->rchild;
    p->dtype = t->lchild->dtype;
    while (p->sibling != NULL)
    { p->sibling->dtype = p->dtype;
      p = p->sibling;
    }
  } /* end if */
} /* end evalType */
```

例6.13　考虑例6.4的文法，它具有继承属性*base*(相关图在例6.8中给出)。这里将那个例子的文法重述如下：

based-num → *num basechar*

basechar → **o** | **d**

num → *num digit* | *digit*

9|8|7|6|5|4|3|2|1|0

这个文法有两个新特性。首先，它有两个属性，合成属性*val*以及*val*依赖的继承属性*base*。其次，*base*属性是从*based-num*左边的子孙向右边的子孙继承的(即从*basechar*到*num*)。因此，在这种情况下，我们必须从右向左而不是从左向右给*based-num*的子孙赋值。我们继续给出*EvalWithBase*程序的伪代码用以计算*base*和*val*。对于这种情况，在一遍中*base*用前序遍历计算，而*val*则用后序遍历计算(后面即将讨论多属性和多遍问题)。这段伪代码如下(参见表6-4的属性文法)：

```
procedure EvalWithBase (T: treenode);
begin
  case nodekind of T of
  based-num:
      EvalWithBase ( right child of T );
      assign base of right child of T to base of left child;
      EvalWithBase ( left child of T );
      assign val of left child of T to T.val;
  num:
      assign T.base to base of left child of T;
      EvalWithBase ( left child of T );
      if right child of T is not nil then
        assign T.base to base of right child of T;
        EvalWithBase ( right child of T );
        if vals of left and right children ≠ error then
            T.val := T.base* ( val of left child ) + val of right child;
        else T.val := error;
      else T.val := val of left child;
  basechar:
      if child of T = 0 then T.base := 8
      else T.base := 10;
  digit:
    if T.base := 8 and child of T = 8 or 9 then T.val := error
    else T.val := numval ( child of T );
  end case;
end EvalWithBase;
```

我们把相应的语法树C语言声明的构造和将*EvalWithBase* 伪代码转换为C语言代码的工作留作练习。

在组合合成和继承属性的属性文法中，如果合成属性依赖于继承属性(及其他合成属性)，但继承属性不依赖于任何合成属性，那么就可能在分析树或语法树的一遍遍历中计算所有的属性。上例就是这一点很好的一个例子，赋值顺序可以组合*PostEval* 和*PreEval* 伪代码程序进行概括：

```
procedure CombinedEval ( T:treenode );
begin
  for each child C of T do
    compute all inherited attributes of C;
```

CombinedEval (*C*);
compute all synthesized attributes of T;
end;

继承属性依赖于合成属性的情形更加复杂，需要对分析树或语法树进行多遍遍历，如下一个例子所示。

例6.14　考虑下列表达式文法的简化版本：

$$exp \rightarrow exp \ / \ exp \mid \mathbf{num} \mid \mathbf{num.num}$$

这个文法只有一个操作符——除，用记号 /表示。它还有两种数的形式，整型数由数字序列组成，用记号**num**表示，浮点数用记号**num.num**表示。这个文法的思想是：根据操作数是浮点数还是整型数，操作符可能会有不同的解释。特别对于除法，根据是否允许有小数部分而结果完全不同。如果不允许，除法通常称作**div**操作，**5/4**的值就是5 **div** 4 = 1。如果是浮点数除法，**5/4**的值就是1.2。

现在假设一种编程语言要求混合表达式提供浮点计算能力，根据语义使用相应的操作。因此，表达式**5/2/2.0** (假定除法是左结合)的含义是1.25，而**5/2/2**的含义是1[⊖]。描述这些语义需要3个属性：一个合成的布尔值属性 *isFloat*，指示表达式的任何部分是否有浮点数值；一个继承属性*etype*，它有两个值*int* 和*float*，它们给出每个子表达式的类型以及哪一个表达式依赖*isFloat*；最后是每个子表达式计算的 *val*，它依赖继承属性*etype*。这个情况也要求对顶层表达式进行区分(这样我们知道没有更多的子表达式需要考虑)。我们给文法增加一个开始符号：

$$S \rightarrow exp$$

属性等式在表6-7中给出。在文法规则$exp \rightarrow \mathbf{num}$ 的等式中，我们使用*Float* (***num***.*val*) 表示将整型值***num***.*val* 转换成浮点数值的函数。我们还使用“ / ”表示浮点数除法，使用“ **div** ”表示整数除法。

在这个例子中，属性*isFloat*、*etype*和*val*可以用对分析树或语法树的两遍遍历来计算。第一遍通过后序遍历计算合成属性*isFloat*。第二遍用前序和后序遍历的组合计算继承属性*etype*和合成属性*val*。我们把这些遍的描述、表达式相应的属性等式以及在语法树上进行后续遍的伪代码或C语言代码构造留作练习。

表6-7　**例**6.14**的属性文法**

文法规则	语义规则
S →*exp*	*exp.etype* = **if** *exp.isFloat* **then** *float* **else** *int* S.*val* = *exp.val*
$exp_1 \rightarrow exp_2$ / exp_3	$exp_1.isFloat = exp_2.isFloat$ **or** $exp_3.isFloat$ $exp_2.etype = exp_1.etype$ $exp_3.etype = exp_1.etype$ $exp_1.val =$ **if** $exp_1.etype = int$ **then** $exp_2.val$ **div** $exp_3.val$ **else** $exp_2.val$ / $exp_3.val$

⊖ 这个规则与C语言中使用的规则不同。例如，在C语言中，**5/2/2.0**的值是1.0，而不是1.25。

（续）

文法规则	语义规则
exp → ***num***	*exp.isFloat* = **false** *exp.val* = 　**if** *exp.etype* = *int* **then** ***num***.*val* 　**else** *Float* (***num***.*val*)
exp →***num*.*num***	*exp.isFloat* = **true** *exp.val* = ***num*.*num***.*val*

6.2.3 作为参数和返回值的属性

通常在计算属性时，利用参数和返回的函数值与属性值进行通信，而不是把它们作为字段存储在语法树的记录结构中，这样做是有意义的。当许多属性值都相同，或者仅仅在计算其他属性值时临时使用，就尤为重要。对于这种情况，使用语法树的空间在每个节点存储属性值就没什么意义了。事实上，递归遍历程序用前序计算继承属性，而用后序计算合成属性，在子节点把继承属性作为参数传递给递归函数调用，并接收合成属性作为那些相同调用的返回值。前几章已经出现了几个这样的例子。特别地，算术表达式合成属性值的计算能通过递归分析程序返回当前表达式的值进行。类似地，在语法分析期间，因为直到它的构造完成，也没有用以记录作为属性的自身的数据结构存在，所以语法树自身作为合成属性必须通过返回值计算。

对于更复杂的情况，例如当要返回不止一个合成属性时，就有必要使用记录结构或联合作为返回值，或者针对不同的情况把递归程序分割成几个程序。我们用一个例子来说明这一点。

例6.15　考虑例6.13中的递归程序*EvalWithBase*。在这个程序中，一个数的*base*属性只计算一次，之后就用于随后的*val*属性的计算。类似地，在一个完整的数的值的计算中，数的部分*val*属性只是临时使用。把*base*转换成参数(作为继承属性)和把*val*转换成返回值是有意义的。将*EvalWithBase*程序修改如下：

```
function EvalWithBase (T: treenode; base: integer ): integer;
var temp, temp2: integer;
begin
  case nodekind of T of
  based-num:
      temp:= EvalWithBase ( right child of T );
      retutn EvalWithBase ( left child of T, temp );
  num:
      temp := EvalWithBase ( left child of T, base );
      if right child of T is not nil then
        temp2 := EvalWithBase ( right child of T, base );
        if temp≠error and temp2≠error then
            return base*temp + temp2
        else return error;
      else return temp;
```

```
    basechar:
        if child of T = 'o' then return 8
        else return 10;
    digit:
        if base = 8 and child of T = '8' or '9' then return error
        else return numval (child of T );
    end case;
end EvalWithBase;
```

当然，只有工作在 *base*属性和 *val*属性才有相同的 *integer*数据类型，因为在一种情况下*EvalWithBase*返回*base*属性(当分析树节点是一个*basechar*节点)，在另一种情况下*EvalWithBase*返回*val*属性。在第1次调用*EvalWithBase*时也有些不规则(在分析树根节点*base-num*节点)，即使它可能不存在，随后再忽略掉，此时也必须提供一个 *base*值。例如，启动计算时，必须进行这样的调用

EvalWithBase (*rootnode*， 0)；

哑元的基值为0。因此，区分3种情况：*base_num*、*basechar*和*digit*，并且对这3种情况写出3个独立的程序是合理的。伪代码如下：

```
function EvalBasedNum (T: treenode ) : integer;
(*only called on root node*)
begin
   return EvalNum ( left child of T, EvalBase (right child of T) );
end EvalBasedNum;

function EvalBase ( T: treenode ): integer;
(*only called on basechar node*)
begin
   if child of T = 'o' then return 8
   else return 10;
end EvalBase;

function EvalNum ( T: treenode; base: integer ): integer;
var temp, temp2: integer;
begin
   case nodekind of T of
   num:
        temp := EvalWithBase ( left child of T, base );
        if right child of T is not nil then
           temp2 := EvalWithBase ( right child of T, base );
           if temp≠error and temp2≠error then
                return base*temp + temp2
           else return error;
        else return temp;
```

```
    digit:
        if base = 8 and child of T = 8 or 9 then return error
        else return numval ( child of T );
    end case;
end EvalNum;
```

6.2.4 使用扩展数据结构存储属性值

对那些不能方便地把属性值作为参数或返回值使用的情况 (特别是当属性值有重要的结构时，在翻译时可能专门需要)，把属性值存储在语法树节点也是不合理的。在这些情况中，如查表、图以及其他一些数据结构对于获得属性值的正确活动和可用性都是有用的。通过由到表示用于维护属性值的相应的数据结构调用替换属性等式 (表示属性值的赋值)就可反映出属性文法自身可以被修改的这个需要。这导致语义规则不再表示一个属性文法，但在描述属性的语义时仍然有用，而程序的操作清晰了。

例6.16 考虑前一个例子使用参数和返回值的*EvalWithBase*程序。因为属性一旦设置，在值计算过程中就固定了，我们可以使用一个非局部变量存储它的值，而不是在每次作为一个参数传递(如果*base*不固定，这样一个递归过程是危险的甚至是不正确的)。因此，我们可以改变*EvalWithBase*的伪代码如下：

```
function EvalWithBase (T: treenode ):integer;
var temp , temp2: integer;
begin
   case nodekind of T of
   based-num:
        SetBase ( right child of T );
        retutn EvalWithBase ( left child of T );
   num:
        temp:= EvalWithBase ( left child of T );
        if right child of T is not nil then
          temp2:= EvalWithBase ( right child of T );
          if temp≠error and temp2≠error then
              return base*temp + temp2
          else return error;
        else return temp;
   digit:
        if base  = 8 and child of T = 8 or 9 then return error
        else return numval ( child of T );
    end case;
end EvalWithBase;

procedure SetBase (T:treenode );
begin
   if child of T = 0 then base:= 8
```

else *base* **:** = 10**;**
end *SetBase***;**

这里我们把向非局部变量 *base*赋值的过程分离到*SetBase*程序中，它只在*basechar*节点上调用。*EvalWithBase*剩下的代码则只是直接引用*base*，而不是作为一个参数传递。

也可以改变语义规则来反映非局部变量 *base*的使用。在这种情况下，规则就像下面的一样使用赋值明确地指示非局部变量*base*：

文法规则	语义规则
based-num →	
num basechar	*based-num.val* = *num.val*
basechar → **o**	*base* : = 8
basechar → **d**	*base* := 10
num_1 → num_2 *digit*	num_1*.val* =
	if *digit.val* = *error* **or** num_2*.val* = *error*
	then *error*
	else num_2*.val* *.*base* + *digit.val*
etc.	etc.

在这个意义上，*base*现在不再是一个到目前为止所使用的属性，语义规则也不再构成一个属性文法。然而，如果将*base*看作是具有相关特性的变量，这些规则就仍然为编译器编写者充分定义了*base-num*的语义。

对于语法树，外部数据结构最初的一个例子是符号表 (symbol table)，结合程序中声明的常量、变量和过程存储属性。符号表是一种目录数据结构，有 *insert*、*lookup*、*delete*这样的操作。下一节讨论在一个典型的程序设计语言中符号表的问题。这一节介绍下面这个简单的例子。

例6.17　考虑表6-3中简单声明的属性文法，这个属性文法的属性赋值程序在例 6.12给出。一般地，声明中信息使用声明标识符作为关键字插入到符号表中并存储在哪里，以供之后程序的其他部分翻译使用。因此，假设对这个符号表所在的文法通过 *insert*程序而把标识符名、它声明的数据类型和名数据类型插入到符号表中。声明如下：

procedure *insert* (*name* **:** *string* **;** *dtype* **:** *typekind*)**;**

因此替代在语法树中存储每个变量的数据类型，使用这个程序把它插入到符号表中。而且因为每个声明只有一个相关类型，所以就可在处理过程中使用一个全局变量存储每个声明的常量*dtype*。结果的语义规则如下：

文法规则	语义规则
decl →*type var-list*	
type → **`int`**	*dtype* = *integer*
type → **`float`**	*dtype* = *real*
$var\text{-}list_1$ → ***`id`*****`,`** $var\text{-}list_2$	*insert* (***`id`*****`.`***name*, *dtype*)
var-list → ***`id`***	*insert* (***`id`*****`.`***name*, *dtype*)

在对*insert*的调用中，我们使用***`id`*****`.`***name*查阅标识符字符串，即假定由扫描器或分析程序

要计算的。这些语义规则与相应的属性文法完全不同；事实上，对于文法规则 *decl*就完全没有语义规则。因为对*insert*的调用依赖于在*type*规则中设置的*dtype*，所以虽然很清楚在相关的*vat-list*规则之前必须处理*type*规则，但依赖仍不像表达的那样清晰。

相应的属性赋值程序*EvalType*的伪代码如下(与例6-12的代码相比较)：

```
procedure EvalType ( T: treenode );
begin
  case nodekind of T of
  decl:
      EvalType ( type child of T );
      EvalType ( var-list child of T );
  type:
      if child of T = int then dtype:= integer;
      else dtype:= real;
  var-list:
      insert ( name of first child of T,  dtype )
      if third child of T is not nil then
        EvalType ( third child of T );
  end case;
end EvalType;
```

6.2.5 语法分析时属性的计算

有一个很自然会出现的问题，即在分析阶段的同时要计算哪种扩展属性，而无须等到通过语法树的递归遍历进行对源代码的多遍处理。这对于语法树自身特别重要，如果合成属性要被后面的语义分析使用，它必须在语法分析期间构造。在历史上，因为注意的重点是编译器进行一遍翻译的能力，所以对语法分析阶段计算所有属性的可能性产生了很大的兴趣。现在这一点已不太重要了，因此我们没有对所有已经开发的特定技术提供详尽的分析。然而，这个思想和要求总的来看是有价值的。

在一次分析中哪些属性能成功地计算在很大程度上要取决于使用分析方法的能力和性质。这样一个重要的限制是所有主要的分析方法都从左向右处理输入程序 (这也是前面两章研究LL和LR分析技术的第1个L的内容)。它等价于要求属性能通过从左向右遍历分析树进行赋值。对于合成属性这不是一个限制，因为节点的子节点可以用任意顺序赋值，特别是从左向右。但是对于继承属性，这就意味着在相关图中没有“向后”的依赖 (在分析树中依赖从右指向左)。例如，例6.4的属性文法违反了这个特性，因为*based-num*通过后缀**o**或**d**给出其基于的进制，在字符串结尾看到后缀并进行处理之前， *val*属性都不能进行计算。属性文法确保的这个特性称作L-属性(从左向右)，我们给出以下定义。

定义 属性$a_1, \ldots, a_k$的一个属性文法是L-属性(L-attributed)，如果对每个继承属性a_j和每个文法规则

$$X_0 \to X_1 X_2 \ldots X_n$$

a_j的相关等式都有以下形式

$$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k,\ X_1.a_1, \ldots, X_1.a_k,\ \ldots,\ X_{i-1}.a_1, \ldots, X_{i-1}.a_k)$$

也就是说，在X_i处a_j的值只依赖于在文法规则中X_i左边出现的符号$X_0, \ldots, X_{i-1}$的属性。

作为一个特例，我们已经注意到S- 属性文法是L- 属性文法。

给定一个L- 属性文法，其继承属性不依赖于合成属性，如前所述，通过把继承属性转换成参数以及把合成属性转换成返回值，递归下降的分析程序可以对所有的属性赋值。然而，LR分析程序，如Yacc产生的LALR(1)分析程序，适合于处理主要的合成属性。反过来说，其原因在于LR分析程序的功能强于LL分析程序。当已知在一个派生中使用的文法规则时，属性才变成可计算的，这是因为只有在那时才能确定属性计算的等式。但是， LR分析程序将确定在派生中使用的文法规则推迟到文法规则的右部完全形成时才使用。这使得使用继承属性十分困难，除非它们的特性对于所有可能的右部选择都是固定的。我们将简要地讨论在 Yacc应用中最通常的情况下使用分析栈来计算属性。更复杂的技术将在“注意与参考”一节讲述。

1) **LR分析中合成属性的计算** 对于LR分析程序而言这是一种简单的情况。LR分析程序中通常由一个值栈存储合成属性(如果对每个文法符号有不止一个属性，可能是联合或结构)。值栈将和分析栈并行操作，根据属性等式每次在分析栈出现移进或规约来计算新值。我们用表 6-8来说明这一点，属性文法在表6-5中，这是简单算术表达式的二义性版本。为简单起见，我们对文法使用缩写符号，并且忽略了表中 LR分析算法的一些细节。特别地，没有指明状态号、没有显示增加的开始符号，也没有表达隐含的二义性消除规则。这个表除了通常的分析动作之外，还有两个新栏：值栈和语义动作。语义动作指示当分析栈出现规约时值栈发生的计算 (移进看作是把记号值同时推进分析栈和值栈，因此这和单独的分析程序不同)。

作为语义动作的例子，考虑表6-8的第10步。值栈包含整数值12和5，由记号**+**分割开，5在值栈的栈顶。分析动作通过$E \rightarrow E + E$规约，根据表6-5，相应的语义动作是按照等式$E_1.val = E_2.val + E_3.val$计算。在栈顶分析程序相应的动作如下(伪代码)：

```
pop t3              {从值栈中取出E3.val}
pop                 {丢弃+记号}
pop t2              {从值栈中取出E2.val}
t1 = t2 + t3        {加}
push t1             {将结果压进值栈}
```

在Yacc中第10步的规约表示的情形可以写成如下规则：

```
E :E + E { $$ = $1 + $3}
```

这里伪变量**$i**表示规则右部的内容被规约，并通过从右部向后计数转换成值栈的内容。因此，可以在栈顶找到对应于最右端的**E**的**$3**，而在栈顶下面的两个位置找到**$1**。

表6-8 在LR分析中表达式 3*4+5 的语法和语义动作

	分析栈	输入	语法动作	值栈	语义动作
1	$	**3*4+5** $	移进	$	
2	$ ***n***	***4+5** $	规约$E \rightarrow$ ***n***	$ ***n***	*E.val* = ***n*** *.val*
3	$ *E*	***4+5** $	移进	$ 3	
4	$ *E* *	**4+5** $	移进	$ 3 *	
5	$ *E* * ***n***	**+5** $	规约$E \rightarrow$ ***n***	$ 3 * ***n***	*E.val* = ***n*** *.val*

（续）

	分析栈	输入	语法动作	值栈	语义动作
6	$ *E* * *E*	**+5** $	规约*E* →*E* * *E*	$ 3 * 4	$E_1.val = E_2.val * E_3.val$
7	$ *E*	**+5** $	移进	$ 12	
8	$ *E* +	**5** $	移进	$ 12 +	
9	$ *E* + ***n***	$	规约*E* → ***n***	$ 12 + ***n***	$E.val =$ ***n*** val
10	$ *E* + *E*	$	规约*E* → *E* + *E*	$ 12 + 5	$E_1.val = E_2.val + E_3.val$
11	$ *E*	$		$ 17	

2) **在LR分析中继承前面计算的合成属性** 因为LR分析使用从左向右的赋值策略，因为这些值已经被压进了值栈，所以与规则右边非终止符相关的动作可以把符号的合成属性使用到规则的左边。为了简要说明这一点，可考虑产生式选择 *A*→*B C*，假设*C*有一个继承属性*I* 和以某种方式依赖于*B*：*C.i* = *f* (*B.s*) 的合成属性*s*。通过在*B* 和*C* 之间引入一个ε-产生式安排值栈栈顶的存储，在识别*C* 之前的*C.i* 值可以存进一个变量中：

文法规则	语义规则
A → *B C D*	
B → . . .	{计算*B.s* }
D → ε	*saved_i* = *f* (*valstack*[*top*])
C → . . .	{现在*saved_i* 是可用的}

Yacc中的这个处理十分容易，因为无须明确地引入ε- 产生式。只需在调度的规则处写入存储计算属性的动作就行了：

```
A :B { saved_i = f($1); } C ;
```

(这里伪变量**$1**指的是**B**的值，动作执行时它在栈顶)。Yacc中这样的嵌入动作在第4章5.5.6节中已讲述过。

当总能预测出过去计算的合成属性的位置时，可以使用这个策略的另一方面。对于这种情况，无须将值拷贝到变量中，在值栈中能够直接访问。例如，考虑下列具有继承的 *dtype*属性的L- 属性文法：

文法规则	语义规则
decl → *type var-list*	*var-list.dtype* = *type.dtype*
type → **int**	*type.dtype* = *integer*
type → **float**	*type.dtype* = *real*
$var\text{-}list_1$ → $var\text{-}list_2$ ***,id***	*insert* (***id.*** *name*, $var\text{-}list_1.dtype$)
	$var\text{-}list_2.dtype = var\text{-}list_1.dtype$
var-list → ***id***	*insert* (***id.*** *name.var-list.dtype*)

在这种情况中，在第1个*var-list*识别之前可以将*dtype*属性作为非终结符*type*的合成属性计算进入到值栈中。然后当*var-list*的每条规则规约时，在值栈中通过从栈顶向后计数就可以在固定位置找到*dtype*。当*var-list* →***id***规约时，*dtype*就在栈顶的下面，而当$var\text{-}list_1$ →$var\text{-}list_2$ ***,id*** 规约时，*dtype*在栈顶下面的3个位置。通过在上面的属性文法中为*dtype*消除两个拷贝规则并直接访问值栈，可以实现这个LR分析程序。

文法规则	语义规则
decl → *type var-list*	
type → **int**	*type.dtype* = *integer*
type → **float**	*type.dtype* = *real*
$var\text{-}list_1$ → $var\text{-}list_2$ **,id**	*insert* (**id.***name*, *valstack*[*top* - 3])
var-list → **id**	*insert* (**id.***name*, *valstack*[*top* - 1])

(注意：因为*var-list*没有合成属性*dtype*，分析程序必须在栈中压入一个虚值以保持栈中的正确位置)。

这种方法存在几个问题。首先，它要求程序员在分析过程中直接访问值栈，这在自动产生分析程序时是有风险的。例如，在当前的规则被认可之下 Yacc没有像上面的方法所要求的访问值栈的伪变量转换。因此，要在 Yacc中实现这一方案就必须编写特定的代码。第 2个问题是这个技术仅使用在前面计算的属性的位置能从文法中推断出来的情况下。例如，我们编写了上面的声明文法，*var-list*是右递归的(就像例6.17中的)，然后在栈中有**id** 的一个仲裁数，而*dtype*在栈中的位置是未知的。

到目前为止，在LR分析中处理继承属性最好的技术是使用外部数据结构，如符号表或非局部变量，保存继承属性值，并增加 ε-产生式(或像在Yacc中的嵌入动作)，考虑在适当的时刻这些数据结构的变化。例如，刚讨论的 *dtype*问题的一种解决办法可以从对 Yacc的嵌入动作的讨论中得到(见5.5.6节)。

我们要认识到，即使后一种方法没有排除陷阱，在文法中增加 ε-产生式也可能会增加分析冲突，因此对任意的变量*k*，LALR(1)文法均能转变成非LR(*k*)文法(见练习6.15以及“注意与参考”一节)。在实际情况中，这很少发生。

6.2.6 语法中属性计算的相关性

作为本节最后一个主题，值得注意的是属性严重依赖于文法结构的特性。可能会有这样的情况，不改变语言合法的字符串而修改文法会使属性的计算更简单或更复杂。当然，有以下定理：

定理：(Knuth [1968])给定一个属性文法，通过适当地修改文法，而无须改变文法的语言，所有的继承属性可以改变成合成属性。

我们给出一个例子，说明如何通过修改文法继承属性转换成合成属性。

例6.18 考虑前一个例子中简单声明的文法：

decl → *type var-list*
type → **int** | **float**
var-list → **id,** *var-list* | **id**

表6-3的属性文法的*dtype*属性是继承属性。然而，如果重写这个文法如下：

decl → *var-list* **id**
var-list → *var-list* **id,** | *type*
type → **int** | **float**

那么产生了相同的字符串，但根据下面的属性文法，属性*dtype*现在变成了合成属性：

文法规则	语义规则
decl → *var-list* **id**	**id** .*dtype* = *var-list*.*dtype*
$var\text{-}list_1$ → *var -*$list_2$ **id ,**	**id** .*dtype* = $var\text{-}list_2$.*dtype* $var\text{-}list_1$.*dtype* = $var\text{-}list_2$.*dtype*
var-list → *type*	*var-list*.*dtype* = *type*.*dtype*
type → **int**	*type*. *dtype* = *integer*
type → **float**	*type*.*dtype* = *real*

我们在图6-10中已说明了文法的这个改变对语法树和*dtype*属性计算的影响怎样，它显示了字符串**float x,y**语法树及其属性值和依赖。在图中父节点或兄弟节点值的两个**id**.*dtype*值的依赖用虚线画出。而这些依赖的出现违反了在这个属性文法中没有继承属性的要求，事实上这些依赖总是留在语法树中(即是非递归的)，并且可能由相应的父节点的操作实现。因此，这些操作不看成是继承的。

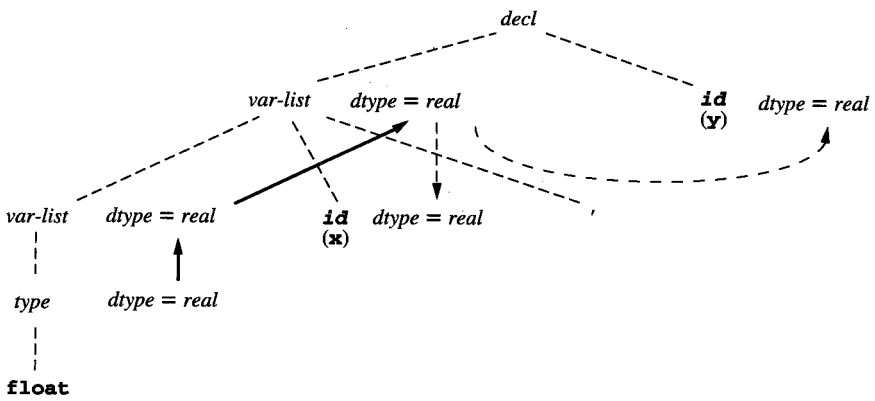


图6-10 字符串**float x,y**的语法树，显示例**6.18**中属性文法指定的**dtype** 属性

事实上，状态理论并不像它看起来那样有用。为了把继承属性转换成合成属性而对文法进行改变通常会使文法和语义规则更加复杂和难以理解。因此，我们不推荐使用这种方法来处理计算继承属性的问题。另一方面，如果一个属性计算看起来非常困难，可能是因为这个文法用了不适合于计算它的方法来定义，也就值得对这个文法进行修改。

6.3 符号表

符号表是编译器中的主要继承属性，并且在语法树之后，形成了主要的数据结构。虽然因为一些例外我们推迟了对符号表的讨论，但它仍对语义分析阶段概念上的框架最适合，读者也会意识到在实际的编译器中，符号表通常紧密地和语法分析程序甚至扫描器相关，它们可能需要直接向符号表中输入信息，或者通过它消除二义性 (C语言中一个这样的例子可参见练习6.22)。然而，在一种设计非常仔细的语言(如Ada或Pascal)中，有可能甚至有理由将符号表的操作推迟到一个完整的阶段之后，这时在语法上已知要翻译的程序是正确的。例如，在 TINY编译器中就这样做了，其符号表将在本章的后面讨论。

符号表主要的操作有插入、查找和删除；其他的一些操作也是必要的。当处理新定义的名

字时，插入操作依赖存储这些名字定义所提供的信息。当相关的代码使用名字时，查找操作找出与名字对应的信息。而当不再运用名字的定义时，需要用删除操作除去名字定义提供的信息。⊖ 这些操作的特性由被翻译的编程语言的规则指定。特别地，在符号表中需要存储什么信息是结构的功能和名字定义的目的。一般包括数据类型信息、应用区域信息 (作用域，下面将讨论)以及在存储器中的最终定位信息。

这一节我们将首先讨论符号表数据结构的组织，以便快速而方便地进行访问。随后将描述一些典型语言的要求和它们在符号表操作上的影响。最后，给出了一个使用属性文法符号表的例子。

6.3.1 符号表的结构

在编译器中符号表是一个典型的目录数据结构。插入、查找和删除这 3种基本操作的效率根据数据结构的组织的不同而变化很大。对不同组织结构效率的分析以及对好的组织策略的研究是数据结构课程的一个主要主题。因此，本书对这个题目不详细讨论，但我们提请读者参考本章最后的“注意与参考”一节提及的资料，获取更多的信息。这里，我们将给出编译器构造中这些表格最有用的数据结构的概要。

目录结构的典型实现包括线性表、各种搜索树结构 (二叉搜索树、AVL树、B树)以及杂凑表(hash表)等。线性表是一种较好的基本数据结构，它能够提供3种基本操作方便而直接的实现，即用恒定次数的插入操作 (通常插入在前面或后面)以及查找和删除操作，表的大小是线性的。这对编译器的实现是很好的，不必去关心编译的速度，就像一个原型或实验编译器，或者用于非常小的程序的解释器。搜索树结构对符号表的用处稍微小一点，部分是因为它们没有提供最好的效率，也因为删除操作的复杂性。杂凑表通常为符号表的实现提供了最好的选择，因为所有3种操作都能在几乎恒定的时间内完成，在实践中也最常使用。因此，对杂凑表的讨论更加详细一些。

杂凑表是一个入口数组，称作“桶 (bucket) ”，使用一个整数范围的索引，通常从 0到表的尺寸减1。杂凑函数(hash fuction)把索引键(在这种情况下是标识符名，组成一个字符串)转换成索引范围内的一个整数的杂凑值，对应于索引键的项存储在这个索引的“桶”中。必须非常小心，杂凑函数在索引范围内尽可能一致地分配键索引，因为杂凑冲突 (collision)(两个键由杂凑函数映射到相同的索引)在查找和删除操作时将引起性能的下降。杂凑函数也需要在一个恒定的时间内操作，或者至少根据键的尺寸时间是线性的 (如果键的尺寸有界，这可以计算恒定时间)。我们不久将研究杂凑函数。

一个重要的问题是杂凑表如何处理冲突 (这称为冲突解决(collision resolution))。一种方法是在每个“桶”中对一个项分配刚好够的空间，通过在连续的“桶”中插入新项来解决冲突 (这有时称作开放寻址(open addressing))。在这种情况中，杂凑表的内容由表所使用的数组的大小限制，当数组填写冲突越来越频繁时，就会引起性能的显著下降。这个方法进一步的问题是，至少对编译器的结构而言实现删除操作比较困难，并且删除不会改进后继表的性能。

编译器结构最好的方案对应于开放寻址也许是另一种方法，称作分离链表 (separate chaining)。在这种方法中每个“桶”实际上是一个线性表，通过把新的项插入到“桶”表中来解决冲突。图6-11给出了这种方案的一个简单的例子，其杂凑表的尺寸是 5 (小得很不现实，仅作演示用)。在那个表中，假定插入了4个标识符(**i**、**j**、**size**和**temp**)，并且**size**和**j**有相同

⊖ 与销毁这个信息相比，删除操作更象是从可视区移走，可能是存储到别处，或者把它标记成不活动的。

的杂凑值(命名为1)。在图中我们看到，“桶”号为1的表中**size**在**j**的前面；表中项的顺序依赖于插入的顺序以及表维护的方式。一种通常的方法是总是插入在表的开始，这样使用这种方法**size**在**j**之后插入。

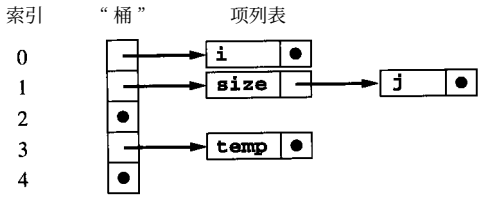


图6-11 分离链接的杂凑表，显示如何解决冲突

图6-11也显示了在每个“桶”中作为链接列表实现的列表(实心圆点用来表示空指针)。这些可以使用编译器实现语言的动态指针分配方法进行分配，或者在编译器自身的空间数组中手工分配。

编译器编写者必须回答的一个问题是要初始化多大的“桶”数组。通常，在编译器构成时，这个大小就固定了。⊖一般的大小范围从几百到上千。如果动态分配实际的入口，即使很小的数组也允许编译很大的程序，只是花费一些额外的编译时间。在任何情况下，“桶”数组的实际大小要选择一个素数，因为这将使一般的杂凑函数运行得更好。例如，如果希望“桶”数组的大小是200，就应该选择211作为数组的大小而不是200(211是大于200的最小的素数)。

现在我们转向描述通用的杂凑函数。在符号表实现中使用的杂凑函数将字符串(标识符名)转换成0. . . *size*-1范围内的一个整数。一般这通过3步来进行。首先，字符串中的每个字符转换成一个非负整数。然后，这些整数用一定的方法组合形成一个整数。最后，把结果整数调整到0. . . *size*-1范围内。

通常使用编译器实现语言内嵌的转换机构把每个字符转换成非负整数。例如，Pascal的**ord**函数将字符串转换成整数，通常是其ASCII值。类似地，在C语言中，如果要在算术表达式中使用字符或把它赋给一整型变量，就自动地将其转换成整数。

使用数学上的取模(modulo)函数很容易调整一个非负整数落到0. . . *size*-1范围内，它返回用*size*去除一个数所得的余数。这个函数在Pascal中称作**mod**，在C中用%表示。使用这种方法时，*size*是一个素数，这一点很重要。否则，随机分配的整数集不会将标定的值随机分配到0. . . *size*-1范围内。

留给杂凑表实现者选择一个方法，把字符的不同整数值组合成一个非负整数。一种简单的方法是忽略许多字符，只把开头的几个字符，或第一个、中间的和最后一个字符的值加在一起。这对编译器来说是不适当的，因为编程者倾向于成组分配变量名，像**temp1**、**temp2**，或**m1tmp**、**m2tmp**等等，并且这种方法在这样的名字中会经常引起冲突。因此，选择的方法要包括每个名字中所有的字符。另一种流行但不适当的方法是简单地把所有字符的值加起来。使用那种方法，所有排列相同的字符，像**tempx**和**xtemp**，会引起冲突。

这些问题的一个好的解决办法是，当加上下一个字符的值时，重复地使用一个常量作为乘法因子。因此，如果c_i是第i个字符的数字值，h_i是在第i步计算的部分杂凑值，那么h_i根据下面的递归公式计算，$h_0 = 0$，$h_{i-1} = \alpha h_i - c_i$，最后的杂凑值用$h = h_n \bmod size$计算。这里$n$是杂凑的名字中字符的个数。这等价于下列公式

$$h = (\alpha^{n-1}c_1 + \alpha^{n-2}c_2 + \cdots + \alpha c_{n-1} + c_n) \bmod size = \left(\sum_{i=1}^{n} \alpha^{n-i}c_i\right) \bmod size$$

当然，在这个公式中α的选择对输出结果有重要影响。α的一种合理的选择是2的幂，如16

⊖ 如果杂凑表增长得太大，在不工作时可有方法增大数组的尺寸(和改变杂凑函数)，但这很复杂也很少使用。

或128，这样乘法可以通过移位来完成。实际上，选择 α=128的结果是把字符串看成是基于128的数，假定字符值 c_i 都小于128 (对ASCII字符为真)。在本文中提到的其他可能性是各种素数(见“注意与参考”一节)。

在公式中 h 的溢出有时也成问题，特别是在双字节整数的机器中对于较大的 α值。如果整数值用于计算，溢出将导致负数(在双字节补码表示中)，并引起执行错误。在这种情况中，通过在求和循环中执行**mod**操作可以得到相同的结果。在程序清单6-2中给出了杂凑函数 h 的C代码的例子，使用前面的公式，α的值为16(进行4次移位，因为16 = 2^4)。

程序清单6-2　符号表的杂凑函数

```
#define SIZE ...
#define SHIFT 4

int hash ( char * key )
{ int temp = 0;
  int i = 0;
  while (key[i] != '\0')
  { temp = ((temp << SHIFT) + key[i]) % SIZE;
    ++i;
  }
  return temp;
}
```

6.3.2 说明

符号表的行为严重依赖于要翻译的语言的特性。例如，当需要调用插入和删除操作时，它们如何作用于符号表，以及什么属性插入到表中，不同的语言都有很大的变化。甚至当符号表建立时翻译/执行过程的时间和符号表需要存在多长时间，对不同的语言也完全不同。这一节我们简要说明几种语言中影响符号表行为和实现的有关声明的问题。

在编程语言中经常出现的4种基本说明是：常量说明、类型说明、变量说明和过程函数说明。

常量声明(constant declaration)包括C语言中的**const**声明，如

```
const int SIZE = 199;
```

(C语言还有一个**#define**机构来创建常量，但那是在预处理时进行的，而不是严格意义上的编译器处理阶段)。

类型声明(type declaration)包括Pascal语言中的类型声明，如

```
type Table = array [1..SIZE] of Entry;
```

以及C语言中的**struct**和**union**说明，如

```
struct Entry
{ char * name;
  int count;
  struct Entry * next;
};
```

这里用名字**Entry**说明了一个结构类型。C语言中还有一个**typedef**机构用来说明类型的别名

```
typedef struct Entry * EntryPtr;
```

变量说明(variable declarations)是说明中最常用的形式，包括FORTRAN语言中的说明，如

```
integer a,b(100)
```

以及C语言中的说明，如

```
int a,b[100];
```

最后，是过程/函数说明(procedure/function declarations)，如程序清单6-2中说明的C函数。这实际上不比过程/函数类型的常数说明多什么东西，但因为它们特别的特性，通常从语言说明中分离出来。这些说明是明确的(explicit)，使用了特定的语言结构进行说明。也可能有隐含的(implicit)说明，说明依附于执行的指令而不明确说明。例如， FORTRAN和BASIC允许使用没有明确说明的变量。在这些隐含说明中，使用了一些约定提供由明确说明给出的信息。例如，FORTRAN语言有类型的约定，如果没有使用明确的说明，字母 I到N开头的变量自动说明为整型，其他的是实型。隐含说明也可称作使用时说明(declaration by use)，因为没有明确说明的变量在第一次使用时可看成隐含包含了其说明。

通常最容易的是用一张符号表保存所有不同种类的说明的名字，特别是当语言禁止在不同种类的说明中使用相同的名字。有时候，对每种说明使用不同的符号表也较容易，例如，所有的类型说明包含在一张符号表中，而所有的变量说明包含在另一张符号表中。对于某些语言，特别是Algol派生的语言，如C、Pascal和Ada，希望程序不同的区域(如过程)都有独立的符号表，并按照语言的语义规则链接到一起(马上我们将更详细地讨论这一点)。

根据说明的种类，限定名字的属性也不同。常量说明给名字赋一个值，因此有时把常量说明称作值约束(value binding)。被约束的值决定编译器如何处理它们。例如， Pascal和Modula-2要求常量说明的值是静态的，因此由编译器计算。在编译期间编译器可以使用符号表用值来替代常量名。其他语言，如 C和Ada，允许常量是动态的，即在执行期间才计算。这样的常量处理起来更像变量，在执行期间必须产生代码来计算它们的值。然而，这样的常量是单一指派(single assignment)，一旦确定了其值就不再改变。常量说明可以明确或隐含地把数据类型约束到名字。例如，在Pascal语言中，常量的数据类型根据其(静态)值隐含地确定，而在C语言中数据类型显式地给出，就像变量说明一样。

类型说明可以把名字约束为新构造的类型，也可以为存在的已命名的类型创建一个别名。类型名通常用来和类型等价算法协作，按照语言的规则完成程序的类型检查。本章后面的一节将专门讨论类型检查，这里不再进一步讨论类型说明。

变量说明最常用于给名字限定数据类型，像在C语言中

```
Table symtab;
```

通过用名字**Table**表示的数据类型约束名字为**symtab**的变量。变量说明也可以隐含地约束其他属性。其中对符号表有主要影响的一个属性是说明的作用域(scope)或说明起作用的程序的区域(也就是变量说明可到达的区域)。作用域通常由变量在程序中说明的位置包含，但也可能被隐含的动态符号影响并和其他说明相互作用。作用域也可能是常量、类型和过程说明的特性。不久我们将更详细地讨论作用域规则。

涉及作用域的变量通过说明明确或隐含地约束，其属性是为说明的变量分配内存，以及执行分配的时机(有时称作生存期(lifetime)或说明的宽度(extent))。例如，在C语言中，所有在函数外部说明的变量都静态分配(即在执行开始前进行)，因此宽度等于整个程序的执行时间，而在函数内说明的变量只在每个函数调用期间才分配 (因此称作自动(automatic)分配)。通过在说明中使用关键字**static**，C语言也允许函数内说明的宽度从自动变为静态，例如

```
int count(void)
{ static int counter = 0;
```

```
  return ++counter;
}
```

函数count有一个静态的局部变量counter，每次调用都保留其值，因此count返回当前它被调用的次数。

C语言也区分说明是用来控制内存分配还是用于类型检查。在 C语言中，任何用关键字extern开头的说明不用来执行分配。因此，如果前一个函数写成

```
int count(void)
{ extern int counter;
  return ++counter;
}
```

变量counter可以在程序的任何地方分配和初始化)。C语言把分配内存的称作说明(definitions)，而保留单词“ **declaration** (说明) ”用于不需要分配内存的说明。因此，用关键字 extern开头的说明不是一个定义，但一个标准的变量说明，如

```
int x;
```

是一个定义。在C语言中，相同的变量可能有许多说明，但只有一个定义。

存储器分配策略像类型检查一样在编译器的设计中形成了一个复杂而重要的部分，它是运行时环境(runtime environment)结构的一部分。下一章全部研究的都是环境，因此这里就不再进一步研究分配了，而转向在符号表中作用域和维护作用域策略的分析。

6.3.3 作用域规则和块结构

编程语言中的作用域规则变化很广，但对许多语言都有几条公共的规则。在本节中，我们讨论其中两条，使用前说明和块结构的最近嵌套规则。

使用前说明(declaration before use)是一条公共规则，在C和Pascal中使用，要求程序文本中的名字要在对它的任何引用之前说明。使用前说明允许符号表在分析期间建立，当在代码中遇到对名字的引用时进行查找；如果查找失败，在使用之前就出现说明错误，编译器给出相应的出错消息。因此，使用前说明有助于实现一遍编译。有些语言不需要使用前说明 (Modula-2是一个例子)，在这样的语言中需要单独的一遍来构成符号表：一遍编译是不可能的。

块结构(block structure)是现代语言的一个公共特性。编程语言中的一块 (block)是能包含说明的任意构造。例如，在Pascal中，块是主程序和过程/函数说明。在C中，块是编译单元(也就是代码文件)、过程/函数说明以及复合语句(用花括号括起来的语句序列{. . .})。在C中结构和联合(Pascal中的记录)也可看成块，因为它们包含字段说明。类似地，面向对象编程语言中的类说明是块。一种语言是块结构 (block structured)的，如果它允许在其他块的内部嵌入块，并且如果一个块中说明的作用域限制在本块以及包含在本块的其他块中，服从最近嵌套规则 (most closely nested rule)：为同一个名字给定几个不同的说明，被引用的说明是最接近引用的那个嵌套块。

为了说明块结构和最近嵌套规则如何影响符号表，考虑程序清单 6-3的C代码片段。在整段代码中，有5个块。首先是整个代码的块，它包含整型变量i和j以及函数f的说明。其次，是f自身的说明，它包含参数 size的说明。再次，是 f函数体的复合语句，它包含字符变量 i和temp的说明(函数说明和相关的函数体也可看成表示一个块)。第四，是包含说明double 的复合语句。最后，是包含说明 char*j的复合语句。在函数 f内部，符号表中有变量 size和temp的单独说明，所有这些名字的使用都参考这些说明。对于名字 i的情况，在f的复合语句

内部**i**有一个的**char**局部说明，根据最近嵌套规则，这个说明代替了围绕代码文件块的 **i**的非局部**int**说明。(非局部的**int i**被称作在**f**内部有一个作用域空洞(scope hole)。)类似地，**f**中两个后来的复合语句中的**j**的说明取代它们各自块内的非局部的 **int**说明。在每种情况中，当局部说明的块存在时，**i**和**j**的原始说明被覆盖。

程序清单6-3 说明嵌套作用域的C代码片段

```
int i,j;

int f (int size)
{ char i, temp;
  ...
  { double j;
    ...
  }
    ...
  { char * j;
    ...
  }
}
```

在许多语言中，像Pascal和Ada (但没有C)，过程和函数都可以嵌套。这对这些语言的运行环境造成了一个复杂的因素 (在下一章研究)，但对嵌套作用域没有造成特别的复杂性。例如，程序清单6-4的Pascal代码包含了嵌套过程**g**和**h**，但和程序清单6-3中的C代码有本质上相同的符号表结构(当然，除了增加的名字**g**和**h**)。

程序清单6-4 说明嵌套作用域的Pascal代码片段

```
program Ex;
var  i,j: integer;

function  f ( size: integer ) : integer;
var  i, temp: char;

  procedure  g;
  var  j: real;
  begin
    ...
  end;

  procedure  h;
  var  j: ^char;
  begin
    ...
  end;

begin (* f *)
  ...
end;

begin  ( * main program * )
  ...
end.
```

为了实现嵌套作用域和最近嵌套规则，符号表插入操作不必改写前面的说明，但必须临时隐藏它们，这样查找操作只能找到名字最近插入的说明。类似地，删除操作不应删除与这个名字相应的所有说明，只需删除最近的一个，而显示前面任何的说明。然后符号表构造可以继续进行：执行插入操作使所有说明的名字进入每个块，执行相应的删除操作使相同的名字从块中退出。换句话说，符号表在处理嵌套作用域期间的行为类似于堆栈的方式。

为了说明这个结构如何能用实际的方法进行维护，考虑早先描述的符号表的杂凑表实现。为简单起见，假定与图6-11类似，进行程序清单6-3中过程**f**的主体说明之后，符号表如图6-12a所示。在处理**f**的主体的第2个复合语句期间(包含说明**char*j**)，符号表如图6-12b所示。最后，函数**f**的块退出之后，符号表如图6-12c所示。注意，对每个名字，每个“桶”中链接表的行为就像名字不同说明的堆栈。

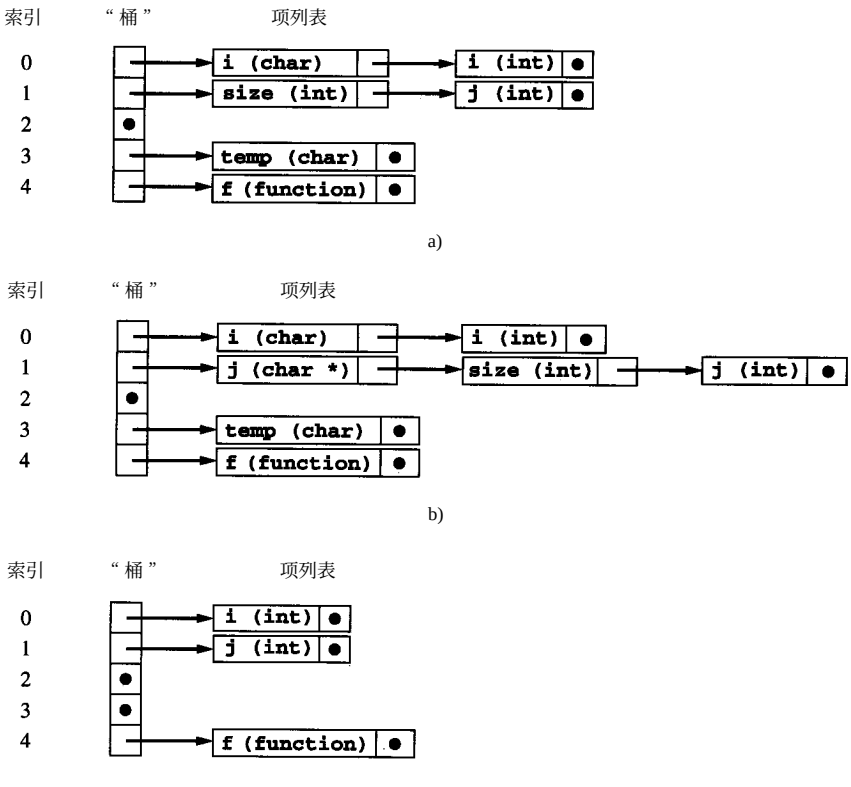


图6-12 符号表内容程序清单6-3

a) 处理f的主体说明之后 b) 处理**f**的主体内第2层嵌套的复合语句的说明之后
c) 退出**f**的主体之后(删除其说明)

有一系列可能的方法可以实现嵌套的作用域。一种解决办法是为每个作用域建立一个新的符号表，再从内到外把它们链接在一起，这样如果查找操作在当前表中没有找到名字，就自动用附上的表继续搜索。离开作用域简单多了，不需要使用删除操作对说明再处理。相反，对应于作用域的整个符号表能在一步中释放。这个结构对应于图 6-12b的一个例子在图6-13给出。在那个图中有3张表，每个作用域一个，从最内层向最外层链接。离开一个作用域只要求重设访问指针(在左边指示)指向最近的外层的作用域。

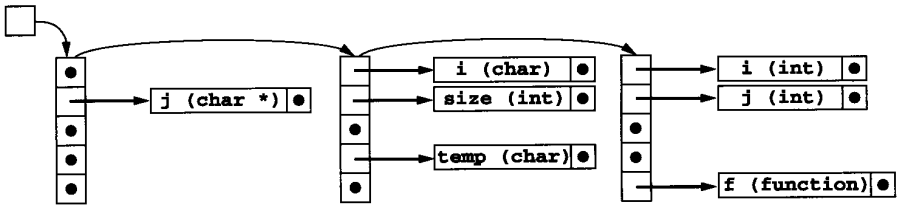


图6-13 对应于图6-12b的符号表结构，每个作用域使用独立的表

在符号表的构造期间还需要一些另外的处理和属性计算，这依赖于具体的语言和编译器操作的细节。一个例子是在Ada中要求非终结符名字在作用域空洞中仍然是可视的，通过使用类似于记录字段选择的符号可以引用它，使用与说明非终结符名字的作用域相关的名字。例如，在程序清单6-4的Pascal代码中，函数**f**内部的全局整型变量**i**，在Ada中作为变量**Ex.i**仍然是可视的(使用程序名标识全局作用域)。因此，会有这样的感觉，在构造符号表时，用名字标识每个作用域，通过累加嵌套作用域名为作用域中说明的每个名字加上前缀。因此，在程序清单6-4中名字**j**的所有出现都区分为**Ex.j**、**Ex.f.g.j**和**Ex.f.h.j**。另外或另一方面，每个作用域需要分配一个嵌套层(nesting level)或嵌套深度(nesting depth)，在每个符号表入口中记录每个名字的嵌套层。因此，在程序清单6-4中，程序的全局变量嵌套层次是0，**f**的说明(其参数和局部变量)的层次是1，**g**和**h**的说明的层次是2。

与刚描述的Ada中作用域选择特性类似的一种机制是C++中作用域限定操作符(scope resolution operator)::。这个操作符允许类说明的作用域从说明的外部进行访问。这可以用来在类说明的外面完成成员函数的说明：

```
class A
{ ... int f();...}; //是f一个成员函数

A::f() //这是A 中f 的说明
{ ... }
```

类、过程(Ada中)以及记录结构都可以看作表示名字的作用域，把局部说明的集合作为一个属性。在这些作用域能被外部引用的情况下，为每个作用域建立一个单独的符号表是有好处的(就像在图6-13)。

至此我们随着程序的文本结构讨论了标准的作用域规则。有时称其为词法作用域(lexical scope)或静态作用域(static scope)(因为符号表静态地建立)。另一种作用域规则用于一些面向动态的语言(LISP、SNOBOL较早的版本，以及一些数据库查询语言)称作动态作用域(dynamic scope)。这个规则要求嵌套作用域的申请随着执行路径进行，而不是程序原来的安排。在程序清单6-5的C代码是一个简单的例子，说明两个规则的不同点。使用C语言标准的作用域规则这段代码打印出1，因为文件层变量**i**的作用域扩展到了过程**f**。如果使用动态作用域，程序就打印出2，因为**f**在中**main**调用，**main**包含了的说明(值为2)，如果使用顺序调用处理非终结符引用，它就扩展到**f**。动态作用域要求在执行时通过执行插入和删除操作建立，因为作用域也在运行时进入或退出。因此，使用动态作用域要求符号表变成环境的一部分，由编译器产生代码来维护它，而不是由编译器直接(静态地)建立符号表。动态作用域也破坏了程序的可读性，因为不模拟程序的执行就不能解决非局部的引用。最后因为变量的数据类型必须用符号表来维护，所以动态作用域与静态类型检查不兼容(注意在程序清单6-5的代码中，如果**main**内的**i**说明成**double**所出现的问题)。因此，在现代语言中动态作用域很少使用，我们不再进一步讨论。

程序清单6-5 说明静态和动态作用域不同之处的代码

```
#include <stdio.h>

int i = 1;

void f ( void )
{ printf ( " %d\n ", i ) ; }

void main ( void )
{ int i = 2;
  f ( );
  return 0;
}
```

最后，我们要注意，在图6-12中的删除操作从符号表中完全消除说明。事实上有必要在表中保留说明(或至少不剥夺它们的存储区)，因为编译器的其他部分在后面可能需要引用。如果它们必须保存在符号表中，那么删除操作只需要仅仅把它们标记成不活动的，而查找操作在搜索符号表时跳过这些标记的说明。

6.3.4 同层说明的相互作用

关于作用域更深一步的问题是在同一嵌套层中 (即相连到一块)说明的相互作用。对不同的说明和要翻译的语言，这些变化很大。许多语言中 (C、Pascal、Ada)一个典型的要求是在同一层中说明不能使用相同的名字。因此，在C中，下面连续的说明将引起编译错误：

```
typedef int i;
int i;
```

为检查这个要求，在每次插入前编译器必须执行一次查找，通过某种机制 (如嵌套层)确定在同一层中任何已存在的说明是否有相同的名字。

更困难的是在相同层的序列中名字相互之间有多少可用的信息。例如，考虑下面的 C代码片段

```
int i = 1;

void f ( void )
{ int i = 2, j = i+1;
  ...
}
...
```

这里有一个问题：**f**内部**j**的值是初始化成2还是3，即使用的是**i**的局部说明还是非局部说明。根据最近嵌套规则，应该是使用最近的说明—局部说明。事实上这是 C的方法。但是这预示在处理时每个说明加进符号表，称作顺序说明 (sequential declaration)。可以替代所有要“同时”处理的说明，在说明部分的最后立即加进符号表。然后说明中任意表达式的名字将引用前面的说明，不再处理新的说明。这样的说明结构称作并列说明 (collateral declaration)，一些函数式语言，像ML和Scheme，有这样的说明结构。这样的说明规则要求说明不立即加进存在的符号表中，而累加进一个新的表中(或临时结构)，在处理完所有的说明之后再加进现存的表中。

最后，是递归说明(recursive declaration)结构的情况，说明可以引用其自身或相互引用。这对于过程/函数说明特别必要，相互递归函数组是公共的(举例来说，在递归下降分析程序中)。在最

简单的形式中，递归函数调用其自身，如在下面的C代码中的函数计算两个整数的最大公因子：

```
int gcd ( int n, int m )
{ if ( m == 0 ) return n;
  else return gcd ( m,n % m );
}
```

要使能正确编译它，编译器必须在处理函数体之前把函数名 **gcd**加进符号表。否则，当递归调用遇到**gcd**时就找不到这个名字(或含义不正确)。在更复杂的情况中，有一组互相递归调用的函数，如下面的C代码片段

```
void f ( void )
{... g ( ) ... }

void g ( void )
{... f ( ) ... }
```

在处理函数体之前只是把每个函数加进符号表是不够的。说明上面的 C代码编译时在**f**内部调用**g**甚至会产生错误。在C语言中对这个问题的解决是在**f**的说明之前为**g**加上一个称作函数原型(fuction prototype)的说明：

```
void g ( void ); /函数原型说明 */

void f ( void )
{ ... g ( ) ... }

void g ( void )
{... f ( ) ... }
```

这样的说明可以看作是作用域修正(scope modifier)，扩展名字**g**的作用域使其包含**f**。因此，当到达**g**的(第1个)原型说明时(与它自己的位置作用域属性一起)，编译器把**g**加进符号表。当然在到达**g**的主说明(或定义)之前**g**的函数体一直是不存在的；而且， **g**的所有原型必须进行类型检查以确保在结构上的统一。

相互递归问题还有不同的解决办法。例如，在 Pascal中提供了向前(forward)说明作为过程/函数作用域的扩展。在Moudle-2中，过程和函数(还有变量)的作用域规则扩展它们的作用域包含整个说明块，这样就自然进行相互递归调用而不必使用另外的语言机制。这就要求一个处理步骤，所有的过程和函数在处理其主体的任何部分之前加进符号表。类似的相互递归调用说明也可用于一些其他的语言。

6.3.5 使用符号表的属性文法的一个扩充例子

现在考虑一个例子，来演示我们已描述过的说明的一些特性，并研究一个属性文法使这些特性在符号表的行为清晰呈现。这个例子中使用的文法是浓缩了简单算术表达式文法，包括了对说明的扩充：

S → *exp*
exp → **(** *exp* **)** | *exp* **+** *exp* | ***id*** | ***num*** | **let** *dec-list* **in** *exp*
dec-list → *dec-list* **,** *decl* | *decl*
decl → ***id*** **=** *exp*

因为属性文法包含层次属性，因此就需要在语法树的根节点进行初始化，这个文法包括一个顶层的开始符号*S*。这个文法只有一个操作符 (加，记号为**+**)，所以非常简单。它也是二义性的，我们假定分析器已经构造了语法树，或者已经处理了二义性 (我们把等价的无二义性文法留作练习)。不过可以包含括号，这样如果愿意，就可以用这个文法写出无二义性的表达式。就像

在前一个例子中使用的类似的文法，我们假定 ***num***和***id*** 是记号，其结构由扫描器确定 (假定***num***是一个数字序列，***id*** 是字符序列)。

包含说明的文法增加的是*let* 表达式(let *exp*ression)：

exp → **let** *dec-list* **in** *exp*

在*let* 表达式中，说明由通过逗号分开的、形如***id* = *exp***的说明序列组成，一个例子是

```
let x = 2+1, y = 3+4 in x + y
```

非正式地，*let* 表达式的语义如下。**let**记号后面的说明是建立表达式的名字，当这些名字出现在记号**in**后面的*exp*中时代表它们表示的表达式的值 (*exp*是*let*表达式的主体(body))。**let**表达式的值是主体的值，其计算方法是用相应 *exp*的值代替说明中的每个名字，并根据语义规则计算主体的值。例如，在前一个例子中，**x**代表值3**(2+1**的值)，**y**代表值7**(3+4**的值)。因此，表达式自身的值是10(等于**x+y**的值，**x**的值是3，**y**的值是7)。

从刚给出的语义中，我们看到说明在 *let* 表达式中表示一种常量说明 (或约束)，而*let* 表达式表示这个语言的块。为完成这些表达式语义的非正式讨论，需要描述 *let* 表达式中的作用域规则和说明的相互作用。注意，这个文法允许仲裁 *let* 表达式相互之间的嵌套，例如在表达式

```
let x = 2, y = 3 in
  ( let x = x+1, y = ( let z=3 in x+y+z )
    in ( x+y )
  )
```

中，我们为*let* 表达式的说明建立下列作用域规则。首先，在相同的 *let* 表达式中不能说明相同的名字，因此，形如

```
let x=2, x=3 in x+1
```

的表达式是非法的，将导致出错。其次，如果任意一个名字没有在某个外围 *let* 表达式中说明，也将导致错误。因此，表达式

```
let x=2 in x+y
```

是错误的。再次，在*let* 表达式中每个说明的作用域按照块结构的最近嵌套规则扩充到 *let* 的主体之外。因此，表达式

```
let x=2 in ( let x=3 in x )
```

的值是3而不是2 (因为在内层的*let* 表达式中的**x**引用说明**x=3**，而不是说明**x=2**)。

最后。在顺序的相同 *let* 表达式说明的列表中，我们说明了说明的相互作用。即每个说明使用前一个说明来处理其表达式中的名字。因此，在表达式

```
let x=2,y=x+1 in ( let x=x+y, y=x+y in y )
```

中，第一个**y**的值是3(使用前一个**x**的说明)，第2个**x**的值是5 (使用括起来的*let* 的说明)，第2个**y**的值是8 (使用括起来的**y**的说明和刚说明**x**的值)。因此，整个表达式的值是8。类似地我们请读者计算前面三重嵌套的*let* 表达式的值。

现在我们要开发属性等式，使用符号表记录 *let* 表达式中的说明并表示刚描述的作用域规则和相互作用。为简单起见，我们仅用符号表确定表达式是否是错的。我们不写出计算表达式的值的等式，而把它留作练习。作为替代，我们计算布尔值的合成属性 *err*，根据前面说明的规则，如果表达式是错的其值为 **true**，如果表达式正确其值为 **false**。为实现这一点，需要两个继承属性，*symtab*表示符号表，*nestlevel* 确定两个说明是否在相同的*let* 块内。*nestlevel* 的值是一个非负整数，表示块的当前嵌套层。在最外层它的值初始化为 0。

symtab 属性需要一般的符号表操作。因为要写属性等式，在某种程度上表达符号表操作无须间接的结果，编写插入操作像参数一样操作符号表，返回一个新的符号表加入新的信息，而原始的符号表没有改变。因此， *insert* (*s*, *n*, *l*) 返回一个新的符号表，包含来自符号表*s* 的所有信息，另外把名字*n* 与嵌套层*l* 相联系，而不改变*s* (因为我们只确定正确性，不必要联系*n* 的值，只是一个嵌套层)。因为这个说明保证能恢复原始的符号表 *s*，就无须一个明确的删除操作。最后，为了测试必须满足正确性的两个准则 (出现在表达式中的所有名字必须在前面说明，而且在同层中不出现重复说明)，必须能测试符号表中一个名字的出现，也能取出与出现的名字相关的嵌套层。用两个操作实现这一点， *isin* (*s*, *n*)返回一个布尔值，决定 *n* 是否在符号表*s*中，*lookup* (*s*, *n*)返回一个整数值，如果存在它给出 *n* 最近说明的嵌套层，或者如果 *n* 不在符号表*s* 中(这将允许使用*lookup*表示等式，而不首先执行*isin* 操作)其值为－1。最后，必须说明初始符号表中没有入口，我们把这写作*emptytable*。

对符号表使用这些操作和转换，现在写出表达式的 *symtab*、*nestlevel* 和*err* 3个属性的属性等式。完整的属性文法在表6-9中。

表6-9 带有let块的表达式的属性文法

文法规则	语义规则
$S \rightarrow exp$	$exp.symtab = emptytable$ $exp.nestlevel = 0$ $S.err = exp.err$
$exp_1 \rightarrow exp_2 + exp_3$	$exp_2.symtab = exp_1.symtab$ $exp_3.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_3.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$ **or** $exp_3.err$
$exp_1 \rightarrow$ **(** exp_2 **)**	$exp_2.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$
$exp \rightarrow$ **id**	$exp.err =$ **not** *isin* (*exp.symtab*, **id**.*name*)
$exp \rightarrow$ **num**	$exp.err =$ **false**
$exp_1 \rightarrow$ **let** *dec-list* **in** exp_2	*dec-list.intab* = $exp_1.symtab$ *dec-list.nestlevel* = $exp_1.nestlevel + 1$ $exp_2.symtab$ = *dec-list.outtab* $exp_2.nestlevel$ = *dec-list.nestlevel* $exp_1.err$ = (*dec-list.outtab* = *errtab*) **or** $exp_2.err$
$dec\text{-}list_1 \rightarrow dec\text{-}list_2$ **,** *decl*	$dec\text{-}list_2.intab = dec\text{-}list_1.intab$ $dec\text{-}list_2.nestlevel = dec\text{-}list_1.nestlevel$ *decl.intab* = $dec\text{-}list_2.outtab$ *decl.nestlevel* = $dec\text{-}list_2.nestlevel$ $dec\text{-}list_1.outtab$ = *decl.outtab*
dec-list → *decl*	*decl.intab* = *dec-list.intab* *decl.nestlevel* = *dec-list.nestlevel* *dec-list.outtab* = *decl.outtab*

（续）

文法规则	语义规则
decl → **id** = *exp*	*exp.symtab* = *decl.intab* *exp.nestlevel* = *decl.nestlevel* *decl.outtab* = **if** (*decl.intab* = *errtab*) **or** *exp.err* **then** *errtab* **else if** *lookup*(*decl.intab*, **id**.*name*) = *decl.nestlevel*) **then** *errtab* **else** *insert*(*decl.intab*, **id**.*name*,*decl.nestlevel*)

在最顶层，分配了两个继承属性值而留下了合成属性的值。因此，文法规则 S →*exp*有3个相关的属性等式

$$exp.symtab = emptytable$$
$$exp.nestlevel = 0$$
$$S.err = exp.err$$

文法规则*exp* →(*exp*)也有类似的规则。

对于规则exp_1 →exp_2 **+** exp_3(像通常一样，在写属性等式时对非终结符编号)，下面的规则表示右边的表达式从左边的表达式继承了属性 *symtab*和*nestlevel*，并且如果右边的表达式包含了至少一个错误，左边的表达式也就包含一个错误：

$$exp_2.symtab = exp_1.symtab$$
$$exp_3.symtab = exp_1.symtab$$
$$exp_2.nestlevel = exp_1.nestlevel$$
$$exp_3.nestlevel = exp_1.nestlevel$$
$$exp_1.err = exp_2.err \textbf{ or } exp_3.err$$

仅当在当前的符号表中不能找到名字**id**时，规则*exp* → **id**产生一个错误(我们把标识符的名字写成**id**.*name*，并假定它由扫描器或分析程序计算)，这样相关的属性等式是

$$exp.err = \textbf{not } isin\ (exp.symtab, \textbf{id}.name)$$

另一方面，规则*exp* → **num** 永远不会产生错误，因此属性等式是

$$exp.err = \texttt{false}$$

现在开始讨论let表达式，使用文法规则

$$exp_1 \rightarrow \texttt{let}\ dec\text{-}list\ \texttt{in}\ exp_2$$

和相关的说明规则。在let表达式的规则中，*dec-list*由一系列必须加进当前符号表的说明组成。通过用*dec-list*联系两个独立的符号表来表示这一点：从 exp_1 继承的输入表*intab*，以及输出表*outtab*，它包含必须传递到exp_2的新的(和旧的)说明。因为新的说明可能包含错误(如重复说明相同的名字)，必须考虑从*dec-list* 传递的一个特别的*errtab*符号表。最后，当*dec-list* 包含一个错误(这种情况*outtab*出错)或者*let* 表达式的主体exp_2 包含一个错误，*let* 表达式也出错。属性等式是

$$dec\text{-}list.intab = exp_1.symtab$$
$$dec\text{-}list.nestlevel = exp_1.nestlevel + 1$$
$$exp_2.symtab = dec\text{-}list.outtab$$
$$exp_2.nestlevel = dec\text{-}list.nestlevel$$
$$exp_1.err = (dec\text{-}list.outtab = errtab) \textbf{ or } exp_2.err$$

注意当进入*let* 块时嵌套层也增加1。

还剩下为说明列表和单个的说明开发等式。根据说明的顺序性规则，说明列表必须在处理列表过程中累积。因此，给定规则

$$dec\text{-}list_1 \rightarrow dec\text{-}list_2 \textbf{ , } decl$$

像*intab*传递到*decl*一样，*outtab*从$dec\text{-}list_2$ 传递，因此给定*decl*访问说明先于它在列表中。对于单个说明的情况($dec\text{-}list \rightarrow decl$)，执行了标准的继承和移位。完整的等式在表6-9中。

最后，讨论单个说明的情况

$$decl \rightarrow \textbf{\textit{id}} = exp$$

在这个情况中继承属性*decl.intab*立即传递到*exp*(因为在这个语言中说明是非递归的，因此*exp*必须找到这个***id*** 名字的前一个说明，而不是当前的那个)。然后，如果没有错误，***id***.*name*用当前嵌套层插入到表中，并作为说明继承*outtab*传递回去。在3种情况会出现错误。首先，在前面的说明中已经有了错误，这种情况是： *decl.intab*是*errtab*，*errtab*必须作为*outtab*传递。其次，错误可能在*exp*中出现：如果是这样，则由*exp.err*指出，*outtab*也会引起变成*errtab*。再次，说明可能是一个名字在同一嵌套层次中的重复说明。这必须通过执行一次 *lookup*来检查，这个错误也必须把*outtab*强制变成*errtab*来报告(注意，如果*lookup*没有找到***id***.*name*，则返回－1，在当前的嵌套层没有匹配，因此没有错误产生)。*decl.outtab*完整的等式在表6-9中给出。

这就完成了属性等式的讨论。

6.4 数据类型和类型检查

编译器的主要任务之一是数据类型信息的计算和维护(类型推论(type inference))以及使用这些信息确保程序的每一部分在语言的类型规则作用下有意义(类型检查(type checking))。通常地，这两个任务密切相关并一起执行，但只提及类型检查。数据类型信息可以是静态的或动态的或是两者的混合。在大多数LISP语系的语言中，类型信息完全是动态的。在这样的语言中，编译器必须在执行时产生代码完成类型推论和类型检查。在大多数传统语言中，如Pascal、C和Ada，类型信息主要是静态的，主要在程序执行之前进行正确性检查。静态类型信息也用来确定每个变量分配所需要的存储器大小以及存储器的访问方式，这可以用来简化运行环境 (将在下一章讨论这一点)。这一节将只关心静态数据类型。

数据类型信息可以用几种不同的形式出现在程序中。在理论上，数据类型 (data type)是值的集合，更精确一点，是那些值上某几种操作的值的集合。例如，数据类型 **integer**在一个编程语言中指的是数学整数的子集，以及算术操作，如+和*，由语言说明提供。在编译器构造的实际领域，这些集合通常用类型表达式(type expression)描述，有一个类型名，如**integer**，或结构表达式，如**array [1..10] of real**，其操作通常假定或隐含。类型表达式在一个程序中可能出现几次。这些表达式包括变量说明，如

```
var x: array [1..10] of real;
```

它把类型与一个变量名和类型说明相关，又如

```
type RealArray = array [1..10] of real;
```

它说明一个新的类型名，用于以后的类型或变量说明。这样的类型信息是明确的。类型信息也可能是隐含的，如Pascal中常量说明的例子

```
const greeting = "Hello!";
```

这里根据Pascal的规则，**greeting**隐含说明为**array [1..6] of char**类型。

包含在说明中显式或隐含定义的类型信息保持在符号表中，当引用相关的名字时，由类型检查器取出，新的类型则从这些类型中推断出来，并和语法树中相应的节点关联。例如，在表达式

```
a [ i ]
```

中名字**a**和**i**的数据类型从符号表中取出，如果**a**的类型是**array [1..10] of real**，而**i**的类型是**integer**，那么子表达式**a[i]**的类型为**real**，并被判断为正确的(**i**的值是否在1和10之间的问题是范围检查(range checking)问题，不能像通常一样静态确定)。

编译器表示数据类型的方式、符号表维护类型信息的方式以及类型检查器推断类型使用的规则等，都依赖于语言中可用的类型表达式的种类和语言中管理这些类型表达式使用的类型规则。

6.4.1 类型表达式和类型构造器

编程语言通常包含一些内嵌的类型，如**int**和**double**。这些预说明(predefined)类型或者对应于由各种机器体系结构内部提供的数字数据类型，其操作作为机器指令已经存在，或者是像**boolean**或**char**一样属性很容易实现的基本类型。这些数据类型是简单类型 (simple type)，其值呈现为无明确的内部结构。对于整数一种典型的表示是 2字节，或4字节作为2字节的补充形式。实数或浮点数的典型表示，是 4或8字节数，带一个符号位，一个指数字段和一个分数(或尾数)字段。字符的典型表示是一字节的 ASCII代码，对布尔值是一个字节，只使用最低的一位(1=true，0=false)。有时语言也在如何实现这些预说明类型上加强限制。例如， C语言标准要求**double**浮点数类型至少有10位十进制数字的精度。

在C语言中一种有趣的预说明类型是**void**类型。这个类型没有值，因此表示空的集合。它用于表示没有返回值的函数(即过程)，也可表示一个指针指向未知类型。

在一些语言中，可以说明一些新的数据类型。典型的例子是子界类型 (subrange type)和枚举类型(enumerated type)。例如，在Pascal语言中由0~9组成的整数的子界类型可以说明为

```
type Digit = 0..9;
```

在C语言中由名字为**red**、**green**和**blue**组成的枚举类型说明为

```
typedef enum { red, green, blue } Color;
```

子界和枚举都可以作为整数实现，或者使用较小的足以表示所有值的存储器。

给定一个预说明类型的集合，使用类型构造器 (type constructor)，如数组(**array**)、记录(**record**)和结构(**struct**)，可以创建新的类型。这些构造可以看作是函数，把存在的类型作为参数，而用依赖于构造的一个结构返回新的类型。这些类型通常称作结构类型 (structured type)。在这些类型的分类中，了解类型表示的值的集合的特性是重要的。通常，在类型构造的参数值的基本集上，它密切对应于一组操作。我们通过列出一些公共的构造并把它们与集合操作比较来说明这一点。

1) **数组** 数组类型构造有两个类型参数，一个是索引类型(index type)，另一个是元素类型(component type)，并产生一个新的数组类型。在类Pascal语言中我们写作

array [索引类型**] of** 元素类型

例如，Pascal类型表达式

```
array [Color] of Char;
```

创建一个数组类型，其索引类型是**Color**，元素类型是**Char**。通常对于索引类型有一些限制。例如，在Pascal语言中索引类型限制为所谓的序数类型(ordinal types)：这些类型的每个值都有一个直接前驱和直接后继。这样的类型包括整数和字符的子界类型以及枚举类型。对照地在C语言中，整数作用域只允许从0开始，并只能指定大小来代替索引类型。事实上在C语言中没有关键字对应于**array**，只是仅仅加上在括号表示作用域的后缀来说明数组类型。因此对前面Pascal的类型表达式在C中没有直接的等式，但C的类型说明

```
typedef char Ar [3];
```

说明了**Ar**类型，是与前面的类型等价的类型结构(假定**Color**有3个值)。

数组表示的是元素类型的值的序列，并由索引类型的值进行索引。也就是说，如果索引类型有值I的集合，元素类型有值C的集合，那么对应于类型**array** [索引类型] **of** 元素类型值的集合是通过I的元素索引的C的元素有限序列的集合，或者在数学项中，函数I→C的集合。数组类型的值的相关操作由单个下标的操作组成，它可以用于给元素赋值或从元素中取出值：**x:= a[red]**或**a[blue] := y**

数组一般根据索引从小到大分配连续的存储空间，允许在执行期间使用自动的偏移量计算。所需存储空间的大小是*n***size*，这里*n* 是在索引类型中值的数目，*size*是元素类型的一个值所需存储器的大小。因此，如果每个整数占据4个字节，类型**array [0..9] of integer**的一个变量需要40字节的存储空间。

数组说明中的一种复杂情况是多维数组(multidimensioned arrays)。这通常可以通过重复应用数组类型构造来说明，如

```
array [0..9] of array [Color] of integer
```

或者进行简化，把索引集合列在一起：

```
array [0..9,Color] of integer
```

在第1种情况中出现的下标如**a[1][red]**，第2种情况下写成**a[1,red]**。多重下标的一个问题是在存储器中值的序列可以用不同的方式，索引过程是：首先是第一个索引，然后是第2个索引，或者相反。用第一种索引方式的结果是在存储器中值的顺序是**a[0,red]**，**a[1,red]**，**a[2,red]**，**.....**，**a[9,red]**，**a[0,blue]**，**a[1,blue].....**等等(这称作列前提形式(column-major form))，用第2种索引方式的结果是在存储器中值的顺序是**a[0,red]**，**a[0,blue]**，**a[0,green]**，**a[1,red]**，**a[1,blue],a[1,green].....**等等(这称作行前提形式(row-major form))。如果多维数组重复说明的版本和简化版本是等价的(也就是说**a[0,red]= a[0][red]**)那么必须使用行前提形式，因为可以分开加上不同的索引：**a[0]**的类型必须是**array[Color]of integer**而且必须指向一个连续的存储区。FORTRAN语言，没有多维数组的局部索引，传统上使用列前提形式实现。

有时候，语言允许使用没有说明索引作用域的数组。这样的开放索引数组(open-indexed array)对函数的数组参数说明特别有用，这样函数可以处理不同大小的数组。例如，C语言说明

```
void sort ( int a[], int first, int last )
```

可以用来说明一个分类程序，用于任意大小的数组 **a** (当然，在调用时必须使用一些方法来确定实际的大小。在这个例子中，使用了其他的参数)。

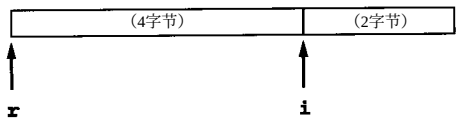
2) **记录** 记录(record)或结构(structure)类型构造器接受一个名字列表和相关的类型并构造一个新的类型。如在C语言中

```
struct
{ double r;
  int i;
}
```

记录与数组不同，不同类型的元素可以组合起来 (在数组中所有的元素都有相同的类型)，使用名字(而不是索引)访问不同的元素。记录类型的值大致对应于其元素类型值的笛卡儿积，使用名字而不是位置访问元素。例如，前面给定的记录大致对应于笛卡儿积 $R\times I$，这里R是相应的**double**类型的数据集合，I 是相应的**int**类型的数据集合。更准确地，给定的记录对应于笛卡儿积(**r**$\times R$)$\times$(**i**$\times I$)，这里名字**r**和**i**识别各个元素。这些名字通常使用圆点符号 (dot notation)选择表达式中相应的元素。因此，如果**x**是给定记录类型的一个变量，那么**x.r**表示第1个元素，**x.i**表示第2个元素。

一些语言具有纯的笛卡儿积类型构造器。这样的一种语言是 ML，这里**int*real**是整数和实数笛卡儿积的符号。类型**int*real**的值写成两个一组，如**(2, 3.14)**，元素通过投影函数**fst**和**snd** (表示第1和第2)访问：**fst(2,3.14) = 2**和**snd(2,3.14) = 3.14**。

记录或笛卡儿积的标准实现方法是顺序地分配存储器，为每个元素类型分配一块存储器。因此，如果一个实数需要4个字节，一个整数需要两个字节，那么前面给定的记录结构需要 6个字节的存储器，分配如下：

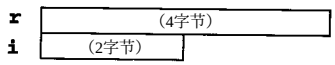


3) **联合** 联合类型对应于联合操作集合。在C语言中它可以直接通过**union**说明使用，例如说明

```
union
{ double r;
  int i;
}
```

说明了实数和整数的联合类型。严格地讲，这是一个脱节的联合 (disjoint union)，因为每个值可以看成是实数或整数，但不能同时是两者。通过访问值的元素名字可以解释的更清楚一些：如果x是给定的联合类型的一个变量，那么 **x.r**表示**x**是实数时的值，而**x.i**表示**x**是整数时的值。在数学上联合的说明写成(**r**$\times R$) (**i**$\times I$)。

联合的标准实现方法是为每个元素并行地分配存储器，这样每个元素类型的存储器与所有其他的类型相重叠。因此，如果一个实数需要 4个字节，一个整数需要两个字节，那么前面给定的联合结构仅需要4个字节的存储器(其元素所需的最大的存储器)，存储器分配如下：



事实上，这样的实现需要联合被解释成脱节的联合，因为整数的表示不会适合相应的实数的表示。实际上，编程者区分这些值没有什么意义，这样的联合在数据解释中会导致错误，也提供了一种方法绕过类型检查器。例如，在C语言中如果**x**是给定的联合类型的一个变量，那么

```
x.r = 20;
printf ( "%d",x.i );
```

将引起一个编译错误，不会打印出值2，而是一个垃圾值。

在联合类型中的这种不安全性已由许多不同的语言设计者处理。例如在 Pascal中，联合类型使用一种所谓的可变记录 (variant record) 说明，其中序数类型的值记录在一个判别式(discriminant)元素中并用来区分想要的值。因此，在Pascal中前面的联合类型可以写成

```
record case isReal: boolean of
  true: (r: real);
  false: (i: integer);
end;
```

现在这个类型的变量**x**有3个元素：**x.isReal**(布尔值)、**x.r**和**x.i**，并且**isReal**字段在存储器中分配一个独立的非覆盖的空间。当赋一个实数值时，如 **x.r := 2.0**，同时也要赋值**x.isReal := true**实际上，这个机构相对而言是没有用的(至少在类型检查时编译器的使用)，因为判别式的赋值和值的辨别可以分开。当然， Pascal允许通过在说明中删除其名字说明判别式元素(但不使用其值区分case)，如在

```
record case bollean of
  true:(r: real);
  false:(i: integer);
end;
```

中不再为判别式分配存储空间。因此， Pascal编译器在合法性检查时几乎从不进行任何尝试使用判别式。另一方面， Ada有类似的机制，但是强调只要一个联合元素被赋值，判别式必须同时赋值进行合法性检查。

在像ML这样的函数式语言中采用了一种不同的方法，联合类型的说明使用了一根竖线表示联合，为每个元素给定一个名字来区分它们，如：

```
IsReal of real | IsInteger of int
```

现在当使用相应的类型的值时也必须使用名字**IsReal**和**IsInteger**，就像在(**IsReal 2.0**)或(**IsInteger 2**)中一样。名字**IsReal**和**IsInteger**称作值构造器(value constructor)，因为它们“构造”了这个类型的值。因为当它们参照这个类型的值时总必须使用，不会有解释错误发生。

4) **指针** 指针类型由引用另一个类型值的值组成。因此，指针类型的值是一个存储器地址，其中保存着其基类型的值。指针类型经常被看成是数字类型，因为在其上可以进行算术运算，如加上偏移量，乘上比例因子等。然而它们不是真正的简单类型，因为它们是应用指针类型构造器从已有的类型中构造出来的。它也没有标准的集合操作直接对应于指针类型构造器，就像笛卡儿积对应于记录构造器一样。指针类型在类型系统中占有比较特殊的位置。

在Pascal中字符**^**对应于指针类型构造器，因此类型表达式**^integer**表示“整数的指针”。在C中，等价的类型表达式是**int***。指针类型值上的标准基本操作是解除引用 (dereference)操作。例如，在Pascal中，**^**表示解除引用操作符(和指针类型构造器)，如果**p**是类型**^integer**的一个变量，那么**p^**是**p**解除引用的值，类型为**integer**。C中也有类似的规则，*解除指针变量的引用，并写成***p**。

指针类型在描述递归类型时最有用，我们简要地进行讨论。在这里，指针类型构造器最常用于记录类型。

指针类型基于目标机器的地址的大小分配空间。通常是4字节，有时是8字节。有时机器的体系结构强制更复杂的分配方案。例如，在基于DOS的PC中，要区别近指针(段内地址，2字节)和远指针(段外地址，4字节)。

5) **函数** 我们已经注意到数组可以看成从索引集到元素集的函数。许多语言(但不是Pascal或Ada)都有描述函数类型更一般的能力。例如在Modula-2中说明

```
VAR f: PROCEDURE (INTEGER): INTEGER;
```

说明变量**f**是函数(或过程)类型，带有一个整数参数，并产生一个整数结果。在数学符号中，这个集合描述成函数$\{f: I \to I\}$，这里I是整数的集合。在ML语言中，相同的类型被写成**int->int**。C语言也有函数类型，但它们必须用有些笨拙的符号写成“指向函数的指针”。例如，刚给出的Modula-2的说明写成C语言是

```
int (*f) (int);
```

函数类型按照目标机器的地址的大小分配空间。根据语言和运行时环境的组织方式的不同，函数类型需要给代码指针分配空间(指向实现函数的代码)或给代码指针和环境指针分配空间(指向运行环境中的位置)。环境指针的作用将在下一章讨论。

6) **类** 大多数面向对象的语言都有类似于记录说明的类说明，它所包含的操作说明除外，那称作方法(method)或成员函数(member function)。类说明在一个面向对象的语言中可以创建或不创建新的类型(在C++中创建)。即使这样，类说明不仅仅是类型，因为它们允许使用属于类型系统的特性，如继承和动态联编⊖。这些后期的特性必须通过独立的数据结构维护，如类继承(class hierarchy)(直接非循环图)，用于实现继承性，以及虚拟方法表(virtual method table)，用于实现动态联编。下一章我们将再次讨论这些结构。

6.4.2 类型名、类型说明和递归类型

具有丰富类型构造器的语言通常也给编程者提供一个机制给类型表达式赋名。这样的类型说明(type declaration)(有时也称作类型说明(type difinition))包括C语言中的***typedef***机制和Pascal语言中的类型说明。例如C语言代码

```
typedef struct
      { double r;
        int i;
      } RealIntRec;
```

说明名字**RealIntRec**作为记录类型的名字，它由在其之前的**struct**类型表达式构造。在ML语言中，类似的说明是(但没有字段名)：

```
type RealIntRec = real*int;
```

C语言有附加的类型命名机制，名字可以直接用**struct**或**union**构造器关联，而无须直接使用**typedef**。例如，C代码

```
struct RealIntRec
      { double r;
```

⊖ 在一些语言中，如C++，继承是类型系统的镜像，因为子类可以看作是子类型(类型S可以看作是类型T的子类型，如果它所有的值都可看成是T的值，或者，在集合术语中，如果$S \quad T$)。

```
        int i;
      };
```

也说明了类型名**RealIntRec**，但它在变量说明中必须使用**struct**构造器名：

```
struct RealIntRec x; /*说明x 是一个RealIntRec类型的变量 */
```

就像变量说明使变量名进入符号表一样，类型说明也使说明的类型名进入符号表。产生的一个问题是是否类型名也像变量名样可以重用。通常这是不允许的 (作用域嵌套规则允许除外)。对这个规则C语言有一个小的例外，与**struct**或**union**相关的名字可以像**typedef**名字一样重用：

```
struct RealIntRec
      { double r;
        int i;
      };
typedef struct RealIntRec RealIntRec;
/* 一个合法的说明 */
```

通过考虑**struct**说明引入的类型名为整个字符串“ **struct RealIntRec**”可实现这一点，它与**typedef**引入的类型名**RealIntRec**不同。

类型名与符号表中属性相关的方法和变量说明相似。这些属性包括作用域 (它在符号表结构中可以继承)和对应于类型名的类型表达式。因为类型名可以出现在类型表达式中，与前一节讨论的函数的递归说明类似，就出现了类型名递归使用的问题。在现代的编程语言中这样的递归数据类型(recursive data type)特别重要，包括列表、树以及其他结构。

在处理递归类型方面，语言通常分成两组。第 1组由允许在类型说明中直接使用递归的语言组成。这样的一种语言是ML。例如。在ML中，包含整数的二叉搜索树可以说明为

```
datatype intBST = Nil | Node of int*intBST*intB
```

这可以看成**intBST**的说明，即是**Nil**值和整数与**intBST**自身两个拷贝(一个表示左子树，一个表示右子树)笛卡儿积的联合。等价的C语言说明(形式稍有改变)是

```
struct intBST
{ int isNull;
  int val;
  struct intBST left,right;
};
```

然而，这个说明在C语言中将产生一个错误消息，是由对类型名 **intBST**的递归使用引起的。问题是这些说明没有确定分配一个**intBST**类型的变量所需的存储器的大小。这一类语言，如ML，能接受这样的说明，在执行之前不需要这样的信息，提供一种一般的存储器自动分配和释放机制。这样的存储器管理工具是运行时环境的一部分，将在下一章讨论。 C语言没有这样的机制，因此必须使这样的递归类型说明是非法的。 C是第2组语言的代表—在类型说明中不允许直接使用递归。

对只允许间接使用递归的语言的解决办法是通过指针。在 C语言中**intBST**正确的说明是

```
struct intBST
{ int val;
  struct intBST *left,*right;
};
typedef struct intBST * intBST;
```

或

```
typedef struct intBST * intBST;
struct intBST
{ int val;
  intBST left, right;
};
```

(在C语言中，递归说明需要使用递归类型名说明的**struct**或**union**形式)。在这些说明中每个类型所需的存储空间大小直接由编译器计算，但值的空间必须由编程者通过使用 **malloc**这样的分配过程进行手工分配。

6.4.3 类型等价

给定语言可能的类型表达式，类型检查器经常需要回答何时两个类型表达式表示相同的类型。这就是类型等价(type equivalence)问题。一种语言有许多种可能的方法说明类型等价。这一节我们简要讨论类型等价最常用的形式。在这里，这样描述类型等价，当在编译器的语义分析程序中，即当函数

function *typeEqual* (*t*1, *t*2 **:** *TypeExp*) **:** *Boolean*;

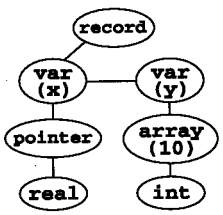
接受两个类型表达式，如果根据语言的类型等价规则它们表示相同的类型就返回 *true*，否则返回*false*。对于不同的类型等价算法，将给出这个函数的几种不同的伪代码描述。

一种有关直接描述类型等价算法的方法是类型表达式在编译器内表示。一种简单的方法是使用语法树表示，因为这使得从说明的语法直接转换到类型的内部表示十分容易。对于这样表示的一个具体例子，考虑图6-14给出的类型表达式和类型说明的文法。其中有我们已讨论过的许多类型结构的简单版本。但是没有允许关联新的类型名到类型表达式的类型说明 (因此不可能有递归类型，尽管出现了指针类型)。对应于图中的文法规则，要为类型表达式描述一个可能的语法树结构。

首先考虑类型表达式

```
record
  x: pointer to real;
  y: array [10] of int
end
```

这个类型表达式可以用下面的语法树表示



这里记录的子节点表示成同属列表，因为记录元素的数目是任意的。注意，表示简单类型的节点构成了树的叶子。

var-decls → *var-decls* **;** *var-decl* | *var-decl*

var-decl → ***id*** **:** *type-exp*

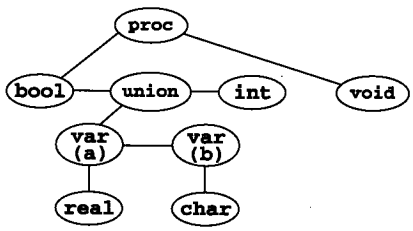
type-exp → *simple-type* | *structured-type*
simple-type → **int | bool | real | char | void**
structured-type→ **array** [***num***] **of** *type-exp*|
record *var-decls* **end** |
union *var-decls* **end** |
pointer to *type-exp* |
proc (*type-exps* **)** *type-exp*
type-exps → *type-exps* **,** *type-exp* | *type-exp*

图6-14 类型表达式的简单文法

类似地，类型表达式

```
proc ( bool, union a:real; b:char end, int ) : void
```

可以用以下语法树表示



注意，参数类型也给定为同属列表，而结果类型（在这个例子中是 **void**）通过使它直接成为**proc**的一个子节点来区分。

我们描述的第一种类型等价，也是仅有的可用于缺省类型名的，是结构等价 (structural equivalence)。在这个等价观点中，两个类型当且仅当它们有相同的结构时它们才相同。如果用语法树表示类型，这个规则说两个类型是等价的，当且仅当它们的语法树结构是同一的。在练习中有一个例子，说明如何检查结构等价。程序清单 6-6给出了函数*typeEqual*的伪代码描述，它是图 6-14中文法给出的两个类型表达式的结构等价，使用了我们刚描述过的语法树。

我们从程序清单6-6的伪代码描述中注意到这个版本的结构等价意味着两个数组是不等价的，除非它们有相同的大小和元素类型，两个记录是不等价的，除非它们有相同的元素并且元素有相同的名字和顺序。在一个结构等价算法中，可能有一些不同的选择。例如，在确定等价性时数组的大小可以被忽略，也可能允许结构或联合的元素以不同的顺序出现。

当在类型说明中说明了类型表达式新的类型名时，可以说明限制性更强的类型等价。在图6-15中，我们修改了图6-14的文法以包含类型说明，同时限制变量说明和类型子表达式为简单类型和类型名。对这些说明不能再写成

```
record
  x: pointer to real;
  y: array [10] of int
end
```

而必须代替为

```
t1 = pointer to real;
```

```
t2 = array [10] of int;
t3 = record
       x: t1;
       y: t2
end
```

程序清单6-6　函数typeEqual的伪代码，测试图6-14文法类型表达式的结构等价

```
function typeEqual ( t1, t2 : TypeExp ) : Bollean;
var temp : Boolean;
  p1, p2 : TypeExp;
begin
  if t1 and t2 are of simple type then return t1 = t2
  else if t1.kind = array and t2.kind = array then
    return t1.size = t2.size and typeEqual ( t1.child1, t2.child1 )
  else if t1.kind = record and t2.kind = record
      or t1.kind = union and t2.kind = union then
  begin
    p1 := t1.child1 ;
    p2 := t2.child1 ;
    temp := true ;
    while temp and p1≠nil and p2≠nil do
      if p1.name≠p2.name then
        temp := false
      else if not typeEqual ( p1.child1 , p2.child1 )
      then temp := false
      else begin
        p1 := p1.sibling ;
        p2 := p2.sibling ;
      end ;
    return temp and p1 = nil and p2 = nil ;
  end
  else if t1.kind = pointer and t2.kind = pointer then
    return typeEqual ( t1.child1 , t2.child1 )
  else if t1.kind = proc and t2.kind = proc then
  begin
    p1 := t1.child1 ;
    p2 := t2.child1 ;
    temp := true ;
    while temp and p1≠nil and p2≠nil do
      if not typeEqual ( p1.child1 , p2.child1 )
      then temp := false
      else begin
```

```
            p1 := p1.sibling ;
            p2 := p2.sibling ;
          end;
      return temp and p1 = nil and p2 = nil
             and typeEqual( t1.child2 , t2.child2 )
    end
    else return false;
end ; (* typeEqual *)
```

```
var-decls → var-decls ; var-decl | var-decl
var-decl → id : simple-type-exp
type-decls → type-decls ; type-decl | type-decl
type-decl → id = type-exp
type-exp → simple-type-exp | structured-type
simple-type-exp → simple-type | id
simple-type → int|bool|real|char|void
structured-type → array [num] of simple-type-exp|
                  record var-decls end |
                  union var-decls end |
                  pointer to simple-type-exp |
                  proc ( type-exps ) simple-type-exp
type-exps → type-exps , simple-type-exp | simple-type-exp
```

图6-15 带类型说明的类型表达式

现在可以说明基于类型名的类型等价，这种形式的类型等价称作名等价 (name equivalence)：两个类型表达式是等价的，当且仅当它们是相同的简单类型或有相同的类型名。这是一种非常强的类型等价，因为给定类型说明

```
t1 = int;
t2 = int
```

类型**t1**和**t2**是不等价的 (因为名字不同)对**int**也不等价。纯的名等价非常容易实现，因为*typeEqual*函数可写成以下几行：

```
function typeEqual( t1,t2 : TypeExp ) : Boolean;
var temp : Boolean ;
    p1, p2 : TypeExp ;
begin
  if t1 and t2 are of simple type then
    return t1 = t2
  else if t1 and t2 are type names then
    return t1 = t2
  else return false ;
end;
```

当然，对应于类型名的实际的类型表达式必须进入符号表，以允许后面为存储器分配计算存储器大小，以及检查操作的有效性，如指针解除引用和元素选择。

名等价中一个复杂的因素是类型表达式不同于简单类型，或类型名在变量说明中继续被使用，或者作为类型表达式的子表达式。在这些情况下，类型表达式可能没有给定明确的名字，编译器将产生一个类型表达式的中间名，与其他任何名字都不同。例如，给定变量说明

```
x: array [10] of int;
y: array [10] of int;
```

对应于类型表达式`array [10] of int`，变量`x`和`y`被赋予不同的(和唯一的)类型名。

在出现类型名时可能保留结构等价。对这种情况，当遇到一个名字时，必须从符号表中取出它对应的类型表达式。这可以通过在程序清单6-6的代码中加进下列情况来实现，

else if *t1 and t2 are type names* **then**
  return *typeEqual* (*getTypeExp*(*t1*) , *getTypeExp*(*t2*))

这里*getTypeExp*是一个符号表操作，返回与其参数(必须是一个类型名)相关的类型表达式结构。这要求每个类型名必须用表示其结构的类型表达式插入到符号表，或者至少由类型说明产生类型名的链，如

```
t2 = t1;
```

在符号表中最后带回类型结构。

当可能有递归类型引用时，实现结构等价必须小心，因为刚才描述的算法会导致无限循环。通过改变调用*typeEqual*(*t1* , *t2*)的方法可以避免这种情况，这里*t1*和*t2*是类型名，假定它们已经潜在地等价。然后如果函数曾经返回相同的调用，在那里就可说明成功。例如，考虑类型说明

```
t1 = record
       x: int;
       t: pointer to t2;
     end;
t2 = record
       x: int;
       t: pointer  to  t1;
     end;
```

给定调用*typeEqual* (*t1* , *t2*)，函数*typeEqual* 将假定*t1*和*t2*潜在地等价。然后取出*t1*和*t2*的结构，并且算法将成功进行直到调用*typeEqual* (*t2* , *t1*)分析指针说明的子孙类型。这个调用将立即返回true，因为在初始调用中已经假定了它们潜在等价。通常，这个算法需要进行成功地假设，那一对类型名是相等的，并在一个列表中累积假设。最后，当然，算法或者成功，或者失败(即它不会无限循环)，因为在任何给定的程序中只有有限的类型名。我们把 *typeEqual*伪代码的修改细节留作练习(见注意与参考节)。

类型等价最后一个变化是 Pascal和C使用的名等价的一个弱化的版本，称作说明等价(declaration equivalence)。在这个方法类型中，像

```
t2 = t1;
```

这样的说明是作为类型别名(aliase)解释的，而不是新的类型(作为名等价中)。因此，给定说明

```
t1 = int;
t2 = int
```

t1和**t2**对**int**等价(即它们仅是类型名**int**的别名)。在这个类型等价版本中，每个类型名等价于某个基类型名，它或者是一个预说明类型，或者是由类型构造器产生的类型表达式给定的。例如，给定说明

```
t1 = array [10] of int;
t2 = array [10] of int;
t3 = t1;
```

类型名**t1**和**t3**根据说明等价是等价的，但和**t2**都不等价。

为实现说明等价，符号表必须提供一个新的操作 *getBaseTypeName*，它取出基类型名而不是相关的类型表达式。在符号表中，一个类型名如果是预说明类型或由类型表达式给出，而不只是另一个类型名，它就被区分为基类型名。注意，说明等价类似于名等价，在检查递归类型时解决无限循环问题，因为如果两个基类型名有相同的名字只能是说明等价。

Pascal一律使用说明等价，而 C对结构和联合使用说明等价，但对指针和数组使用结构等价。

有时，一种语言将提供结构、说明或名等价，对不同的类型说明使用不同形式的等价。例如，ML语言允许使用保留关键字**type**把类型名说明为别名，如说明

```
type RealIntRec = real*int;
```

这把**RealIntRec**说明成笛卡儿积类型**real*int**的别名。另一方面，说明完全创建了一个新的类型，如说明

```
datatype intBST = Nil | Node of int*intBST*intBST
```

注意，**datatype**说明也必须包含值构造器名(在给定的说明中是**Nil**和**Node**)。而不像**type**说明。这使新类型的值能从已存在的类型的中区分出来。因此，给定说明

```
datatype NewRealInt = Prod of real*int;
```

值**(2.7,10)**是类型**RealIntRec**或**real*int**的，而值**Prod(2.7,10)**是类型**NewReal-Int**的(而不是**real*int**)。

6.4.4 类型推论和类型检查

现在，基于类型的表示和前一节讨论的*typeEqual*操作，我们对一个简单语言的语义分析动作方面的类型检查器进行描述。使用的语言具有图 6-16给定的文法，包括图6-14中类型表达式的一个小的子集，加上了少量的表达式和语句。我们还假定符号表的可用性包括变量名和相关的类型，插入操作，在表中插入名字和类型，及查找操作，返回名字的相关类型。在属性文法中我们将不指定这些操作本身的特性。我们将分别讨论每种语言构造的类型推断和类型检查规则。语义动作的完整列表在表6-10中给出。这些动作没有用纯的属性文法形式给出，并且使用符号:=而不是表6-10规则中的等号来指示。

program → *var-decls* **;** *stmts*
var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → ***id*** **:** *type-exp*
type-exp → **int|bool|array [***num***]of** *type-exp*
stmts → *stmts* **;** *stmt* | *stmt*
stmt → **if** *exp* **then** *stmt* | ***id*** **:=** *exp*

图6-16 说明类型检查的简单文法

1) **说明** 说明引起标识符的类型进入符号表。因此，文法规则

var-decl → **id** : *type-exp*

有相应的语义动作

insert (**id** .*name*, *type-exp.type*)

把标识符插入到符号表并关联一个类型。在这个插入中相关的类型根据 *type-exp* 的文法规则构造。

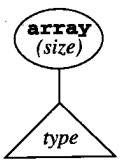
表6-10 用于图6-16 简单文法类型检查的属性文法

文法规则	语义规则
var-decl → **id** : *type-exp*	*insert*(**id**.*name*, *type-exp.type*)
type-exp → **int**	*type-exp.type* := *integer*
type-exp → **bool**	*type-exp.type* := *boolean*
$type\text{-}exp_1$ → **array** [**num**] **of** $type\text{-}exp_2$	$type\text{-}exp_1.type$:= *makeTypeNode* (*array*.**num**.*size*., $type\text{-}exp_2.type$)
stmt → **if** *exp* **then** *stmt*	**if not** *typeEqual*(*exp.type* , *boolean*) **then** *type-error*(*stmt*)
stmt → **id** := *exp*	**if not** *typeEqual*(*lookup*(**id**.*name*), *exp.type*) **then** *type-error*(*stmt*)
exp_1 → exp_2 + exp_3	**if not** (*typeEqual*($exp_2.type$, *integer*) **and** *typeEqual*($exp_3.type$, *integer*)) **then** *type-error*(exp_1) ; $exp_1.type$:= *integer*
exp_1 → exp_2 **or** exp_3	**if not** (*typeEqual*($exp_2.type$, *boolean*) **and** *typeEqual*($exp_3.type$, *boolean*) **then** *type-error*(exp_1); $exp_1.type$:= *boolean*
exp_1 → exp_2 [exp_3]	**if** *isArrayType*($exp_2.type$) **and** *typeEqual*($exp_3.type$, *integer*) **then** $exp_1.type$:= $exp_2.type.child1$ **else** *type-error*(exp_1)
exp → **num**	*exp.type* := *integer*
exp → **true**	*exp.type* := *boolean*
exp → **false**	*exp.type* := *boolean*
exp → **id**	*exp.type* := *lookup*(**id** .*name*)

假定类型保持某种树形结构，因此在图6-16中的文法的一种结构类型**array**对应于语义动作

makeTypeNode (*array.size.type*)

构成一个类型节点



这里数组节点的子孙是*type*参数给定的类型树。在树的表示中假定简单类型 *integer*和*boolean*构成了标准叶子节点。

2) **语句** 语句本身没有类型，但对类型正确性而言需要检查子结构。一般的情形是在示例文法中两个语句规则， if语句和赋值语句。在 if语句的情况中，条件表达式必须是布尔类型。这通过规则

if not *typeEqual* (*exp.type* , *boolean*) **then** *type-error*(*stmt*)

表示，这里*type-error*指示一个错误报告机制，其属性将简要地描述。

在赋值语句的情况下，要求被赋值的变量和其接受的值的表达式有相同的类型。这依赖于*typeEqual*函数表示的类型等价算法。

3) **表达式** 常量表达式，像数字及布尔值`true`和`false`，隐含地说明了*integer*和*boolean*类型。变量名在符号表中通过*lookup*操作确定它们的类型。其他表达式通过操作符构成，如算术操作符`+`、布尔操作符`or`、以及下标操作符`[]`。对每种情况子表达式都必须是指定操作的正确类型。对于下标的情况，这由规则

if *isArrayType*(exp_2.*type*)
and *typeEqual*(exp_3.*type* , *integer*)
then exp_1.*type* := exp_2.*type.child1* **else** *type-error*(exp_1)

指示这里函数*isArrayType*测试其参数是数组类型，即类型的树形表示有一个根节点，表示数组类型构造器。下标表达式导出的类型是数组的基类型，在数组类型的树形表示中它是根节点的(第一个)子节点表示的类型，这通过exp_2.*type.child1*指示。

现在留下了描述在出现错误时这样的类型检查器的行为，像表 6-10中语义规则的*type-error*过程所指示的那样。主要的问题是何时产生错误消息以及在错误出现时如何继续类型检查。每次出现类型错误时不是都产生错误消息；另一方面，单个的错误可能会引起一连串的许多错误(有时也恰好出现语法错误)。事实上，如果*type-error*过程能确定在有关的位置已经出现了类型错误，那么就可能抑制错误消息的产生。这可以通过一个特别的内部错误类型 (用一空的类型树表示)发信号。如果在一个子结构中 *type-error*遇到这个错误类型，就没有错误消息产生。同时，如果错误类型意味着结构的类型不能被确定，那么类型检查器可以当作它的类型 (实际上是未知的)使用错误类型。例如，在表 6-10的语义规则中，给定一个下标表达式 $exp_1 \rightarrow exp_2$ `[`exp_3`]`，如果exp_2 不是数组类型，那么exp_1 不能被赋予一个有效的类型，并且在语义动作中没有类型赋值。这假定类型域被某个错误类型初始化。另一方面，在操作符 `+`和`or`的情况，即使出现类型错误，这个假设可以使结果意味着整型或布尔型，表 6-10中的规则也使用它们给结果分配一个类型。

6.4.5 类型检查的其他主题

在这一小节我们简要讨论前面已经讨论过的类型检查算法的一些常用的扩展。

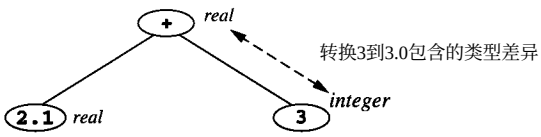
1) **重载** 一个操作符是重载的，如果同一操作符名用于了两个不同的操作。重载常用的例

子是算术操作符的情况，通常表示不同数值的操作。例如，**2+3**表示整数加，而**2.1+3.0**表示浮点数加，必须通过不同的指令或指令集在内部实现。这样的重载可以扩展到用户说明的函数或过程，相关的操作使用相同的名字，但说明不同的参数或不同的类型。例如，对两个整数和实数值，我们定义取最大值的过程：

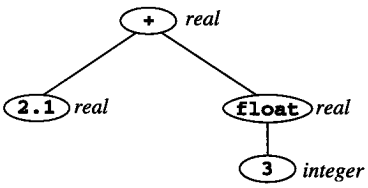
```
procedure max (x,y: integer): integer;
procedure max (x,y: real): real;
```

在Pascal和C中这是非法的，因为它表示了在相同的作用域中相同名字的重说明。然而，在Ada和C++中，这样的说明是合法的，因为类型检查器可以根据参数的类型确定要使用哪一个**max**过程。使用这样的类型检查分清名字的多种含义，可以根据语言的规则用多种方法实现。一种方法用类型参数增加了符号表的*lookup*过程，允许符号表找到正确的匹配。另一种不同的解决方法是对符号表保持名字所有可能类型的一个集合，并把这个集合返回给类型检查器。这对更复杂的情形是有用的，唯一的类型可以不必立即确定。

2) **类型转换和强制**　语言类型规则的一种常用扩充是允许混合类型的算术表达式，如**2.1+3**，一个实数和一个整数相加。在这样的情况下，必须建立一种通用的类型与所有子表达式的类型兼容，在应用操作符之前必须用某种操作把运行的值转换到相应的表示。例如，在表达式**2.1+3**中，在进行加之前整数值**3**必须转换成浮点数，结果表达式将是浮点数类型。语言进行这样的转换有两种途径。例如，Modula-2要求编程者提供一个转换函数，这样刚才给出的例子可以写成**2.1+FLOAT(3)**，否则将导致类型错误。另一种可能性(在C语言中使用)是基于子表达式的类型，为类型检查器提供一个自动的转换操作。这样的一个自动转换称作强制(coercion)。强制能由类型检查器隐含地表示，根据子表达式的类型推断出表达式的类型。如



这要求后面的代码产生器检查表达式的类型确定是否需要应用转换。另一种情况，通过在语法树中插入一个转换节点，类型检查器可以隐含地提供转换，如



类型转换和强制也用于赋值，如

```
r = i;
```

在C语言中，如果**r**的类型是**double**，**i**的类型是**int**，在存储**i**为**r**的值之前，它的值强制为**double**。这样的赋值在转换期间可能会丢失信息，就像相反方向的赋值(在C中也是合法的)：

```
i = r;
```

类似的情形出现在面向对象的语言中，通常允许子类对象向超类对象赋值（信息也有相应的丢失)。例如，在C++中，如果是**A**一个类，是**B**一个子类，并且如果**x**是的**A**对象，**y**是**B**的对象，那么**x=y**是允许的，但反过来不行(这称作子类型原理(subtype principle))。

3) **多态性类型** 有一种语言是多态性(polymorphic)的，如果允许语言的构造有多种类型。直到现在我们讨论的语言的类型检查本质上是单态 (monomorphic)的，所有的名字和表达式都要求有唯一的类型。对这种单态性要求的一种放松是重载。但是，重载通过多态性的一种形式，只能用于相同名字多种独立说明的情形。当单个说明需要用于任意的类型时就出现了另一种不同的情形。例如，交换两个变量值的过程在原理上可以用于任何类型的变量 (只要它们类型相同)：

```
procedure swap (var x,y: anytype);
```

这个**swap**过程的类型称作被类型***anytype***限定(parametrized)，***anytype***被看作是类型变量(type variable)，能假设成任意实际的类型。可以这样表达这个过程的类型

procedure (*var anytype*, *var anytype*): *void*

这里*anytype*的每次出现都引用相同的(但是未指定的)类型。这样的类型实际上是类型模式(type pattern)或类型方案(type scheme)而不是实际的类型，类型检查器对每次使用*swap*的情形都需要确定实际的类型，匹配这个类型模式或说明一个类型错误。例如，给定代码

```
var x,y: integer;
    a,b: char;
. . .
swap(x,y);
swap(a,b);
swap(a,x);
```

在调用**swap(x,y)**时，**swap**过程根据其给定的多态性类型模式“指定”到(单态)类型

procedure (*var integer*, *var integer*): *void*

而在调用**swap(a,b)**时，它被指定类型

procedure (*var char*, *var char*): *void*

另一方面，在调用**swap(a,x)**时，**swap**过程的类型为

procedure (*var char*, *var integer*): *void*

并且这个类型不能从**swap**的类型模式通过代替类型变量*anytype*来产生。存在类型检查算法进行这种一般的多态性类型检查，特别是在 ML这样的现代的函数式语言中，但其中包括复杂的模式匹配技术，这里不进行研究(参见“注意与参考”一节)。

6.5 TINY语言的语义分析

这一节我们基于前一章构造的 TINY语法分析程序，开发 TINY语言的语义分析程序代码。语义分析程序所基于的TINY的语法和语法树结构在3.7节描述。

TINY语言在其静态语义要求方面特别简单，语义分析程序也将反映这种简单性。在 TINY中没有明确的说明，也没有命名的常量、数据类型或过程；名字只引用变量。变量在使用时隐含地说明，所有的变量都是整数数据类型。也没有嵌套作用域，因此变量名在整个程序有相同的含义，符号表也不需要保存任何作用域信息。

在TINY中类型检查也特别简单。只有两种简单类型：整型和布尔型。仅有的布尔型值是两个整数值的比较的结果。因为没有布尔型操作符或变量，布尔值只出现在 if或repeat语句的测试表达式中，不能作为操作符的操作数或赋值的值。最后，布尔值不能使用 *write*语句输出。

我们把对TINY语义分析程序的代码的讨论分成两个部分。首先，讨论符号表的结构及其

相关的操作。然后，语义分析程序自身的操作，包括符号表的构造和类型检查。

6.5.1 TINY的符号表

在TINY语义分析程序符号表的设计中，首先确定什么信息需要在符号表中保存。一般情况这些信息包括数据类型和作用域信息。因为 TINY没有作用域信息，并且所有的变量都是整型，TINY符号表不需要保存这些信息。然而，在代码产生期间，变量需要分配存储器地址，并且因为在语法树中没有说明，因此符号表是存储这些地址的逻辑位置。现在，地址可以仅仅看成是整数索引，每次遇到一个新的变量时增加。为使符号表更加有趣和有用，还使用符号表产生一个交叉参考列表，显示被访问变量的行号。

作为符号表产生信息的例子，考虑下列TINY程序的例子(加上了行号)：

```
 1: { Sample program
 2:   in TINY language --
 3:   computes factorial
 4: }
 5: read x; { input an integer }
 6: if 0 < x then { don't compute if x <= 0 }
 7:   fact := 1;
 8:   repeat
 9:     fact := fact * x;
10:     x := x - 1
11:   until x = 0;
12:   write fact { output factorial of x }
13: end
```

这个程序的符号表产生之后，语义分析程序将输出 (**TraceAnalyze= True**)下列信息到列出的文件中：

```
Symbol table:
Variable Name   Location   Line  Numbers
----------------------------------------
x               0          5     6      9     10     10     11
fact            1          7     9      9     12
```

注意，在符号表中同一行的多次引用产生了那一行的多个入口。

符号表的代码包含在**symtab.h**和**symtab.c**文件中，在附录B中列出(分别是第1150到1179行和第1200到1321行)。

符号表使用的结构是在6.3.1节中描述的分离的链式杂凑表，杂凑函数是程序清单 6-2给出的。因为没有作用域信息，所以不需要 *delete*操作，*insert*操作除了标识符之外，也只需要行号和地址参数。需要的其他的两个操作是打印刚才列出的文件中的汇总信息，以及 *lookup*操作，从符号表中取出地址号(后面的代码产生器需要，符号表生成器也要检查是否已经看见了变量)。因此，头文件**symtab.h**包含下列说明：

```
void st_insert ( char * name, int lineno, int loc );
int st_lookup ( char * name );
void printSymTab(FILE * listing);
```

因为只有一个符号表，它的结构不需要在头文件中说明，也无须作为参数在这些过程中出现。

在**symtab.c**中相关的实现代码使用了一个动态分配链表，类型名是**LineList**(第1236行到第1239行)，存储记录在杂凑表中每个标识符记录的相关行号。标识符记录本身保存在一个

“桶”列表中，类型名是**BucketList**(第1247行到第1252行)。**st_insert**过程在每个“桶”列表(第1262行到第1295行)前面增加新的标识符记录，但行号在每个行号列表的尾部增加，以保持行号的顺序(**st_insert**的效率可以通过使用环形列表或行号列表的前/后双向指针来改进；参见练习)。

6.5.2 TINY语义分析程序

TINY的静态语义共享标准编程语言的特性，符号表的继承属性，而表达式的数据类型是合成属性。因此，符号表可以通过对语法树的前序遍历建立，类型检查通过后序遍历完成。虽然这两个遍历能容易地组合成一个遍历，为使两个处理步骤操作的不同之处更加清楚，仍把它们分成语法树上两个独立的遍。因此，语义分析程序与编译器其他部分的接口，放在文件**analyze.h**中(附录B，第1350行到第1370行)，由两个过程组成，通过下列说明给出

```
void buildSymtab(TreeNode *);
void typeCheck(TreeNode *);
```

第1个过程完成语法树的前序遍历，当它遇到树中的变量标识符时，调用符号表**st_insert**过程。遍历完成后，它调用**printSymTab**打印列表文件中存储的信息。第2个过程完成语法树的后序遍历，在计算数据类型时把它们插入到树节点，并把任意的类型检查错误记录到列表文件中。这些过程及其辅助过程的代码包含在**analyze.c**文件中（附录B，第1400行到第1558行）。

为强调标准的树遍历技术，实现**buildSymtab**和**typeCheck**使用了相同的通用遍历函数**traverse**(第1420行到第1441行),它接受两个作为参数的过程(和语法树)，一个完成每个节点的前序处理，一个进行后序处理：

```
static void traverse ( TreeNode * t,
               void (* preProc) (TreeNode * ),
               void (* postProc) (TreeNode * ) )
{ if (t != NULL)
  { preProc(t);
    { int i;
      for (i=0; i < MAXCHILDREN; i++)
        traverse(t->child[i],preProc, postProc);
    }
    postProc(t);
    traverse(t->sibling, preProc, postProc);
  }
}
```

给定这个过程，为得到一次前序遍历，当传递一个“什么都不做”的过程作为**preproc**时，需要说明一个过程提供前序处理并把它作为**preproc**传递到**traverse**。对于TINY符号表的情况，前序处理器称作**insertNode**，因为它完成插入到符号表的操作。“什么都不做”的过程称作**nullProc**，它用一个空的过程体说明(第1438行到第1441行)。然后建立符号表的前序遍历由**buildSymtab**过程(第1488行到第1494行)内的单个调用

```
traverse (syntaxTree, insertNode, nullProc);
```

完成。类似地，**typeCheck**(第1556行到第1558行)要求的后序遍历由单个调用

```
traverse (syntaxTree, nullProc, checkNode);
```

完成。这里**checkNode**是一个适当说明的过程，计算和检查每个节点的类型。现在还剩下描述过程**insertNode**和**checkNode**的操作。

insertNode过程(第1447行到第1483行)必须基于它通过参数 (指向语法树节点的指针)接受的语法树节点的种类，确定何时把一个标识符 (与行号和地址一起)插入到符号表中。对于语句节点的情况，包含变量引用的节点是赋值节点和读节点，被赋值或读出的变量名包含在节点的**attr.name**字段中。对表达式节点的情况，感兴趣的是标识符节点，名字也存储在**attr.name**中。因此，在那 3个位置，如果还没有看见变量 **insertNode**过程包含一个

```
st_insert (t->attr.name, t->lineno, location++);
```

调用(与行号一起存储和增加地址计数器)，并且如果变量已经在符号表中，则

```
st_insert (t->attr.name,t->lineno,0);
```

(存储行号但没有地址)。

最后，在符号表建立之后， **buildSymtab**完成对**printSymTab**的调用，在标志**TraceAnalyze**的控制下(在**main.c**中设置)，在列表文件中写入行号信息。

类型检查遍的**checkNode**过程有两个任务。首先，基于子节点的类型，它必须确定是否出现了类型错误。其次，它必须为当前节点推断一个类型(如果它有一个类型)并且在树节点中为这个类型分配一个新的字段。这个字段在**TreeNode**中称作**type**字段(在**globals.h**中说明，见附录B，第216行)。因为仅有表达式节点有类型，这个类型推断只出现在表达式节点。在TINY中只有两种类型，整型和布尔型，这些类型在全局说明的枚举类型中说明(见附录B，第203行)：

```
typedef enum {Void, Integer, Boolean} ExpType;
```

这里类型 **Void**是“ 无类型 ”类型，仅用于初始化和错误检查。当出现一个错误时，**checkNode**过程调用**typeError**过程，基于当前的节点，在列表文件中打印一条错误消息。

还剩下归类**checkNode**的动作。对表达式节点，节点可以是叶子节点 (常量或标识符，种类是**ConstK**或**IdK**)，或者是 操作符节点(种类**OpK**)。对叶子节点的情况(第1517行到第1520行)，类型总是**Integer** (没有类型检查发生)。对操作符节点的情况(第1508行到第1516行)，两个子孙子表达式的类型必须是**Integer** (因为后序遍历已经完成，已经计算出它们的类型)。然后，**OpK**节点的类型从操作符本身确定 (不关心是否出现了类型错误)：如果操作符是一个比较操作符(<或=)，那么类型是**Boolean**；否则是**Integer**。

对语句节点的情况，没有类型推断，但除了一种情况，必须完成某些类型检查。这种情况是**ReadK**语句，这里被读出的变量必须自动成为**Integer**类型，因此没有必要进行类型检查。所有4种其他语句种类需要一些形式的类型检查：**IfK**和**RepeatK**语句需要检查它们的测试表达式，确保它们是类型**Boolean**(第1527行到第1530行和第1539行到第1542行)，而**WriteK**和**AssignK**语句需要检查(第1531行到第1538行)确定被写入或赋值的表达式不是布尔型的 (因为变量只能是整型值，只有整型值能被写入)：

```
x := 1 < 2; { error - Boolean value
                  cannot be assigned }
write 1 = 2; { also an error }
```

练习

6.1 通过下面文法给出的数的整数值，写出一个属性文法：

number → *digit number* | *digit*
digit → **0|1|2|3|4|5|6|7|8|9**

6.2 通过下面文法给出的十进制数的浮点数值，写出一个属性文法 (提示：使用一个属性*count*计算小数点右面数字的个数)。

dnum → *num*.*num*
num → *num digit* | *digit*
digit → **0|1|2|3|4|5|6|7|8|9**

6.3 前一个练习中的十进制数文法可进行重写，不需要属性 *count* (包括它的等式也可避免)。重写文法实现这一点，并为*dnum*的值给出一个新的属性文法。

6.4 考虑一个表达式文法，它可写成消除左递归的预分析程序：

exp → *term exp′*
exp′ → **+** *term exp′* | **-** *term exp′* | ε
term → *factor term′*
term′ → ***** *factor term′* | ε
factor → **(***exp***)** | ***number***

写出用这个文法给出的表达式的值的属性文法。

6.5 重写表6-2的属性文法，代替*val*计算*postfix*串属性，包含简单整数表达式的后缀形式。例如，**(34-3)*42**的*postfix*属性是“ 34 3 **-** 42 + * ”。可以假设一个串联操作符 ||和***number***.strval属性存在。

6.6 考虑下面的整数二叉树文法(线性形式)：

btree →(***number*** *btree btree*)| **nil**

写出一个属性文法检查二叉树是有序的，即第一个子树中数的值≤当前数的值，并且第2个子树所有数的值≥当前数的值。例如，**(2 (1 nil nil) (3 nil nil))**是有序的，而**(1 (2 nil nil) (3 nil nil))**不是。

6.7 考虑下面简单的类Pascal 说明的文法：

decl → *var-list* **:** *type*
var-list → *var-list* **,** ***id*** | ***id***
type → **integer** | **real**

写出变量类型的一个属性文法。

6.8 考虑练习**6.7**的文法。重写这个文法使变量的类型可以说明为纯的合成属性，并给出具有这个特性的类型的新的属性文法。

6.9 重写例6.4的文法和属性文法，使*based-num* 的值能通过单独的合成属性计算。

6.10 **a.** 画出对应于例6.14中每个文法规则的相关图，表达式是**5/2/2.0**。

b. 描述要求在**5/2/2.0**的语法树上计算属性的两遍，包括节点访问可能的顺序和在每点计算的属性值。

c. 写出过程的伪代码，完成**b**中描述的计算。

6.11 画出对应于练习**6.4**中属性文法的每个文法规则的相关图，字符串是**3*(4+5)*6**。

6.12 画出对应于练习**6.7**中属性文法的每个文法规则的相关图，画出说明 **x,y,z:real**

的相关图。

6.13 考虑下面的属性文法：

文法规则	语义规则
$S \rightarrow A\ B\ C$	$B.u = S.u$
	$A.u = B.v + C.v$
	$S.v = A.v$
$A \rightarrow a$	$A.v = 2 * A.u$
$B \rightarrow b$	$B.v = B.u$
$C \rightarrow c$	$C.v = 1$

a. 画出字符串*abc*的语法树(语言仅有的字符串)，画出相关属性的相关图。描述属性等式的正确顺序。

b. 假设在属性等式开始前$S.u$赋值为3。当等式完成时$S.v$的值的多少？

c. 假设属性等式修改如下：

文法规则	语义规则
$S \rightarrow A\ B\ C$	$B.u = S.u$
	$C.u = A.v$
	$A.u = B.v + C.v$
	$S.v = A.v$
$A \rightarrow a$	$A.v = 2 * A.u$
$B \rightarrow b$	$B.v = B.u$
$C \rightarrow c$	$C.v = C.u - 2$

如果等式开始前$S.u = 3$，属性等式完成后$S.v$的值是多少？

6.14 说明对如下给定的属性文法：

文法规则	语义规则
decl → *type var-list*	*vat-list.dtype* = *type.dtype*
type → **int**	*type.dtype* = *integer*
type → **float**	*type.dtype* = *real*
$var\text{-}list_1$ → **id,** $var\text{-}list_2$	**id**.*dtype* = $var\text{-}list_1$.*dtype*
	$var\text{-}list_2$.*dtype* = $var\text{-}list_1$.*dtype*
var-list → **id**	**id**.*dtype* = *var-list.dtype*

如果在LR分析期间属性*type.dtype*保存在值栈中，那么当发生*var-list* 归约时，这个值不能在栈中的固定位置找到。

6.15 **a.** 说明文法$B \rightarrow B\ b \mid a$ 是 SLR(1)，但文法

$$B \rightarrow A\ B\ b \mid a$$
$$A \rightarrow \varepsilon$$

(由前面文法加上ε产生式构造)，对任意k 都不是LR(k)。

b. 给定(a)部分的文法(带ε-产生式)，Yacc产生的分析程序接受什么字符串？

c. 这种情形与“实际的”编程语言语义分析期间出现的情况是否相似？试说明。

6.16 把6.3.5节的表达式文法重写成无二义性文法，用这样的方法那一节写的表达式保持合法性，用这个新文法重写表6-9的属性文法。

6.17 使用并列说明代替顺序说明重写表6-9的属性文法。

6.18 写一个属性文法，计算6.3.5节中表达式文法的每个表达式的值。

6.19 修改程序清单6-6函数*typeEqual*的伪代码，合并类型名并提出确定252页描述的递归类型的结构等价的算法。

6.20 考虑下列表达式的(二义)文法：

$$exp \rightarrow exp\ \texttt{+}\ exp \mid exp\ \texttt{-}\ exp \mid exp\ \texttt{*}\ exp \mid exp\ /\ exp$$
$$\mid \texttt{(}exp\texttt{)} \mid \textbf{\textit{num}} \mid \textbf{\textit{num}}\texttt{.}\textbf{\textit{num}}$$

假设在计算任何这样的表达式时遵循C语言的规则：如果两个表达式是混合类型的，那么整型的子表达式转换成浮点型，并应用浮点型操作符。写一个属性文法把这样的表达式转换成在Modula-2中也是合法的表达式：从整数到浮点数的转换使用**FLOAT**函数表达，如果两个操作数都是整数，除法操作符**/**就被视为**div**。

6.21 考虑对图6-16的下列文法的扩展，它包括了函数说明和调用：

program → *var-decls* **;** *fun-decls* **;** *stmts*
var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → ***id*** **:** *type-exp*
type-exp → **int|bool|array [num] of** *type-exp*
fun-decls → **fun id (** *var-decls* **):** *type-exp* **;** *body*
body → *exp*
stmts → *stmts* **;** *stmt* | *stmt*
stmt → **if** *exp* **then** *stmt* | **id :=** *exp*
exp → *exp* **+** *exp* | *exp* **or** *exp* | *exp* **[***exp***]** | **id (** *exps* **)**
|num|true|false|id
exps → *exps* **,** *exp* | *exp*

a. 为新的函数类型结构设计一个合适的树结构，为两个函数类型写一个*typeEqual*函数。

b. 写出函数说明和函数调用类型检查的语义规则(由规则*exp* → ***id*** **(***exps***)** 表示)，类似于表6-10的规则。

6.22 考虑下列C表达式的二义文法。给定表达式

```
(A)-x
```

如果**x**是一个整型变量，**A**在**typedef**中说明等价于**double**，那么这个表达式计算**-x**的值为**double**类型。另一方面，如果**A**是一个整型变量，则计算两个变量的整型差值。

a. 描述分析程序如何使用符号表区分这两种解释。

b. 描述扫描器如何使用符号表区分这两种解释。

6.23 对应于TINY类型检查器的强制类型约束写一个属性文法。

6.24 写出一个TINY语义分析程序符号表构造的属性文法。

编程练习

6.25 写出例**6.4**基数语法树的C语言说明，并使用这些说明把例6.13的*EvalWithBase*伪代码转换成C代码。

6.26 **a.** 重写程序清单4-1的递归下降求值程序，代替表达式的值而打印出后缀转换式（见练习**6.5**)。

b. 重写递归下降求值程序，打印出值和后缀转换式。

6.27 **a.** 为一个简单整数计算重写程序清单5-1的Yacc规范，代替表达式的值而打印出后缀转换式(见练习**6.5**)。

b. 重写Yacc规范，打印出值和后缀转换式。

6.28 写出一个Yacc规范，打印出练习**6.20**中文法给出的表达式的Modula-2转换式。

6.29 写出一个程序的Yacc规范，用于计算带let-块的表达式的值(表6-9) (可以把**`let`**和**`in`**记号缩写成一个字符，并约束标识符或使用Lex产生一个合适的扫描器)。

6.30 写出一个程序的Yacc和Lex规范，进行语言的类型检查，其文法在图6-16给出。

6.31 重写TINY语义分析程序符号表的实现，数据结构**`LineList`**加进一个向后的指针，并提高**`insert`**操作的效率。

6.32 重写TINY语义分析程序，使其只对语法树进行一遍遍历。

6.33 TINY语义分析程序在变量使用之前，没有确保其已被赋值。因此，下面的TINY代码在语义上认为是正确的：

```
y := 2+x;
x := 3;
```

重写TINY分析程序进行“合理的”检查，在表达式中一个变量的赋值发生在使用之前。什么妨碍这样的检查十分简单？

6.34 **a.** 重写TINY语义分析程序，允许布尔值存储到变量中。这将要求在符号表中给定变量的数据类型为布尔型或整型。TINY程序的类型正确性现在必须包括对变量所有的赋值(和使用)与它们的数据类型一致的要求。

b. 根据**a**中的修改，写出一个TINY类型检查器的属性文法。

注意与参考

属性文法的早期工作主要是Knuth[1968]进行的。在编译器构造中使用属性文法的进一步研究出现在Lorho[1984]。正式使用属性文法指定编程语言的语义是Slonneger和Kurtz[1995]的研究，这里为类似于TINY的语言的静态语义给出了一个完整的属性文法。属性文法的其他数学特性可以在Mayoh[1981]中找到。一种非闭环的测试可以在Jazayeri、Ogden和Rounds[1975]中找到。

分析期间属性的赋值问题在Fischer和LeBlanc[1991]的研究中更加详细一些。在LR分析期间确保其进行的条件在Jones[1980]中。在调度动作中加进ε-产生式(像在Yacc中)保持确定性的LR分析的问题在Purdom和Brown中研究。

符号表实现的数据结构，包括杂凑表及其效率分析，能在许多文章中找到；例如参见Aho、

Hopcroft和Ullman[1983]或Cormen、Leiserson和Rivest[1990]。选择杂凑函数的细致的研究在Knuth[1973]中。

类型系统、类型正确性和类型推断形成了理论计算机科学研究的一个主要的领域，并应用到许多语言中。通常的概要参见Louden[1993]。更进一步的观点参见Cardelli和Wegner[1985]。C和Ada使用类型等价的混合形式，类似于Pascal的等价说明，很难简洁地描述。较早的语言，如FORTRAN77、Algol60和Algol68使用结构等价。像ML和Haskell使用严格的名等价，用类型同义词代替结构等价。在6.4.3节中描述的结构等价算法可以在Koster[1969]中找到；类似算法的现代的应用在Amadio和Cardelli[1993]中。多态的类型系统和类型推导算法在Peyton Jones[1987]和Reade[1989]中描述。在ML和Haskell中使用的多态的类型推导称作**Hindley-Milner**类型推导[Hindley，1969；Milner，1978]。

本章中我们没有描述任何属性求值的自动构造工具，因为通常并不使用(不像扫描器和分析程序产生器Lex和Yacc)。基于属性文法的一些有趣的工具是LINGUIST[Farrow，1984]和GAG[Kastens，Hutt和Zimmermann，1982]。合成器产生器[Reps和Teitelbaum，1989]是一个成熟的工具，用于基于属性文法的上下文有关编辑器的产生。对这个工具可做的一件有趣的事是构造一个语言编辑器，自动地提供基于使用的变量声明。