第6章　中间代码生成

在编译器的分析－综合模型中，前端对源程序进行分析并产生中间表示，后端在此基础上生成目标代码。理想情况下，和源语言相关的细节在前端分析中处理，而关于目标机器的细节则在后端处理。基于一个适当定义的中间表示形式，可以把针对源语言 i 的前端和针对目标机器 j 的后端组合起来，构造得到源语言 i 在目标机器 j 上的一个编译器。这种创建编译器组合的方法可以大大减少工作量：只要写出 m 种前端和 n 种后端处理程序，就可以得到 $m \times n$ 种编译程序。

本章的内容涉及中间代码表示、静态类型检查和中间代码生成。为简单起见，我们假设一个编译程序的前端处理按照图 6-1 所示方式进行组织，顺序地进行语法分析、静态检查和中间代码生成。有时候这几个过程也可以组合起来，在语法分析中一并完成。我们将使用第 2 章和第 5 章中的语法制导定义来描述类型检查和翻译过程。大部分的翻译方案可以基于第 5 章中给出的自顶向下或自底向上的语法分析技术来实现。所有的方案都可以通过生成并遍历抽象语法树来实现。

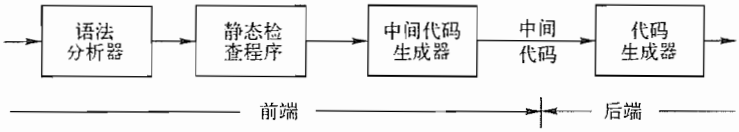


图 6-1　一个编译器前端的逻辑结构

静态检查包括类型检查(type checking)，类型检查保证运算符被应用到兼容的运算分量。静态检查还包括在语法分析之后进行的所有语法检查。例如，静态检查保证了 C 语言中的一条 break 指令必然位于一个 while/for/switch 语句之内。如果不存在这样的语句，静态检查将报告一个错误。

本章介绍的方法可以用于多种中间表示，包括抽象语法树和三地址代码。这两种中间表示方法都在2.8 节中介绍过。之所以名为“三地址代码”，是因为这些指令的一般形式 $x = y\ op\ z$ 具有三个地址：两个运算分量 y 和 z，一个结果变量 x。

在将给定源语言的一个程序翻译成特定的目标机器代码的过程中，一个编译器可能构造出一系列中间表示，如图 6-2 所示。高层的表示接近于源语言，而低层的表示接近于目标机器。语法树是高层的表示，它刻画了源程序的自然的层次性结构，并且适用于静态类型检查这样的处理。

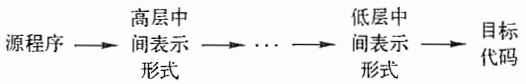


图 6-2　编译器可能使用一系列的中间表示

低层的表示形式适用于机器相关的处理任务，比如寄存器分配、指令选择等。通过选择不同的运算符，三地址代码既可以是高层的表示方式，也可以是低层的表示方式。在 6.2.3 节将看到，对表达式而言，语法树和三地址代码只是在表面上有所不同。对于循环语句，语法树表示了语句的各个组成部分，而三地址代码包含标号和跳转指令，用来表示目标语言的控制流。

不同的编译器对中间表示的选择和设计各有不同。中间表示可以是一种真正的语言，也可以由编译器的各个处理阶段共享的多个内部数据结构组成。C 语言是一种程序设计语言。它具有很好的灵活性和通用性，可以很方便地把 C 程序编译成高效的机器代码，并且有很多 C 的编译器可用，因此 C 语言也常常被用作中间表示。早期的 C++ 编译器的前端生成 C 代码，而把 C 编译器作为其后端。

6.1 语法树的变体

语法树中的各个结点代表了源程序中的构造，一个结点的所有子结点反映了该结点对应构造的有意义的组成成分。为表达式构建的无环有向图(Directed Acyclic Graph，以后简称 DAG)指出了表达式中的*公共子表达式*(多次出现的子表达式)。在本节我们将看到，可以用构造语法树的技术去构造 DAG。

6.1.1 表达式的有向无环图

和表达式的语法树类似，一个 DAG 的叶子结点对应于原子运算分量，而内部结点对应于运算符。与语法树不同的是，如果 DAG 中的一个结点 N 表示一个公共子表达式，则 N 可能有多个父结点。在语法树中，公共子表达式每出现一次，代表该公共子表达式的子树就会被复制一次。因此，DAG 不仅更简洁地表示了表达式，而且可以为最终生成表达式的高效代码提供重要的信息。

例 6.1 图 6-3 给出了下面的表达式的 DAG

```
a + a * (b - c) + (b - c) * d
```

叶子结点 a 在表达式中出现了两次，因此 a 有两个父结点。值得注意的是，结点“-”代表公共子表达式 b-c 的两次出现。该结点同样有两个父结点，表明该子表达式在子表达式 a*(b-c)和(b-c)*d 中两次被使用。尽管 b 和 c 在整个表达式中出现了两次，但它们对应的结点只有一个父结点，因为对它们的使用都出现在同样的公共子表达式 b-c 中。 □

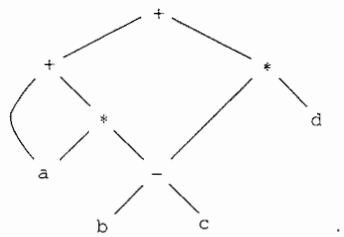


图 6-3 表达式 a+a*(b-c)+(b-c)*d 的 DAG

图 6-4 给出的 SDD(语法制导定义)既可以用来构造语法树，也可以用来构造 DAG。它在例 5.11 中曾用于构造语法树。在那里，函数 *Leaf* 和 *Node* 每次被调用都会构造出一个新结点。要构造得到 DAG，这些函数就要在每次构造新结点之前首先检查是否已存在这样的结点。如果存在一个已被创建的结点，就返回这个已有的结点。例如，在构造一个新结点 *Node*(*op*, *left*, *right*)之前，我们首先检查是否已存在一个结点，该结点的标号为 *op*，且其两个子结点为 *left* 和 *right*。如果存在这样的结点，*Node* 函数返回这个已存在的结点，否则它创建一个新结点。

	产生式	语义规则
1)	$E \to E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
	$T \to T_1 * F$	$T.node =$ **new** $Node('*', T_1.node, F.node)$
4)	$T \to (\ E\ )$	$T.node = E.node$
5)	$T \to$ **id**	$T.node =$ **new** $Leaf($**id**, **id**.$entry)$
6)	$T \to$ **num**	$T.node =$ **new** $Leaf($**num**, **num**.$val)$

图 6-4 生成语法树或 DAG 的语法制导定义

例 6.2 图 6-5 给出了构造图 6-3 所示 DAG 的各个步骤。如上所述，函数 *Node* 和 *Leaf* 尽可能地返回已存在的结点。我们假设 *entry-a* 指向符号表中与 a 对应的项，其他标识符的处理方式与此类似。

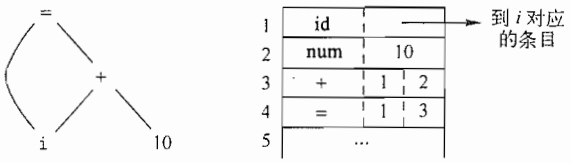
当在第 2 步再次调用 *Leaf*(**id**, *entry-a*)时，函数返回的是之前调用生成的结点，因此 $p_2 = p_1$。类似地，第 8 步和第 9 步返回的结点分别和第 3 步及第 4 步返回的结果相同(即 $p_8 = p_3$，$p_9 = p_4$)。同样，第 10 步返回的结点必然和第 5 步中返回的结点相同，即 $p_{10} = p_5$。 □

1) $p_1 = Leaf(\mathbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\mathbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\mathbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\mathbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\mathbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\mathbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\mathbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

图 6-5 图 6-3 所示的 DAG 的构造过程

6.1.2 构造 DAG 的值编码方法

语法树或 DAG 图中的结点通常存放在一个记录数组中，如图 6-6 所示。数组的每一行表示一个记录，也就是一个结点。在每个记录中，第一个字段是一个运算符代码，也是该结点的标号。在图 6-6b 中，各个叶子结点还有一个附加的字段，它存放了标识符的词法值(在这里，它是一个指向符号表的指针或一个常量)。内部结点则有两个附加的字段，分别指明其左右子结点。



a) DAG b) 数组

图 6-6 $i = i + 10$ 的 DAG 的结点在数组中的表示

在这个数组中，我们只需要给出一个结点对应的记录在此数组中的整数下标就可以引用该结点。在历史上，这个整数称为相应结点或该结点所表示的表达式的*值编码*(value number)。例如，在图 6-6 中，标号为“+”的结点的值编码为 3，其左右子结点的值编码分别为 1 和 2。在实践中，我们可以用记录指针或对象引用来代替整数下标，但是我们仍然把一个结点的引用称为该结点的“值编码”。如果使用适当的数据结构，值编码可以帮助我们高效地构造出表达式的 DAG。下一个算法将给出构造的方法。

假定结点按照如图 6-6 所示的方式存放在一个数组中，每个结点通过其值编码引用。设每个内部结点的范型为三元组 $<op, l, r>$，其中 op 是标号，l 是其左子结点对应的值编码，r 是其右子结点对应的值编码。假设单目运算符对应的结点有 $r = 0$。

算法 6.3 构造 DAG 的结点的值编码方法。

输入： 标号 op、结点 l 和结点 r。

输出： 数组中具有三元组 $<op, l, r>$ 形式的结点的值编码。

方法： 在数组中搜索标号为 op、左子结点为 l 且右子结点为 r 的结点 M。如果存在这样的结点，则返回 M 结点的值编码。若不存在这样的结点，则在数组中添加一个结点 N，其标号为 op，左右子结点分别为 l 和 r，返回新建结点对应的值编码。 □

虽然算法 6.3 可以产生我们期待的输出结果，但是每次定位一个结点时都要搜索整个数组，这个开销是很大的，当数组中存放了整个程序的所有表达式时尤其如此。更高效的方法是使用

散列表，将结点放入若干“桶”中，每个桶通常只包含少量结点。散列表是能够高效支持词典(dictionary)功能的少数几个数据结构之一[⊖]。词典是一种抽象的数据类型，它可以插入或删除一个集合中的元素，可以确定一个给定元素当前是否在集合中。类似于散列表这样为词典设计的优秀数据结构可以在常数或接近常数的时间内完成上述的操作，所需时间和集合的大小无关。

要给 DAG 中的结点构造散列表，首先需要建立*散列函数*(hash function)h。这个函数为形如 $<op, l, r>$ 的三元组计算“桶”的索引。它通过计算索引把三元组分配到各个桶中，并使得不大可能存在某个“桶”的元组数量大大超过平均数很多。通过对 op、l、r 的计算，可以确定地得到桶索引 $h(op, l, r)$。因而我们可以多次重复这个计算过程，总是得到结点 $<op, l, r>$ 的相同的桶索引。

桶可以通过链表来实现，如图 6-7 所示。一个由散列值索引的数组保存桶的*头*(bucket header)。每个头指向列表中的第一个单元。在一个桶的链表中，链表的各个单元记录了某个被散列函数分配到此桶中的某个结点的值编码。也就是说，在以数组的第 $h(op, l, r)$ 个元素为头的链表中可以找到结点 $<op, l, r>$。

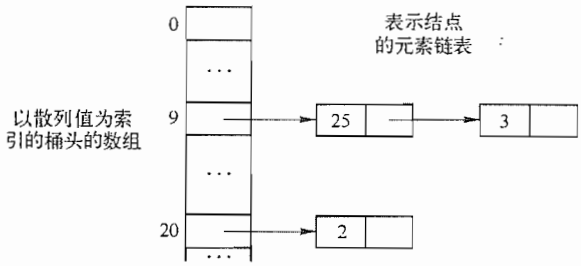


图 6-7 用于搜索桶的数据结构

因此，给定一个输入结点(op, l, r)，我们首先计算桶索引 $h(op, l, r)$，然后在该桶的单元中搜索这个结点。通常情况下有足够多的桶，因此链表中不会有很多单元。然而，我们必须查看一个桶中的所有单元，并且对于每一个单元中的值编码 v，我们必须检查输入结点的三元组 $<op, l, r>$ 是否和单元列表中值编码为 v 的结点相匹配(如图 6-7 所示)。如果我们找到了匹配的结点，就返回 v。如果没有找到匹配的结点，我们知道其他桶中也不会有这样的结点。因此，我们就创建一个新的单元，添加到“桶”索引为 $h(op, l, r)$ 的单元链表中，并返回新建结点对应的值编码。

6.1.3 6.1 节的练习

练习 6.1.1：为下面的表达式构造 DAG

$$((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$$

练习 6.1.2：为下列表达式构造 DAG，且指出它们的每个子表达式的值编码。假定 + 是左结合的。

1) $a+b+(a+b)$

2) $a+b+a+b$

3) $a+a+(a+a+a+(a+a+a+a))$

⊖ 参见 Aho, A. V. 、J. E. Hopcroft 和 J. D. Ullman 所著的《数据结构与算法》(Data Structures and Algorithms, Addison-Wesley 出版社 1983 年出版)。其中有关于支持词典功能的数据结构的讨论。

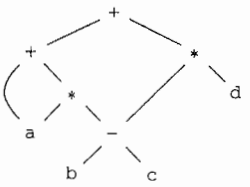
6.2 三地址代码

在三地址代码中，一条指令的右侧最多有一个运算符。也就是说，不允许出现组合的算术表达式。因此，像 x + y * z 这样的源语言表达式要被翻译成如下的三地址指令序列。

```
t1 = y * z
t2 = x + t1
```

其中 t_1 和 t_2 是编译器产生的临时名字。因为三地址代码拆分了多运算符算术表达式以及控制流语句的嵌套结构，所以适用于目标代码的生成和优化。具体的过程将在第 8、9 章中详细介绍。因为可以用名字来表示程序计算得到的中间结果，所以三地址代码可以方便地进行重组。

例 6.4 三地址代码是一棵语法树或一个 DAG 的线性表示形式。三地址代码中的名字对应于图中的内部结点。图 6-8 中再次给出了图 6-3 中的 DAG，以及该图对应的三地址代码序列。 □



a) DAG

```
t1 = b - c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4
```

b) 三地址代码

图 6-8 一个 DAG 及其对应的三地址代码

6.2.1 地址和指令

三地址代码基于两个基本概念：地址和指令。按照面向对象的说法，这两个概念对应于两个类，而各种类型的地址和指令对应于相应的子类。另一种方法是用记录的方式来实现三地址代码，记录中的字段用来保存地址。6.2.2 节将简要介绍被称为四元式和三元式的记录表示方式。

地址可以具有如下形式之一：

- 名字。为方便起见，我们允许源程序的名字作为三地址代码中的地址。在实现中，源程序名字被替换为指向符号表条目的指针。关于该名字的所有信息均存放在该条目中。
- 常量。在实践中，编译器往往要处理很多不同类型的常量和变量。6.5.2 节将考虑表达式中的类型转换问题。
- 编译器生成的临时变量。在每次需要临时变量时产生一个新名字是必要的，在优化编译器中更是如此。当为变量分配寄存器的时候，我们可以尽可能地合并这些临时变量。

下面我们介绍本书的其余部分常用的几种三地址指令。改变控制流的指令将使用符号化标号。每个符号化标号表示指令序列中的一条三地址指令的序号。通过一次扫描，或者通过回填技术就可以把符号化标号替换为实际的指令位置。回填技术将在 6.7 节中讨论。下面给出几种常见的三地址指令形式：

1）形如 $x = y\ op\ z$ 的赋值指令，其中 *op* 是一个双目算术符或逻辑运算符。x、y、z 是地址。

2）形如 $x = op\ y$ 的赋值指令，其中 *op* 是单目运算符。基本的单目运算符包括单目减、逻辑非和转换运算。将整数转换成浮点数的运算就是转换运算的一个例子。

3）形如 $x = y$ 的复制指令，它把 y 的值赋给 x。

4）无条件转移指令 *goto L*，下一步要执行的指令是带有标号 *L* 的三地址指令。

5）形如 `if` *x* `goto` *L* 或 `if False` *x* `goto` *L* 的条件转移指令。分别当 *x* 为真或为假时，这

两个指令的下一步将执行带有标号 L 的指令。否则下一步将照常执行序列中的后一条指令。

6）形如 `if` x *relop* y `goto` L 的条件转移指令。它对 x 和 y 应用一个关系运算符（<、==、>=等）。如果 x 和 y 之间满足 *relop* 关系，那么下一步将执行带有标号 L 的指令，否则将执行指令序列中跟在这个指令之后的指令。

7）过程调用和返回通过下列指令来实现：`param` x 进行参数传递，`call` p，n 和 y = `call` p，n 分别进行过程调用和函数调用；`return` y 是返回指令，其中 y 表示返回值，该指令是可选的。这些三地址指令的常见用法见下面的三地址指令序列

```
param x1
param x2
...
param xn
call p, n
```

它是过程 $p(x_1, x_2, \cdots, x_n)$ 的调用的一部分。“`call` p，n”中的 n 是实在参数的个数。这个 n 并不是冗余的，因为存在嵌套调用的情况。也就是说，前面的一些 `param` 语句可能是 p 返回之后才执行的某个函数调用的参数，而 p 的返回值又成为这个后续函数调用的另一个参数。过程调用的实现将在 6.9 节中加以介绍。

8）带下标的复制指令 $x = y[i]$ 和 $x[i] = y$。$x = y[i]$ 指令将把距离位置 y 处 i 个内存单元的位置中存放的值赋给 x。指令 $x[i] = y$ 将距离位置 x 处 i 个内存单元的位置中的内容设置为 y 的值。

9）形如 $x = \&y$、$x = *y$ 或 $*x = y$ 的地址及指针赋值指令。指令 $x = \&y$ 将 x 的右值设置为 y 的地址（左值）[⊖]。这个 y 通常是一个名字，也可能是一个临时变量。它表示一个诸如 `A[i][j]` 这样具有左值的表达式。x 是一个指针名字或临时变量。在指令 $x = *y$ 中，假定 y 是一个指针，或是一个其右值表示内存位置的临时变量。这个指令使得 x 的右值等于存储在这个位置中的值。最后，指令 $*x = y$ 则把 y 的右值赋给由 x 指向的目标的右值。

例 6.5 考虑语句

```
do i = i+1; while (a[i] < v);
```

图 6-9 给出了这个语句的两种可能的翻译。在图 6-9a 的翻译中，第一条指令上附加了一个符号化标号 L。图 6-9b 中的翻译显示了每条指令的位置号，我们在图中选择以 100 作为开始位置。在两种翻译中，最后一条指令都是目标为第一条指令的条件转移指令。乘法运算 `i * 8` 适用于每个元素占 8 个存储单元的数组。 □

```
L:   t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a [ t2 ]
     if t3 < v goto L
```

a) 符号标号

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

b) 位置号

图 6-9 给三地址指令指定标号的两种方法

选择使用哪些运算符是中间表示形式设计的一个重要问题。显然，这个运算符集合中的运算符要足够丰富，以便实现源语言中的所有运算。接近机器指令的运算符可以使在目标机器上实现中间表示形式更加容易。然而，如果前端必须为某些源语言运算生成很长的指令序列，那么优

⊖ 2.8.3 节曾经提出，左值和右值分别表示赋值左/右部。

化器和代码生成器就需要花费更多的时间去重新发现程序的结构，然后才能为这些运算生成高质量的目标代码。

6.2.2 四元式表示

上面对三地址指令的描述详细说明了各类指令的组成部分，但是并没有描述这些指令在某个数据结构中的表示方法。在编译器中，这些指令可以实现为对象，或者是带有运算符字段和运算分量字段的记录。四元式、三元式和间接三元式是三种这样的描述方式。

一个*四元式*(quadruple)有四个字段，我们分别称为 *op*、arg_1、arg_2、*result*。字段 *op* 包含一个运算符的内部编码。举例来说，在三地址指令 $x = y + z$ 相应的四元式中，*op* 字段中存放 +，arg_1 中为 y，arg_2 中为 z，*result* 中为 x。下面是这个规则的一些特例：

1）形如 x = minus y 的单目运算符指令和赋值指令 $x = y$ 不使用 arg_2。注意，对于像 $x = y$ 这样的赋值语句，*op* 是 =，而对大部分其他运算而言，赋值运算符是隐含表示的。

2）像 param 这样的运算既不使用 arg_2，也不使用 *result*。

3）条件或非条件转移指令将目标标号放入 *result* 字段。

例 6.6 赋值语句 a = b * - c + b * - c 的三地址代码如图 6-10a 所示。这里我们使用特殊的 minus 运算符来表示“ - c”中的单目减运算符“ - ”，以区别于“b - c”中的双目减运算符“ - ”。请注意，单目减的三地址语句中只有两个地址，复制语句 a = t_5 也是如此。

图 6-10b 描述了实现图 6-10a 中三地址代码的四元式序列。 □

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

a) 三地址代码

	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
	...			

b) 四元式

图 6-10 三地址代码及其四元式表示

为了提高可读性，我们在图 6-10b 中直接用实际标识符，比如用 a、b、c 来描述 arg_1、arg_2 以及 *result* 字段，而没有使用指向相应符号表条目的指针。临时名字可以像程序员定义的名字一样被加入到符号表中，也可以实现为 *Temp* 类的对象，这个 *Temp* 类有自己的方法。

6.2.3 三元式表示

一个*三元式*(triple)只有三个字段，我们分别称之为 *op*、arg_1 和 arg_2。请注意，图 6-10b 中的 *result*字段主要被用于临时变量名。使用三元式时，我们将用运算 x *op* y 的位置来表示它的结果，而不是用一个显式的临时名字表示。例如，在三元式表示中将直接用位置(0)，而不是像图 6-10b 中那样用临时名字 t_1 来表示对相应运算结果的引用。带有括号的数字表示指向相应三元式结构的指针。在 6.1.2 节中，位置或指向位置的编码被称为值编码。

三元式基本上和算法 6.3 中的结点范型等价。因此，表达式的 DAG 表示和三元式表示是等价的。当然这种等价关系仅对表达式成立，因为语法树的变体和三地址代码分别以完全不同的方式来表示控制流。

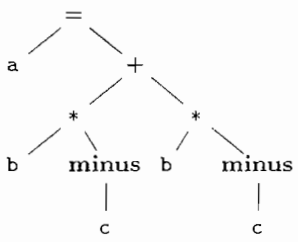
例 6.7 图 6-11 中给出的语法树和三元式表示对应于图 6-10 中的三地址代码及四元式序列。

在图 6-11b 给出的三元式表示中，复制语句 a = t_5 按照下列方式表示为一个三元式：在字段 arg_1 中放置 a，而在字段 arg_2 中放置三元式位置的值编码(4)。 □

像 $x[i] = y$ 这样的三元运算在三元式结构中需要两个条目。例如，我们可以把 x 和 i 置于一个三元式中，并把 y 置于另一个三元式中。类似的，我们可以把 $x = y[i]$ 看成是两条指令 $t = y[i]$ 和 $x = t$，从而用三元式实现这个语句。其中的 t 是编译器生成的临时变量。请注意，实际上 t 是不会出现在三元式中的，因为在三元式结构中是通过相应三元式结构的位置来引用临时值的。

为什么我们需要复制指令?

如图 6-10a 所示，一个简单的翻译表达式的算法往往会为赋值运算生成复制指令。在该图中，我们将 t_5 复制给 a，而不是直接将 $t_2 + t_4$ 赋给 a。通常，每个子表达式都会有一个它自己的新临时变量来存放运算结果。只有当处理赋值运算符 = 时，我们才知道将把整个表达式的结果赋到哪里。一个代码优化过程将会发现 t_5 可以被替换为 a。这个优化过程可能使用 6.1.1 节中描述的 DAG 作为中间表示形式。



a) 语法树

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

b) 三元式

图 6-11 $a = b * -c + b * -c$ 的表示

在优化编译器中，由于指令的位置常常会发生变化，四元式相对于三元式的优势就体现出来了。使用四元式时，如果我们移动了一个计算临时变量 t 的指令，那些使用 t 的指令不需要做任何改变。而使用三元式时，对于运算结果的引用是通过位置完成的，因此如果改变一条指令的位置，则引用该指令的结果的所有指令都要做相应的修改。使用下面将要介绍的间接三元式时就不会出现这个问题。

间接三元式(indirect triple)包含了一个指向三元式的指针的列表，而不是列出三元式序列本身。例如，我们可以使用数组 *instruction* 按照适当的顺序列出指向三元式的指针。这样，图 6-11b 中的三元式序列就可以表示成为图 6-12 所示的形式。

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

图 6-12 三地址代码的间接三元式表示

使用间接三元式表示方法时，优化编译器可以通过对 instruction 列表的重新排序来移动指令的位置，但不影响三元式本身。在用 Java 实现时，一个指令对象的数组和间接三元式表示类似，因为 Java 将数组元素作为对象引用来处理。

6.2.4 静态单赋值形式

静态单赋值形式(SSA)是另一种中间表示形式，它有利于实现某些类型的代码优化。SSA 和三地址代码的区别主要体现在两个方面。首先，SSA 中的所有赋值都是针对具有不同名字的变量的，这也是“静态单赋值”这一名字的由来。图 6-13 给出了分别以三地址代码形式和静态单赋值形式表示的中间程序。注意，SSA 表示中对变量 p 和 q 的每次定值都以不同的下标加以区分。

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

a) 三地址代码

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```

b) 静态单赋值形式

图 6-13 三地址代码形式和 SSA 形式的中间程序

在一个程序中，同一个变量可能在两个不同的控制流路径中被定值。例如，下列源程序

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

中，x 在两个不同的控制流路径中被定值。如果我们对条件语句的真分支和假分支中的 x 使用不同的变量名，那么我们应该在赋值运算 y = x * a 中使用哪个名字？这也是 SSA 的第二个特别之处。SSA 使用一种被称为 φ 函数的表示规则将 x 的两处定值合并起来：

```
if ( flag ) x1 = -1; else x2 = 1;
x3 = φ(x1,x2);
```

如果控制流经过这个条件语句的真分支，$\phi(x_1, x_2)$ 的值为 x_1；否则，如果控制流经过假分支，ϕ 函数的值为 x_2。也就是说，根据到达包含 ϕ 函数的赋值语句的不同控制流路径，ϕ 函数返回不同的参数值。

6.2.5 6.2 节的练习

练习 6.2.1：将算术表达式 a + -(b + c)翻译成

1) 抽象语法树

2) 四元式序列

3) 三元式序列

4) 间接三元式序列

练习 6.2.2：对下列赋值语句重复练习 6.2.1。

1) a = b[i] + c[j]

2) a[i] = b*c - b*d

3) x = f(y+1) + 2

4) x = *p + &y

! **练习 6.2.3**：说明如何对一个三地址代码序列进行转换，使得每个被定值的变量都有唯一的变量名。

6.3 类型和声明

可以把类型的应用划分为类型检查和翻译：

- 类型检查(type checking)。类型检查利用一组逻辑规则来推理一个程序在运行时刻的行

为。更明确地讲，类型检查保证运算分量的类型和运算符的预期类型相匹配。例如，Java 要求 && 运算符的两个运算分量必须是 boolean 型。如果满足这个条件，结果也具有 boolean 类型。

- 翻译时的应用(translation application)。根据一个名字的类型，编译器可以确定这个名字在运行时刻需要多大的存储空间。类型信息还会在其他很多地方被用到，包括计算一个数组引用所指示的地址，插入显式的类型转换，选择正确版本的算术运算符，等等。

在这一节中，我们将考虑在某个过程或类中声明的名字的类型及存储空间布局问题。一个过程调用或对象的实际存储空间是在运行时刻(当该过程被调用或该对象被创建时)进行分配的。然而，当我们在编译时刻检查局部声明时，可以进行相对地址(relative address)的布局，一个名字或某个数据结构分量的相对地址是指它相对于数据区域开始位置的偏移量。

6.3.1 类型表达式

类型自身也有结构，我们使用*类型表达式*(type expression)来表示这种结构：类型表达式可能是基本类型，也可能通过把被称为*类型构造算子*的运算符作用于类型表达式而得到。基本类型的集合和类型构造算子根据被检查的具体语言而定。

例 6.8 数组类型 `int[2][3]`表示“由两个数组组成的数组，其中的每个数组各包含 3 个整数”。它的类型表达式可以写成 *array*(2, *array*(3, *integer*))。该类型可以用如图 6-14 所示的树来描述。*array* 运算符有两个参数：一个数字和一个类型。 □

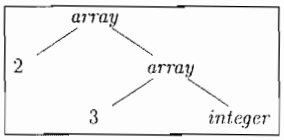


图 6-14 `int[2][3]`的类型表达式

我们将使用如下的类型表达式的定义：

- 基本类型是一个类型表达式。一种语言的基本类型通常包括 *boolean*、*char*、*integer*、*float* 和 *void*。最后一个类型表示“没有值”。
- 类名是一个类型表达式。
- 将类型构造算子 *array* 作用于一个数字和一个类型表达式可以得到一个类型表达式。
- 一个记录是包含有名字段的数据结构。将 *record* 类型构造算子应用于字段名和相应的类型可以构造得到一个类型表达式。在 6.3.6 节中，记录类型的实现方法是把构造算子 *record* 应用于包含了各个字段对应条目的符号表。
- 使用类型构造算子→可以构造得到函数类型的类型表达式。我们把“从类型 s 到类型 t 的函数”写成 $s \rightarrow t$。在 6.5 节中讨论类型检查时，函数类型是有用的。
- 如果 s 和 t 是类型表达式，则其笛卡儿积 $s \times t$ 也是类型表达式。引入笛卡儿积主要是为了保证定义的完整性。它可以用于描述类型的列表或元组(例如，用于表示函数参数)。我们假定 × 具有左结合性，并且其优先级高于→。
- 类型表达式可以包含取值为类型表达式的变量。在 6.5.4 节中将用到编译器产生的类型变量。

图是表示类型表达式的一种比较方便的方法。可以修改 6.1.2 节中给出的值编码方法，以构造一个类型表达式的 DAG。图的内部结点表示类型构造算子，而叶子结点是基本类型、类型名或类型变量。6.1.4 给出了一棵树的实例[⊖]。

⊖ 类型名代表类型表达式，因此可能形成隐式的环，见“类型名和递归类型”部分。如果到达类型名的边被重定向到该名字对应的类型表达式，那么得到的图中就可能因为存在递归类型而出现环。

类型名和递归类型

在 C++ 和 Java 中，类一旦被定义，其名字就可以被用来表示类型名。例如，考虑下列程序片段中的 Node 类。

```
public class Node { ··· }
···
public Node n;
```

类型名还可以用来定义递归类型，在像链表这样的数据结构中要用到递归类型。一个列表元素的伪代码如下：

```
class Cell { int info; Cell next; ··· }
```

它定义了一个递归类型 Cell。这个类包括一个 info 字段和另一个 Cell 类型的字段 next。在 C 中可以通过记录和指针来定义类似的递归类型。本章介绍的技术也适用于递归类型。

6.3.2 类型等价

两个类型表达式什么时候等价呢？很多类型检查规则具有这样的形式，“**如果**两个类型表达式相等，**那么**返回某种类型，**否则**出错”。当给一些类型表达式命名，并且这些名字在之后的其他类型表达式中使用时就可能会产生歧义。关键问题在于一个类型表达式中的名字是代表它自身呢，还是被看作另一个类型表达式的一种缩写形式。

当用图来表示类型表达式的时候，两种类型之间*结构等价*(structurally equivalent)当且仅当下面的某个条件为真：

- 它们是相同的基本类型。
- 它们是将相同的类型构造算子应用于结构等价的类型而构造得到。
- 一个类型是另一个类型表达式的名字。

如果类型名仅仅代表它自身，那么上述定义中的前两个条件定义了类型表达式的*名等价*(name equivalence)关系。

如果我们使用算法 6.3，那么名等价表达式将被赋予相同的值编码。结构等价关系可以使用 6.5.5 节中给出的合一算法进行检验。

6.3.3 声明

我们在研究类型及其声明时将使用一个经过简化的文法，在这个文法中一次只声明一个名字。一次声明多个名字的情况可以像例 5.10 中讨论的那样进行处理。我们使用的文法如下：

$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \mathbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$

上述处理基本类型和数组类型的文法，可以用来演示 5.3.2 节中描述的继承属性。本节的不同之处在于我们不仅考虑类型本身，还考虑各个类型的存储布局。

非终结符号 D 生成一系列声明。非终结符号 T 生成基本类型、数组类型或记录类型。非终结符号 B 生成基本类型 int 和 float 之一。非终结符号 C(表示“分量”)产生零个或多个整数，每个整数用方括号括起来。一个数组类型包含一个由 B 指定的基本类型，后面跟一个由非终结符号 C 指定的数组分量。一个记录类型(T 的第二个产生式)由各个记录字段的声明序列构成，并被花括号括起来。

6.3.4 局部变量名的存储布局

从变量类型我们可以知道该变量在运行时刻需要的内存数量。在编译时刻，我们可以使用这些数量为每个名字分配一个相对地址。名字的类型和相对地址信息保存在相应的符号表条目

中。对于字符串这样的变长数据，以及动态数组这样的只有在运行时刻才能够确定其大小的数据，处理的方法是为指向这些数据的指针保留一个已知的固定大小的存储区域。运行时刻的存储管理问题将在第 7 章中讨论。

地址对齐

数据对象的存储布局受目标机器的寻址约束的影响。比如，将整数相加的指令往往希望整数能够对齐(aligned)，也就是说，希望它们被放在内存中的特定位置上，比如地址能够被 4 整除的位置上。虽然一个有 10 个字符的数组只需要足以存放 10 个字符的字节空间，但编译器常常会给它分配 12 个字节，即下一个 4 的倍数，这样会有 2 个字节没有使用。因为对齐的要求而分配的无用空间被称为“补白”(padding)。当空间比较宝贵时，编译器需要对数据进行压缩(pack)，此时不存在“补白”空间，但可能需要在运行时刻执行额外的指令把被压缩的数据重新定位，以便这些数据看上去仍然是对齐的，从而进行相关运算。

假设存储区域是连续的字节块，其中字节是可寻址的最小内存单位。一个字节通常有 8 个二进制位，若干字节组成一个机器字。多字节数据对象往往被存储在一段连续的字节中，并以初始字节的地址作为该数据对象的地址。

类型的宽度(width)是指该类型的一个对象所需的存储单元的数量。一个基本类型，比如字符型、整型和浮点型，需要整数多个的字节。为方便访问，为数组和类这样的组合类型数据分配的内存是一个连续的存储字节块[⊖]。

图 6-15 中给出的翻译方案(SDT)计算了基本类型和数组类型以及它们的宽度。记录类型将在 6.3.6 节中讨论。这个 SDT 为每个非终结符号使用综合属性 *type* 和 *width*。它还使用了两个变量 t 和 w，变量的用途是将类型和宽度信息从语法分析树中的 B 结点传递到对应于产生式 $C \rightarrow \epsilon$ 的结点。在语法制导定义中，t 和 w 将是 C 的继承属性。

$T \rightarrow B$		$\{ t = B.type;\ w = B.width; \}$
	C	$\{ T.type = C.type;\ T.width = C.width \}$
$B \rightarrow$	**int**	$\{ B.type = integer;\ B.width = 4; \}$
$B \rightarrow$	**float**	$\{ B.type = float;\ B.width = 8; \}$
$C \rightarrow$	ϵ	$\{ C.type = t;\ C.width = w; \}$
$C \rightarrow$	[**num**] C_1	$\{ C.type = array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width; \}$

图 6-15 计算类型及其宽度

T 产生式的产生式体包含一个非终结符号 B、一个动作和一个非终结符号 C，其中 C 显示在下一行上。B 和 C 之间的动作是将 t 设置为 $B.type$，并将 w 设置为 $B.width$。如果 $B \rightarrow$ **int**，则 $B.type$ 被设置为 *integer*，$B.width$ 被设置为 4，即一个整型数的宽度。类似的，如果 $B \rightarrow$ **float**，则 $B.type$ 和 $B.width$ 分别被设置为 *float* 和 8，即宽度为一个浮点数的宽度。

C 的产生式决定了 T 生成的是一个基本类型还是一个数组类型。如果 $C \rightarrow \epsilon$，则 t 变成 $C.type$，且 w 变成 $C.width$。

否则，C 就描述了一个数组分量。$C \rightarrow [\mathbf{num}]C_1$ 的动作将类型构造算子 *array* 应用于运算分量 **num**.*value* 和 $C_1.type$，构造得到 $C.type$。例如，应用 *array* 的结果可能是图 6-14 所示的树形结构。

⊖ 在 C 或 C++ 中，如果所有的指针具有相同的宽度，那么指针的存储分配就比较简单。其原因是我们可以在知道它所指向对象的类型之前就为它分配存储空间。

数组的宽度是将数组元素的个数乘以单个数组元素的宽度而得到的。如果连续存放的整数的地址之间的差距为4，那么一个整数数组的地址计算将包含乘4运算。这样的乘法运算为优化提供了机会，因此让前端程序在其输出中明确描述这些运算将有助于优化。在这一章中，我们将忽略其他与机器相关特性，比如数据对象的地址必须和机器字的边界对齐。

例6.9 类型 int[2][3]的语法分析树用图6-16中的虚线描述。图中的实线描述了 *type* 和 *width* 是如何从 *B* 结点开始，通过变量 *t* 和 *w*，沿着多个 *C* 组成的链下传，然后又作为综合属性 *type* 和 *width* 沿此链返回的。在访问包含 *C* 结点的子树之前，变量 *t* 和 *w* 被赋予 *B.type* 和 *B.width* 的值。变量 *t* 和 *w* 的值在 $C \rightarrow \epsilon$ 对应的结点上使用，然后开始沿着多个 *C* 结点组成的链向上对综合属性求值。 □

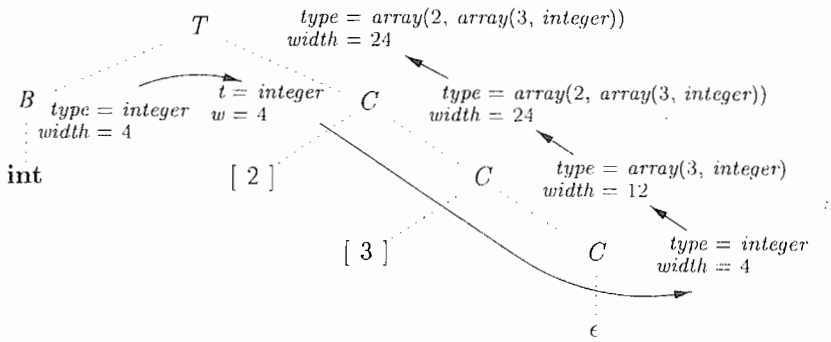


图6-16 数组类型的语法制导翻译

6.3.5 声明的序列

像C和Java这样的语言支持将单个过程中的所有声明作为一个组进行处理。这些声明可能分布在一个Java过程中，但是仍然能够在分析该过程时处理它们。因此，我们可以使用一个变量，比如 *offset*，来跟踪下一个可用的相对地址。

图6-17中的翻译方案处理形如 *T* **id** 的声明的序列，其中的 *T* 如图6-15所示产生一个类型。在考虑第一个声明之前，*offset* 被设置为0。每处理一个变量 *x* 时，*x* 被加入符号表，它的相对地址被设置为 *offset* 的当前值。随后，*x* 的类型的宽度被加到 *offset* 上。

产生式 $D \rightarrow T$ **id** ; D_1 中的语义动作首先执行 *top.put*(**id**.*lexeme*, *T.type*, *offset*)，创建一个符号表条目。这里的 *top* 指向当前的符号表。方法 *top.put* 为 **id**.*lexeme* 创建一个符号表条目，该条目的数据区中存放了类型 *T.type* 和相对地址 *offset*。

```
P → { offset = 0; }
      D
D → T id ;  { top.put(id.lexeme, T.type, offset);
              offset = offset + T.width; }
      D1
D → ϵ
```

图6-17 计算被声明变量的相对地址

如果我们把第一个产生式写在同一行中：

$$P \rightarrow \{ offset = 0; \}\ D \tag{6.1}$$

则图6-17中对 *offset* 的初始化处理就变得更容易理解。生成 ϵ 的非终结符号称为标记非终结符号，其作用是重写产生式，使得所有的语义动作都出现在产生式右部的尾端，具体方法见5.5.4节。使用标记非终结符号 *M*，(6.1)可以被改写为：

$$P \rightarrow M\ D$$
$$M \rightarrow \epsilon\ \{ offset = 0; \}$$

6.3.6 记录和类中的字段

图 6-17 中对声明的翻译方案还可以用于处理记录和类中的字段。要把记录类型加入到图 6-15 所示的文法中，只需要加上下面的产生式：

$$T \rightarrow \textbf{record}\ '\{'\ D\ '\}'$$

这个记录类型中的字段由 D 生成的声明序列描述。图 6-17 中的方法可以用来确定这些字段的类型和相对地址，当然我们需要小心地处理下面两件事：

- 一个记录中各个字段的名字必须是互不相同的。也就是说，在由 D 生成的声明中，同一个名字最多出现一次。
- 字段名的偏移量，或者说相对地址，是相对于该记录的数据区字段而言的。

例 6.10 在一个记录中，把名字 x 用作字段名并不会和记录外对该名字的其他使用产生冲突。因此下列声明中对 x 的三次使用是不同的，互相之间并不冲突。

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

这些声明之后的一个赋值语句 `x = p.x + q.x;` 把变量 `x` 的值设置为记录 `p` 和 `q` 中 `x` 字段的值的和。请注意，`p` 中 `x` 的相对地址和 `q` 中 `x` 的相对地址是不同的。 □

为方便起见，记录类型将使用一个专用的符号表，对它们的各个字段的类型和相对地址进行编码。记录类型形如 *record*(t)，其中 *record* 是类型构造算子，t 是一个符号表对象，它保存了有关该记录类型的各个字段的信息。

图 6-18 中的翻译方案包含一个产生式，该产生式将加入到图 6-15 中关于 T 的产生式中。这个产生式有两个语义动作。在 D 之前嵌入的动作首先保存 *top* 指向的已有符号表，然后让 *top* 指向新的符号表。该动作还保存了当前 *offset* 值，并将 *offset* 重置为 0。D 生成的声明会使类型和相对地址被保存到新的符号表中。D 之后的语义动作使用 *top* 创建一个记录类型，然后恢复早先保存好的符号表和偏移值。

$T \rightarrow$ **record** '{'	{ *Env.push*(*top*); *top* = **new** *Env*(); *Stack.push*(*offset*); *offset* = 0; }
D '}'	{ *T.type* = *record*(*top*); *T.width* = *offset*; *top* = *Env.pop*(); *offset* = *Stack.pop*(); }

图 6-18 处理记录中的字段名

为了使翻译方案更加具体，图 6-18 中的动作给出了某个实现的伪代码。令 *Env* 类实现符号表。对 *Env.push*(*top*) 的调用将 *top* 所指的当前符号表压入一个栈中。然后，变量 *top* 被设置为指向一个新的符号表。类似的，*offset* 被推入名为 *Stack* 的栈中，*offset* 变量被重置为 0。

在 D 中的声明被翻译之后，符号表 *top* 保存了这个记录中所有字段的类型和相对地址。而且，*offset* 还给出了存放所有字段所需的存储空间。第二个动作将 *T.type* 设为 *record*(*top*)，并将 *T.width* 设为 *offset*。然后，变量 *top* 和 *offset* 将被恢复为原先被压入栈中的值，以完成这个记录类型的翻译。

有关记录类型存储方式的讨论还可以被推广到类，因为我们无需为类中的方法保留存储空间。见练习 6.3.2。

6.3.7 6.3 节的练习

练习 6.3.1：确定下列声明序列中各个标识符的类型和相对地址。

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

! 练习 6.3.2：将图 6-18 对字段名的处理方法扩展到类和单继承的类层次结构。

1）给出类 *Env* 的一个实现。该实现支持符号表链，使得子类可以重定义一个字段名，也可以直接引用某个超类中的字段名。

2）给出一个翻译方案，该方案能够为类中的字段分配连续的数据区域，这些字段中包含继承而来的域。继承而来的字段必须保持在对超类进行存储分配时获得的相对地址。

6.4 表达式的翻译

本章剩下的部分将介绍在翻译表达式和语句时出现的问题。在本节中，我们首先考虑从表达式到三地址代码的翻译。一个带有多个运算符的表达式（比如 $a + b * c$）将被翻译成为每条指令最多包含一个运算符的指令序列。一个数组引用 $A[i][j]$ 将被扩展成一个计算该引用的地址的三地址指令序列。我们将在 6.5 节中考虑表达式的类型检查，并在 6.6 节中介绍如何使用布尔表达式来处理程序的控制流。

6.4.1 表达式中的运算

图 6-19 中的语法制导定义使用 S 的属性 *code* 以及表达式 E 的属性 *addr* 和 *code*，为一个赋值语句 S 生成三地址代码。属性 *S. code* 和 *E. code* 分别表示 S 和 E 对应的三地址代码。属性 *E. addr* 则表示存放 E 的值的地址。回忆一下 6.2.1 节，一个地址可以是变量名字、常量或编译器产生的临时量。

产生式	语义规则
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

图 6-19 表达式的三地址代码

考虑图 6-19 中语法制导定义的最后一个产生式 $E \rightarrow \mathbf{id}$。若表达式只是一个标识符，比如说 x，那么 x 本身就保存了这个表达式的值。这个产生式对应的语义规则把 *E. addr* 定义为指向该 **id** 的实例对应的符号表条目的指针。令 *top* 表示当前的符号表。当函数 *top. get* 被应用于 **id** 的这个实例的字符串表示 **id**. *lexeme* 时，它返回对应的符号表条目。*E. code* 被设置为空串。

当规则为 $E \rightarrow (E_1)$ 时，对 E 的翻译与对子表达式 E_1 的翻译相同。因此，*E. addr* 等于 E_1. *addr*，*E. code* 等于 E_1. *code*。

图 6-19 中的运算符 + 和单目 - 是典型语言中的运算符的代表。$E \rightarrow E_1 + E_2$ 的语义规则生成了根据 E_1 和 E_2 的值计算 E 的值的代码。计算得到的值存放在新生成的临时变量中。如果 E_1 的

值计算后被放入 $E_1.addr$，E_2 的值被放到 $E_2.addr$ 中，那么 E_1+E_2 就可以被翻译为 $t=E_1.addr+E_2.addr$，其中 t 是一个新的临时变量。$E.addr$ 被设为 t。连续执行 **new** *Temp*( ) 会产生一系列互不相同的临时变量 t_1，t_2，…。

为方便起见，我们使用记号 $gen(x\ '='\ y\ '+'\ z)$ 来表示三地址指令 $x=y+z$。当被传递给 *gen* 时，变量 x、y、z 的位置上出现的表达式将首先被求值，而像 '=' 这样的引号内的字符串则按照字面值传递[⊖]。其他的三地址指令的生成方法类似，也是将 *gen* 作用于表达式和字符串的组合。

当我们翻译产生式 $E\rightarrow E_1+E_2$ 时，图 6-19 中的语义规则首先将 $E_1.code$ 和 $E_2.code$ 连接起来，然后再加上一条将 E_1 和 E_2 的值相加的指令，从而生成 $E.code$。新增加的这条指令将求和的结果放入一个为 E 生成的临时变量中，用 $E.addr$ 表示。

产生式 $E\rightarrow -E_1$ 的翻译过程与此类似。这个规则首先为 E 创建一个新的临时变量，并生成一条指令来执行单目 - 运算。

最终，产生式 $S\rightarrow$ **id** $=E$; 所生成的指令将表达式 E 的值赋给标识符 **id**。和规则 $E\rightarrow$ **id** 中一样，这个产生式的语义规则使用函数 *top.get* 来确定 **id** 所代表的标识符的地址。$S.code$ 包含的指令首先计算 E 的值并将其保存到由 $E.addr$ 指定的地址中，然后再将这个值赋给这个 **id** 实例的地址 *top.get*(**id**.*lexeme*)。

例 6.11 图 6-19 中的语法制导定义将赋值语句 a = b + - c; 翻译成如下的三地址代码序列：

```
t1 = minus c
t2 = b + t1
a = t2
```

□

6.4.2 增量翻译

code 属性可能是很长的字符串，因此就像 5.5.2 节中讨论的那样，它们通常是用增量的方式生成的。因此，我们不会像图 6-19 所示的那样构造 $E.code$，我们可以设法像图 6-20 中那样只生成新的三地址指令。在这个增量方式中，*gen* 不仅要构造出一个新的三地址指令，还要将它添加到至今为止已生成的指令序列之后。指令序列可以暂时放在内存中以便进一步处理，也可以增量地输出。

图 6-20 中的翻译方案和图 6-19 中的语法制导定义产生相同的代码。采用增量方式时不需再用到 *code* 属性，因为对 *gen* 的连续调用将生成一个指令序列。例如，图 6-20 中对应于 $E\rightarrow E_1+E_2$ 的语义规则直接调用 *gen* 产生一条加法指令。在此之前，翻译方案已经生成了计算 E_1 的值并放入 $E_1.addr$、计算 E_2 的值并放入 $E_2.addr$ 的指令序列。

$S \rightarrow$ **id** $= E$;	{ *gen*(*top.get*(**id**.*lexeme*) '=' $E.addr$); }
$E \rightarrow E_1 + E_2$	{ $E.addr$ = **new** *Temp*(); *gen*($E.addr$ '=' $E_1.addr$ '+' $E_2.addr$); }
| $- E_1$	{ $E.addr$ = **new** *Temp*(); *gen*($E.addr$ '=' 'minus' $E_1.addr$); }
| (E_1)	{ $E.addr = E_1.addr$; }
| **id**	{ $E.addr$ = *top.get*(**id**.*lexeme*); }

图 6-20 增量生成表达式的三地址代码

图 6-20 的方法也可以用来构造语法树，对应于 $E\rightarrow E_1+E_2$ 的语义动作使用构造算子生成新的结点。规则如下：

$$E\rightarrow E_1+E_2\ \{E.addr = \textbf{new}\ Node('+', E_1.addr, E_2.addr);\ \}$$

这里，属性 *addr* 表示的是一个结点的地址，而不是某个变量或常量。

⊖ 在语法制导定义中，*gen* 构造出一条指令并返回它。在翻译方案中，*gen* 构造出一条指令，并增量地将它添加到指令流中去。

6.4.3 数组元素的寻址

将数组元素存储在一块连续的存储空间里就可以快速地访问它们。在 C 和 Java 中，一个具有 n 个元素的数组中的元素是按照 0，1，…，$n-1$ 编号的。假设每个数组元素的宽度是 w，那么数组 A 的第 i 个元素的开始地址为

$$base + i \times w \tag{6.2}$$

其中 *base* 是分配给数组 A 的内存块的相对地址。也就是说，*base* 是 $A[0]$的相对地址。

式(6.2)可以被推广到 C 语言中的二维或多维数组上。对于二维数组，我们在 C 中用 $A[i_1][i_2]$来表示第 i_1 行的第 i_2 个元素。假设一行的宽度是 w_1，同一行中每个元素的宽度是 w_2。$A[i_1][i_2]$的相对地址可以使用下面的公式计算

$$base + i_1 \times w_1 + i_2 \times w_2 \tag{6.3}$$

对于 k 维数组，相应的公式为

$$base + i_1 \times w_1 + i_2 \times w_2 + \cdots + i_k \times w_k \tag{6.4}$$

其中，$w_j(1 \leqslant j \leqslant k)$是对式(6.3)中的 w_1 和 w_2 的推广。

另一种计算数组引用的相对地址的方法是根据第 j 维上的数组元素的个数 n_j 和该数组的每个元素的宽度 $w = w_k$ 进行计算。在二维数组中(即 $k=2$，$w=w_2$)，$A[i_1][i_2]$的地址为

$$base + (i_1 \times n_2 + i_2) \times w \tag{6.5}$$

对于 k 维数组，下列公式计算得到的地址和公式(6.4)所得到的地址相同：

$$base + ((\cdots((i_1 \times n_2 + i_2) \times n_3 + i_3)\cdots) \times n_k + i_k) \times w \tag{6.6}$$

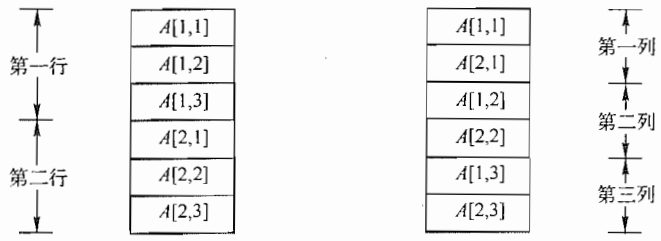
在更一般的情况下，数组元素下标并不一定是从 0 开始的。在一个一维数组中，数组元素的编号方式如下：*low*，*low* +1，…，*high*，而 *base* 是 $A[low]$的相对地址。计算 $A[i]$的地址的式(6.2)就变成：

$$base + (i - low) \times w \tag{6.7}$$

式(6.2)和式(6.7)都可以改写成 $i \times w + c$ 的形式，其中的子表达式 $c = base - low \times w$ 可以在编译时刻预先计算出来。请注意，当 *low* 为 0 时 $c = base$。我们假定 c 被存放在 A 对应的符号表条目中，那么只要把 $i \times w$ 加到 c 上就可以计算得到 $A[i]$的相对地址。

编译时刻的预先计算同样可以应用于多维数组元素的地址计算，见练习 6.4.5。然而，有一种情况下我们不能使用编译时刻预先计算的技术：当数组大小是动态变化的时候。如果我们在编译时刻无法知道 *low* 和 *high*(或者它们在多维数组情况下的泛化)的值，我们就无法提前计算出像 c 这样的常量。因此在程序运行时，像(6.7)这样的公式就需要按照公式所写进行求值。

上面的地址计算是基于数组的按行存放方式的，C 语言都使用这种数据布局方式。一个二维数组通常有两种存储方式，即*按行存放*(一行行地存放)和*按列存放*(一列列地存放)。图 6-21 显示了一个 2×3 的数组 A 的两种存储布局方式，图 6-21a 中是按行存放方式，图 6-21b 中是按列存放方式。Fortran 系列语言使用按列存放方式。



a) 按行存放　　b) 按列存放

图 6-21　二维数组的存储布局

我们可以把按行存放策略和按列存放策略推广到多维数组中。按行存放方式的推广形式按照如下方式来存储元素：当我们扫描一块存储区域时，就像汽车里程表中的数字一样，最右边的下标变化最为频繁。而按列存放方式则被推广为相反的布局方式，最左边的下标变化最频繁。

6.4.4 数组引用的翻译

为数组引用生成代码时要解决的主要问题是将 6.4.3 节中给出的地址计算公式和数组引用的文法关联起来。令非终结符号 L 生成一个数组名字再加上一个下标表达式的序列：

$$L \rightarrow L[E] \mid \mathbf{id}\ [E]$$

与 C 和 Java 中一样，我们假定数组元素的最小编号是 0。我们使用式(6.4)，基于宽度来计算相对地址，而不是像式(6.6)中那样使用元素的数量来计算地址。图 6-22 所示的翻译方案为带有数组引用的表达式生成三地址代码。它包括了图 6-20 中给出的产生式和语义动作，同时还包括了涉及非终结符号 L 的产生式。

$S \rightarrow \mathbf{id} = E\ ;$	{ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$; }
$\mid\ L = E\ ;$	{ $gen(L.array.base\ '['\ L.addr\ ']'\ '='\ E.addr)$; }
$E \rightarrow E_1 + E_2$	{ $E.addr = \mathbf{new}\ Temp()$; $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$; }
$\mid\ \mathbf{id}$	{ $E.addr = top.get(\mathbf{id}.lexeme)$; }
$\mid\ L$	{ $E.addr = \mathbf{new}\ Temp()$; $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']')$; }
$L \rightarrow \mathbf{id}\ [\ E\ ]$	{ $L.array = top.get(\mathbf{id}.lexeme)$; $L.type = L.array.type.elem$; $L.addr = \mathbf{new}\ Temp()$; $gen(L.addr\ '='\ E.addr\ '*'\ L.type.width)$; }
$\mid\ L_1\ [\ E\ ]$	{ $L.array = L_1.array$; $L.type = L_1.type.elem$; $t = \mathbf{new}\ Temp()$; $L.addr = \mathbf{new}\ Temp()$; $gen(t\ '='\ E.addr\ '*'\ L.type.width)$; $gen(L.addr\ '='\ L_1.addr\ '+'\ t)$; }

图 6-22 处理数组引用的语义动作

非终结符号 L 有三个综合属性：

1) $L.addr$ 指示一个临时变量。这个临时变量将被用于累加公式(6.4)中的 $i_j \times w_j$ 项，从而计算数组引用的偏移量。

2) $L.array$ 是一个指向数组名字对应的符号表条目的指针。在分析了所有的下标表达式之后，该数组的基地址，也就是 $L.array.base$，被用于确定一个数组引用的实际左值。

3) $L.type$ 是 L 生成的子数组的类型。对于任何类型 t，我们假定其宽度由 $t.width$ 给出。我们把类型(而不是宽度)作为属性，是因为无论如何类型检查总是需要这个类型信息。对于任何数组类型 t，假设 $t.elem$ 给出了其数组元素的类型。

产生式 $S \rightarrow \mathbf{id} = E;$ 代表一个对非数组变量的赋值语句，它按照通常的方法进行处理。$S \rightarrow L = E;$ 的语义动作产生了一个带下标的复制指令，它将表达式 E 的值存放到数组引用 L 所指的内存位置。回顾一下，属性 $L.array$ 给出了数组的符号表条目。数组的基地址(即 0 号元素的地址)由 $L.array.base$ 给出。属性 $L.addr$ 表示一个临时变量，它保存了 L 生成的数组引用的偏移

量。因此，这个数组引用的位置是 *L. array. base*[*L. addr*]。这个指令将地址 *E. addr* 中的右值放入 *L* 的内存位置中。

产生式 $E \rightarrow E_1 + E_2$ 和 $E \rightarrow$ **id** 与以前相同。新的产生式 $E \rightarrow L$ 的语义动作生成的代码将 *L* 所指位置上的值复制到一个新的临时变量中。和前面对产生式 $S \rightarrow L = E$; 的讨论一样，*L* 所指的地址就是 *L. array. base*[*L. addr*]。其中，属性 *L. array* 仍然给出了数组名，*L. array. base* 给出了数组的基地址。属性 *L. addr* 表示保存偏移量的临时变量。数组引用的代码将存放在由基地址和偏移量给出的位置中的右值放入 *E. addr* 所指的临时变量中。

例 6.12 令 a 表示一个 2×3 的整数数组，c、i、j 都是整数。那么 a 的类型就是 *array*(2, *array*(3, *integer*))。假定一个整数的宽度为 4，那么 a 的类型的宽度就是 24。a[i]的类型是 *array*(3, *integer*)，宽度 w_1 为 12。a[i][j]的类型是整型。

图 6-23 给出了表达式 c + a[i][j]的注释语法分析树。该表达式被翻译成图 6-24 中给出的三地址代码序列。这里我们仍然使用每个标识符的名字来表示它们的符号表条目。 □

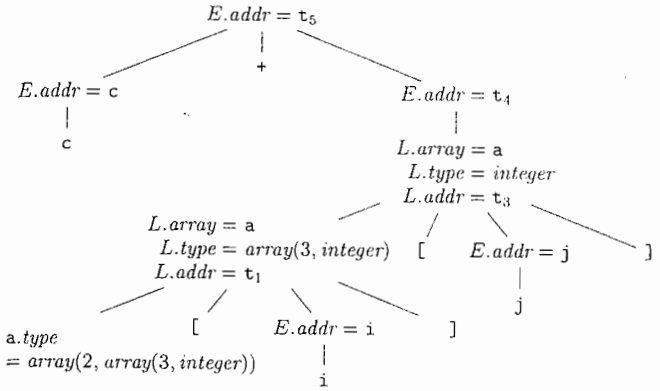


图 6-23 c + a[i][j]的注释语法分析树

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

图 6-24 表达式 c + a[i][j]的三地址代码

6.4.5 6.4 节的练习

练习 6.4.1：向图 6-19 的翻译方案中加入对应于下列产生式的规则：

1) $E \rightarrow E_1 * E_2$

2) $E \rightarrow + E_1$（单目加）

练习 6.4.2：使用图 6-20 中的增量式翻译方案重复练习 6.4.1。

练习 6.4.3：使用图 6-22 所示的翻译方案来翻译下列赋值语句：

1) `x = a[i] + b[j]`

2) `x = a[i][j] + b[i][j]`

!3) `x = a[b[i][j]][c[k]]`

! **练习 6.4.4**：修改图 6-22 中的翻译方案，使之适合 Fortran 风格的数组引用，也就是说，*n* 维数组的引用为 **id**[E_1, E_2, ⋯, E_n]。

练习 6.4.5：将公式(6.7)推广到多维数组上，并指出哪些值可以被存放到符号表中并用来计算偏移量。考虑下列情况：

1) 一个二维数组 *A*，按行存放。第一维的下标从 l_1 到 h_1，第二维的下标从 l_2 到 h_2。单个数组元素的宽度为 *w*。

2) 其他条件和 1 相同，但是采用按列存放方式。

！3）一个 k 维的数组 A，按行存放，元素宽度为 w，第 j 维的下标从 l_j 到 h_j。

！4）其他条件和 3 相同，但是采用按列存放方式。

练习 6.4.6：一个按行存放的整数数组 $A[i, j]$ 的下标 i 的范围为 1～10，下标 j 的范围为 1～20。每个整数占 4 个字节。假设数组 A 从 0 字节开始存放，请给出下列元素的位置：

1）$A[4, 5]$ 2）$A[10, 8]$ 3）$A[3, 17]$

练习 6.4.7：假定 A 是按列存放的，重复练习 6.4.6。

练习 6.4.8：一个按行存放的实数型数组 $A[i, j, k]$ 的下标 i 的范围为 1～4，下标 j 的范围为 0～4，且下标 k 的范围为 5～10。每个实数占 8 个字节。假设数组 A 从 0 字节开始存放。计算下列元素的位置。

1）$A[3, 4, 5]$ 2）$A[1, 2, 7]$ 3）$A[4, 3, 9]$

练习 6.4.9：假定 A 是按列存放的，重复练习 6.4.8。

符号化表示的类型宽度

中间代码应该相对独立于目标机器，这样当代码生成器被替换为对应于另一台机器的代码生成器时，优化器不需要做出太大的改变。然而，正如我们刚刚描述的类型宽度计算方法所示，关于基本类型的信息被融合到了这个翻译方案中。例如，例 6.12 中假定每个整数数组的元素占 4 个字节。一些中间代码，如 Pascal 的 P-code，让代码生成器来填写数组元素的大小，因此中间代码独立于机器的字长。只要用一个符号常量来代替翻译方案中的（作为整数类型宽度的）4，我们就可以在我们的翻译方案中做到这一点。

6.5 类型检查

为了进行类型检查（type checking），编译器需要给源程序的每一个组成部分赋予一个类型表达式。然后，编译器要确定这些类型表达式是否满足一组逻辑规则。这些规则称为源语言的*类型系统*（type system）。

类型检查具有发现程序中的错误的潜能。原则上，如果目标代码在保存元素值的同时保存了元素类型的信息，那么任何检查都可以动态地进行。一个健全（sound）的类型系统可以消除对动态类型错误检查的需要，因为它可以帮助我们静态地确定这些错误不会在目标程序运行的时候发生。如果编译器可以保证它接受的程序在运行时刻不会发生类型错误，那么该语言的这个实现就被称为*强类型的*。

除了用于编译，类型检查的思想还可以用于提高系统的安全性，使得人们安全地导入和执行软件模块。Java 程序被编译成为机器无关的字节码，在字节码中包含了有关字节码中的运算的详细类型信息。导入的代码在被执行之前首先要进行类型检查，以防止因疏忽造成的错误和恶意攻击。

6.5.1 类型检查规则

类型检查有两种形式：综合和推导。*类型综合*（type synthesis）根据子表达式的类型构造出表达式的类型。它要求名字先声明再使用。表达式 $E_1 + E_2$ 的类型是根据 E_1 和 E_2 的类型定义的。一个典型的类型综合规则具有如下形式：

$$\begin{aligned} &\textbf{if} \quad f \text{的类型为} s \to t \text{ 且 } x \text{ 的类型为 } s \\ &\textbf{then} \ \text{表达式} f(x) \text{的类型为 } t \end{aligned} \tag{6.8}$$

这里，f 和 x 表示表达式，而 $s \to t$ 表示从 s 到 t 的函数。这个针对单参数函数的规则可以推广到带

有多个参数的函数。只要稍做修改，规则(6.8)就可以用于 $E_1 + E_2$，我们只需要把它看作一个函数应用 $add(E_1, E_2)$ 就可以了[⊖]。

类型推导(type inference)根据一个语言结构的使用方式来确定该结构的类型。先看一下6.5.4节中的例子，令 *null* 是一个测试列表是否为空的函数。那么，根据这个函数的使用 $null(x)$，我们可以指出 x 必须是一个列表类型。列表 x 中的元素类型是未知的，我们所知道的全部信息是：x 是一个列表类型，其元素类型当前未知。

代表类型表达式的变量使得我们可以考虑未知类型。我们可以用希腊字母 α、β 等作为类型表达式中的类型变量。

一个典型的类型推导规则具有下面的形式：

if $f(x)$ 是一个表达式， (6.9)

then 对某些 α 和 β，f 的类型为 $\alpha \rightarrow \beta$ 且 x 的类型为 α

在类似 ML 这样的语言中需要进行类型推导。ML 语言会检查类型，但是不需要对名字进行声明。

在本节中，我们考虑表达式的类型检查。检查语句的规则和检查表达式类型的规则类似。例如，我们可以把条件语句"**if** (E) S; "看作是对 E 和 S 应用 *if* 函数。令特殊类型 *void* 表示没有值的类型，那么 *if* 函数将被应用在一个布尔型和一个 *void* 型的对象上。此函数的结果类型是 *void*。

6.5.2 类型转换

考虑类似于 $x + i$ 的表达式，其中 x 是浮点数类型而 i 是整型。因为整数和浮点数在计算机中有不同的表示形式，而且使用不同的机器指令来完成整数和浮点数运算。编译器需要把 + 的某个运算分量进行转换，以保证在进行加法运算时两个运算分量具有相同的类型。

假定在必要的时候可以使用一个单目运算符(`float`)将整数转换成浮点数。例如，整数 2 在表达式 `2 * 3.14` 对应的代码中被转换成浮点数：

```
t1 = (float) 2
t2 = t1 * 3.14
```

我们可以扩展这样例子，考虑运算符的整型和浮点型版本。比如，`int *` 表示作用于整型运算分量的运算符，而 `float *` 表示作用于浮点型运算分量的运算符。

我们将扩展 6.4.2 节中的用于表达式翻译的翻译方案，以说明如何进行类型综合。我们引入另一个属性 *E.type*，该属性的值可以是 *integer* 或 *float*。和 $E \rightarrow E_1 + E_2$ 相关的规则可用如下的伪代码给出：

```
if ( E1.type = integer and E2.type = integer ) E.type = integer;
else if ( E1.type = float and E2.type = integer ) ···
···
```

随着需要转换的类型的增多，需要处理的不同情况也急剧增多。因此，在处理大量的类型时，精心组织用于类型转换的语义动作就变得非常重要。

不同语言具有不同的类型转换规则。图 6-25 中的 Java 的转换规则区分了拓宽(widening)转换和窄化(narrowing)转换。拓宽转换可以保持原有的信息，而窄化转换则可能丢失信息。拓宽规则通过图 6-25a 中的层次结构给出：在该层次结构中位于较低层的类型可以被拓宽为较高层的类型。因此，*char* 类型可以被拓宽为 *int* 型和 *float* 型，但是不可以被拓宽为 *short* 类型。窄化转换

⊖ 即使我们在确定类型时需要某些上下文信息，我们仍将使用"综合"这个术语。使用重载函数时(多个函数可能被赋予同一个名字)，在某些语言中，我们还需要考虑 $E_1 + E_2$ 的上下文才能确定其类型规则。

的规则如图 6-25b 所示：如果存在一条从 s 到 t 的路径，则可以将类型 s 窄化为类型 t。可以看出，*char*、*short*、*byte* 之间可以两两相互转换。

如果类型转换由编译器自动完成，那么这样的转换就称为*隐式转换*。隐式转换也称为自动类型转换(coercion)。在很多语言中，自动类型转换仅仅限于拓宽转换。如果程序员必须写出某些代码来引发类型转换运算，那么这个转换就称为*显式*的。显式转换也称为强制类型转换(cast)。

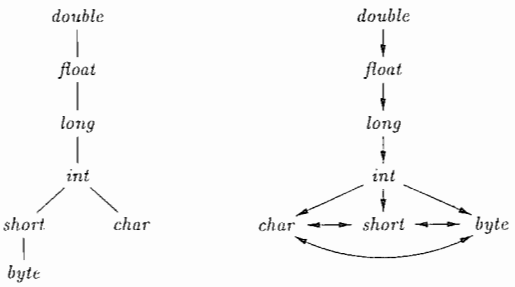


图 6-25 Java 中简单类型的转换

检查 $E \rightarrow E_1 + E_2$ 的语义动作使用了两个函数：

1) $max(t_1, t_2)$接受 t_1 和 t_2 两个类型的参数，并返回拓宽层次结构中这两个类型中的最大者(或者最小上界)。如果 t_1 或 t_2 之一没有出现在这个层次结构中，比如有个类型是数组类型或指针类型，那么该函数返回一个错误信息。

2) 如果需要将类型为 t 的地址 a 中的内容转换成 w 类型的值，则函数 $widen(a, t, w)$将生成类型转换的代码。如果 t 和 w 是相同的类型，则该函数返回 a 本身。否则，它会生成一条指令来完成转换工作并将转换结果放置到临时变量 *temp* 中。这个临时变量将作为结果返回。函数 *widen* 的伪代码如图 6-26 所示，这里假设只有*integer*和*float* 两种类型。

```
Addr widen(Addr a, Type t, Type w)
    if ( t = w ) return a;
    else if ( t = integer and w = float ) {
        temp = new Temp();
        gen(temp '=' '(float)' a);
        return temp;
    }
    else error;
}
```

图 6-26 *widen* 函数的伪代码

图 6-27 中 $E \rightarrow E_1 + E_2$ 的语义动作说明了如何把类型转换加入到图 6-20 所示的翻译表达式的方案中。在这个语义动作中，如果 E_1 的类型不需要被转换成 E 的类型，那么临时变量 a_1 就是 $E_1.addr$。如果需要进行这样的转换，则 a_1 就是 *widen* 函数返回的一个新的临时变量。类似地，a_2 可能是 $E_2.addr$，也可能是一个新临时变量，用于存放转换后的 E_2 的值。如果两个变量都是整型或者都是浮点型，就不需要进行任何转换。我们会发现，将两个不同类型的值相加的唯一方法是把它们都转换成为第三种类型。

```
E → E1 + E2   { E.type = max(E1.type, E2.type);
                a1 = widen(E1.addr, E1.type, E.type);
                a2 = widen(E2.addr, E2.type, E.type);
                E.addr = new Temp();
                gen(E.addr '=' a1 '+' a2); }
```

图 6-27 在表达式求值中引入类型转换

6.5.3 函数和运算符的重载

依据符号所在的上下文不同，*被重载*(overloaded)的符号会有不同的含义。如果能够为一个名字的每次出现确定其唯一的含义，该名字的重载问题就得到了解决。在本节中，我们仅考虑那些只需要查看函数参数就能解决的函数重载。Java 中的重载即是如此。

例 6.13 根据其运算分量的类型，Java 中的 + 运算符既可以表示字符串的连接运算，也可以表示加法运算。用户自定义的函数同样可以重载，例如

```
void err() { … }
void err(String s) { … }
```

请注意，我们可以根据函数 err 的参数来确定选择该函数的哪一个版本。 □

以下是针对重载函数的类型综合规则：

$$\begin{aligned}&\textbf{if } f \text{ 可能的类型为 } s_i \to t_i\ (1 \leqslant i \leqslant n)\text{，其中，} s_i \neq s_j\ (i \neq j)\\&\textbf{and } x \text{ 的类型为 } s_k\ (1 \leqslant k \leqslant n)\\&\textbf{then } \text{表达式 } f(x) \text{ 的类型为 } t_k\end{aligned} \tag{6.10}$$

6.1.2 节中的值编码方法同样可以用于类型表达式，以便根据参数类型高效地解决重载问题。在表示类型表达式的一个 DAG 上，我们给每个结点赋予一个被称为值编码的整数序号。使用算法 6.3，我们可以构造出每个结点的范型，该范型由该结点的标号及其从左到右的子结点的值编码组成。一个函数的范型由其函数名和它的参数的类型组成。根据函数的参数类型解决重载的问题就等价于基于范型解决重载的问题。

仅仅通过查看一个函数的参数类型不一定能够解决重载问题。在 Ada 中，一个子表达式会有一组可能的类型，而不是只有一个确定的类型。它所在的上下文必须提供足够的信息来缩小可选范围，最终得到唯一的可选类型(见练习 6.5.2)。

6.5.4 类型推导和多态函数

类型推导常用于像 ML 这样的语言。ML 是一个强类型语言，但是它不要求名字在使用前先进行声明。类型推导保证了名字使用的一致性。

术语“多态”指的是任何可以在不同的参数类型上运行的代码片段。在本节中，我们考虑参数多态(parametric polymorphism)，这种多态通过参数和类型变量来刻划。我们使用图 6-28 中的 ML 程序作为一个贯穿本节的例子。该程序定义了一个函数 *length*。函数 *length* 的类型可以描述为：“对于任何类型 α，*length* 函数将元素类型为 α 的列表映射为整型”。

```
fun length(x) =
    if null(x) then 0 else length(tl(x)) + 1;
```

图 6-28 计算一个列表长度的 ML 程序

例 6.14 在图 6-28 中，关键字 *fun* 引出了一个函数定义，被定义的函数可以是递归的。这个程序片段定义了带有单个参数 x 的函数 *length*。这个函数的函数体包含了一个条件表达式。预定义的函数 *null* 测试一个列表是否为空。预定义函数 *tl*(tail 的缩写)移除列表中的第一个元素，然后返回列表的余下部分。

函数 *length* 确定一个列表 x 的长度，或者说 x 中元素的个数。列表中的所有元素必须具有相同的类型。不管列表元素是什么类型，都可以用 *length* 函数来求出这个列表的长度。在下面的表达式中，*length* 被应用到两种不同类型的列表中(列表元素用“[”和“]”括起来)：

$$length([\text{"sun"}, \text{"mon"}, \text{"tue"}]) + length([10, 9, 8, 7]) \tag{6.11}$$

字符串列表的长度为 3，整数列表的长度为 4，因此表达式(6.11)的值为 7。 □

使用符号 $\forall$(读作“对于任意类型”)以及类型构造算子 *list*，*length* 的类型可以写作：

$$\forall\, \alpha.\ list(\alpha) \to integer \tag{6.12}$$

符号 $\forall$ 是全称量词(wiversal quantifier)，它所作用的类型变量称为受限的(bound)。受限变量可以被任意地重命名，但是需要把这个变量的所有出现一起重命名。因此，类型表达式 $\forall\beta.\ list(\beta) \to integer$ 和式(6.12)等价。其中带有 $\forall$ 符号的类型表达式被称为“多态类型”。

在多态函数的各次应用中，函数的受限的类型变量可以表示不同的类型。在类型检查中，每次使用多态类型时，我们将受限变量替换为新的变量，并去掉相应的全称量词。

下一个例子对 *length* 类型进行了非正式的推导，推导过程中隐式地使用了公式(6.9)中的推导规则。这里再重复一下：

if $f(x)$是一个表达式

then 对某些 α 和 β，f 的类型为 $\alpha \to \beta$ 且 x 的类型为 α

例 6.15 图 6-29 中的抽象语法树表示图 6-28 中对 *length* 的定义。这棵树的根的标号为 **fun**，它表示函数定义。其他的非叶子结点可以看作是函数应用。标号为 + 的结点表示对两个子结点应用运算符 + 。类似的，标号为 **if** 的结点表示将运算符 **if** 应用于它的三个子结点组成的三元组上(对于类型检查，究竟是 **then** 分支还是 **else** 分支被求值并不是问题。它们不会被同时计算)。

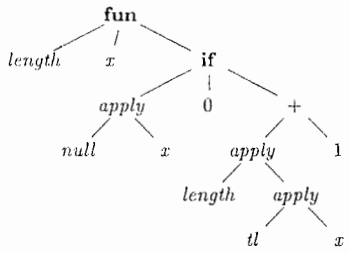


图 6-29 图 6-28 中的函数定义对应的抽象语法分析树

我们可以根据函数 *length* 的函数体推导出它的类型。从左到右考虑标号为 **if** 的结点的子结点。因为 *null* 要被应用在列表上，所以 x 必须是一个列表。我们使用变量 α 作为列表元素类型的占位符，也就是说，x 的类型为“α 的列表”。

如果 $null(x)$为真，则 $length(x)$为 0。因此，*length* 的类型一定是“从 α 的列表到整型的函数”。这个推导得到的类型和在 else 分支 $length(tl(x))+1$ 中对 *length* 的使用是一致的。 □

因为在类型表达式中可能出现变量，所以我们必须重新审视一下类型等价的概念。设想将类型为 $s \to s'$的 E_1 应用到类型为 t 的 E_2 上。我们不能简单地确定 s 和 t 是否等价，而是必须将这两种类型“合一”。非正式地讲，我们将确定是否可以将类型变量 s 和 t 替换为特定的类型表达式，从而使得 s 和 t 在结构上等价。

置换(substitution)是一个从类型变量到类型表达式的映射。我们把对类型表达式 t 中的变量应用置换 S 后得到的结果写作 $S(t)$，详细信息请参见“置换、实例和合一”部分。两个类型表达式 t_1 和 t_2 可以合一(unify)的条件是存在某个置换 S 使得 $S(t_1)=S(t_2)$。在实践中，我们感兴趣的是最一般化的合一置换，这种合一置换对表达式中的变量施加的约束最少。6.5.5 节给出了一个合一算法。

置换、实例和合一

如果 t 是一个类型表达式，且 S 是一个置换(即一个从类型变量到类型表达式的映射)，那么我们用 $S(t)$来表示将 t 中的每个类型变量 α 的所有出现替换为 $S(\alpha)$后得到的结果。$S(t)$被称为 t 的一个实例(instance)。例如，*list*(*integer*)是 *list*(α)的一个实例，因为它是将 *list*(α)中的 α 替换为 *integer* 后的结果。然而，请注意 *integer*$\to$*float* 不是 $\alpha \to \alpha$ 的实例，因为置换必须将 α 的所有出现替换为相同的类型表达式。

对于类型表达式 t_1 和 t_2，如果 $S(t_1)=S(t_2)$，那么置换 S 就是一个合一替换(unifier)。如果对于 t_1 和 t_2 的任何合一替换，比如说 S'，下面的条件成立：对于任意的 t，$S'(t)$是 $S(t)$的一个实例，那么我们就说 S 是 t_1 和 t_2 的最一般化的合一替换(most general unifier)。换句话说，S'对 t 施加的限制比 S 施加的限制更多。

算法 6.16 多态函数的类型推导。

输入：一个由一系列函数定义以及紧跟其后的待求值表达式组成的程序。一个表达式由多个函数应用和名字构成。这些名字具有预定义的多态类型。

输出：推导出的程序中名字的类型。

方法：为简单起见，我们只考虑一元函数。对于带有两个参数的函数 $f(x_1, x_2)$，我们可以将其类型表示为 $s_1 \times s_2 \rightarrow t$，其中 s_1 和 s_2 分别是 x_1 和 x_2 的类型，而 t 是函数 $f(x_1, x_2)$ 的结果类型。通过检查 s_1 是否和 a 的类型匹配，s_2 是否和 b 的类型匹配，就可以检查表达式 $f(a, b)$ 的类型。

检查输入序列中的函数定义和表达式。当一个函数在其后的表达式中被使用时，就使用推导得到的该函数的类型。

- 对一个函数定义 **fun** $\mathbf{id}_1(\mathbf{id}_2) = E$，创建一个新的类型变量 α 和 β。将函数 $\mathbf{id}_1$ 与类型 $\alpha \rightarrow \beta$ 相关联，参数 $\mathbf{id}_2$ 和类型 α 相关联。然后，推导出表达式 E 的类型。假设在对 E 进行类型推导之后，α 表示类型 s 而 β 表示类型 t。推导得到的函数 $\mathbf{id}_1$ 的类型就是 $s \rightarrow t$。使用 $\forall$ 量词来限制 $s \rightarrow t$ 中任何未受约束的类型变量。
- 对于函数应用 $E_1(E_2)$，推导出 E_1 和 E_2 的类型。因为 E_1 被用作一个函数，它的类型一定具有 $s \rightarrow s'$ 的形式（从技术上来说，E_1 的类型必须和 $\beta \rightarrow \gamma$ 合一，其中 β 和 γ 是新的类型变量）。假定推导得到的 E_2 的类型为 t。对 s 和 t 进行合一处理。如果合一失败，表达式返回类型错误，否则推导得到的 $E_1(E_2)$ 的类型为 s'。
- 对一个多态函数的每次出现，将它的类型表达式中的受限变量替换为互不相同的新变量，并移除 $\forall$ 量词。替换得到的类型表达式就是这个多态函数的本次出现所对应的推导类型。
- 对于第一次碰到的变量，引入一个新的类型变量来代表它的类型。 □

例 6.17 在图 6-30 中，我们为函数 *length* 推导出一个类型。图 6-29 中语法树的根表示一个函数定义，因此我们引入变量 β 和 γ，并将类型 $\beta \rightarrow \gamma$ 关联到函数 *length*，将 β 关联到 x。见图 6-30 的 1～2 行。

在根的右子结点上，我们把 **if** 看作一个应用到三元组上的多态函数，这个三元组包括一个布尔型变量以及两个分别代表 **then** 和 **else** 分支的表达式。函数 **if** 的类型是 $\forall \alpha.\ boolean \times \alpha \times \alpha \rightarrow \alpha$。

多态函数的每次应用可能作用于不同的类型，因此我们构造一个新的临时变量 α_i（i 取自 *if*），并移除 $\forall$，见图 6-30 中的第三行。函数 **if** 的左子结点的类型必须和 boolean 类型合一，其他两个子结点的类型必须和 α_i 合一。

行	表达式：类型	合一
1)	$length : \beta \rightarrow \gamma$	
2)	$x : \beta$	
3)	$\mathbf{if} : boolean \times \alpha_i \times \alpha_i \rightarrow \alpha_i$	
4)	$null : list(\alpha_n) \rightarrow boolean$	
5)	$null(x) : boolean$	$list(\alpha_n) = \beta$
6)	$0 : integer$	$\alpha_i = integer$
7)	$+ : integer \times integer \rightarrow integer$	
8)	$tl : list(\alpha_t) \rightarrow list(\alpha_t)$	
9)	$tl(x) : list(\alpha_t)$	$list(\alpha_t) = list(\alpha_n)$
10)	$length(tl(x)) : \gamma$	$\gamma = integer$
11)	$1 : integer$	
12)	$length(tl(x)) + 1 : integer$	
13)	$\mathbf{if}(\cdots) : integer$	

图 6-30 推导图 6-28 中的函数 *length* 的类型

预定义函数 *null* 的类型为 $\forall \alpha.\ list(\alpha) \rightarrow boolean$。我们使用一个新的类型变量 α_n（其中 n 表示 null）来替换受限变量 α，见第 4 行。因为 *null* 被应用于 x，我们推导出 x 的类型 β 必须和 $list(\alpha_n)$ 匹配，见第 5 行。

在 **if** 的第一个子结点上，$null(x)$ 的类型 boolean 和 **if** 函数预期的类型相匹配。在第二个子结

点上，类型 α_i 与 *integer* 进行合一，见第 6 行。

现在考虑子表达式 $length(tl(x))+1$。我们为 *tl* 类型中的约束变量 α 建立新的临时变量 α_t（其中 t 表示“tail”），见第 8 行。根据 $tl(x)$ 的应用，我们推导出 $list(\alpha_t)=\beta=list(\alpha_n)$，见第 9 行。

因为 $length(tl(x))$ 是 + 的一个运算分量，它的类型 γ 必须和 *integer* 合一，见第 10 行。可以推出 *length* 的类型为 $list(\alpha_n)\rightarrow integer$。在检查完这个函数定义之后，类型变量 α_n 仍然保留在 *length* 的类型中。因为没有对 α_n 作出任何假设，当使用该函数时 α_n 可以被替换为任何类型。因此，我们可以把它变成一个受限变量，并把 *length* 的类型写作：

$$\forall\,\alpha_n.\ list(\alpha_n)\rightarrow integer$$ □

6.5.5 一个合一算法

非正式地讲，合一就是判断能否通过将两个表达式 s 和 t 中的变量替换为某些表达式，使得 s 和 t 相同。测试表达式是否等价是合一的一个特殊情况。如果 s 和 t 中只有常量没有变量，则 s 和 t 合一当且仅当它们完全相同。本节中的合一算法可以处理含有环的图，因此它可以用于测试循环类型的结构等价性[⊖]。

我们将实现一种基于图论表示方法的合一算法，其中类型被表示成图的形式。类型变量用叶子结点表示，类型构造算子用内部结点表示。结点被分成若干的等价类。如果两个结点在同一个等价类中，那么它们代表的类型表达式就必须合一。因此，同一个等价类中的内部结点必须具有同样的类型构造算子，且它们的对应子结点必须等价。

例 6.18 考虑下列两个类型表达式

$$((\alpha_1\rightarrow\alpha_2)\times list(\alpha_3))\rightarrow list(\alpha_2)$$

$$((\alpha_3\rightarrow\alpha_4)\times list(\alpha_3))\rightarrow\alpha_5$$

下列的置换 S 是这两个表达式的最一般化的合一替换：

x	$S(x)$
α_1	α_1
α_2	α_2
α_3	α_1
α_4	α_2
α_5	$list(\alpha_2)$

这个置换将上述两个类型表达式映射成如下的表达式

$$((\alpha_1\rightarrow\alpha_2)\times list(\alpha_1))\rightarrow list(\alpha_2)$$

这两个表达式被表示为图 6-31 中标号为→：1 的两个结点。结点上的整数编号指明了在编号为 1 的结点被合一后，各个结点所属的等价类的编号。 □

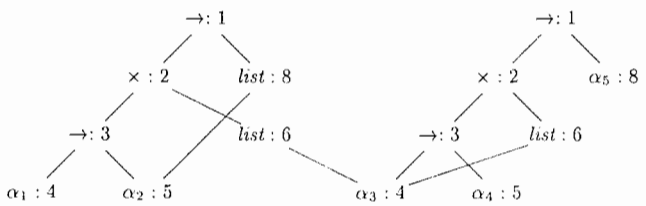


图 6-31 合一后的等价类

⊖ 在有些应用中，对一个变量和一个包含该变量的表达式进行合一是错误的。算法 6.19 允许这种替换。

算法 6.19 类型图中的一对结点的合一处理。

输入：一个表示类型的图，以及需要进行合一处理的结点对 m 和 n。

输出：如果结点 m 和 n 表示的表达式可以合一，返回布尔值 true。反之，返回 false。

方法：结点用一个记录实现，记录中的字段用于存放一个二元运算符和分别指向其左右子结点的指针。字段 *set* 用于保存等价结点的集合。每个等价类都有一个结点被选作这个类的唯一代表，它的 *set* 字段包含一个空指针。等价类中其他结点的 *set* 字段（可能通过该集合中的其他结点间接地）指向该等价类的代表结点。在初始时刻，每个结点 n 自身组成一个等价类，n 是它自己的代表结点。

如图 6-32 所示的合一算法在结点上进行如下两种操作：

- *find*(n)返回当前包含结点 n 的等价类的代表结点。
- *union*(m, n)将包含结点 m 和 n 的等价类合并。如果 m 和 n 所对应的等价类的代表结点中有一个是非变量的结点，则 *union* 将这个非变量结点作为合并后的等价类的代表结点；否则，*union* 把任意一个原代表结点作为新的代表结点。这种在 *union* 的规约中的非对称性非常重要，因为如果一个等价类对应于一个带有类型构造算子的类型表达式或基本类型，我们就不能用一个变量作为该等价类的代表。否则，两个不等价的表达式可能会通过该变量被合一。

```
boolean unify(Node m, Node n) {
    s = find(m); t = find(n);
    if ( s = t ) return true;
    else if ( 结点 s 和 t 表示相同的基本类型 ) return true;
    else if ( s 是一个带有子结点 s1 和 s2 的 op-结点 and
              t 是一个带有子结点 t1 和 t2 的 op-结点 ) {
        union(s, t);
        return unify(s1, t1) and unify(s2, t2);
    }
    else if ( s 或者 t 表示一个变量 ) {
        union(s, t);
        return true;
    }
    else return false;
}
```

图 6-32 合一算法

集合的 *union* 操作的实现很简单，只需要改变一个等价类的代表结点的 *set* 字段，使之指向另一个等价类的代表结点即可。为了找到一个结点所属的等价类，我们沿着各个结点的 *set* 字段中的指针前进，直到到达代表结点（即 *set* 字段指针为空指针的结点）为止。

请注意，图 6-32 中的算法分别使用 s = *find*(m)和 t = *find*(n)，而不是直接使用 m 和 n。如果 m 和 n 在同一个等价类中，那么代表结点 s 和 t 相等。如果 s 和 t 表示相同的基本类型，则调用 *unify*(m, n)返回 *true*。如果 s 和 t 都是代表某个二目类型构造算子的内部结点，那么我们尝试合并它们的等价类，并递归地检查它们的各个子结点是否等价。因为首先进行合并操作，我们在递归检查子结点之前减少了等价类的个数，因此算法终止。

将一个变量置换为一个表达式的实现方法如下：把代表该变量的叶子结点加入到代表该表达式的结点所在的等价类中。假设 m 或 n 表示一个变量的叶子结点，同时假设这个结点已经放入满足下面条件的等价类中，即这个等价类中的一个结点代表的表达式或者带有一个类型构造算子，或者是一个基本类型。那么，*find* 将会返回一个反映该类型构造算子或基本类型的代表结

点，使一个变量不会和两个不同的表达式合一。 □

例 6.20 假设例 6.18 中的两个表达式可以用图 6-33 中的两个初始图表示，图中的每个结点所在的等价类仅仅包含该结点。当应用算法 6.19 来计算 *unify*(1, 9)时，注意到结点 1 和 9 表示同一个运算符。因此将结点 1 和 9 合并成同一个等价类，并调用 *unify*(2, 10)和 *unify*(8, 14)。执行 *unify*(1, 9)得到的结果就是前面在图 6-31 中显示的图。 □

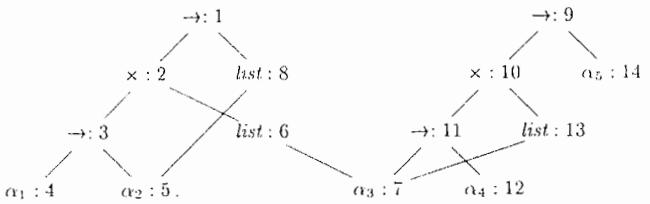


图 6-33 初始图，其中的每个结点在只包含该结点自身的等价类中

如果算法 6.19 返回 true，我们可以按照如下方法构造出一个置换 S 作为合一替换。对于每个变量 α，*find*(α)给出 α 的等价类的代表结点 n。n 所表示的表达式为 $S(\alpha)$。例如，在图 6-31 中，我们看到 α_3 的代表结点为 4，这个结点表示 α_1。结点 8 是 α_5 的代表结点，这个结点表示 *list*(α_2)。置换 S 的结果如例 6.18 所示。

6.5.6 6.5 节的练习

练习 6.5.1：假定图 6-26 中的函数 *widen* 可以处理图 6-25a 的层次结构中的所有类型，翻译下列表达式。假定 c 和 d 是字符型，s 和 t 是短整型，i 和 j 为整型，x 是浮点型。

1) `x = s + c`

2) `i = s + c`

3) `x = (s + c) * (t + d)`

练习 6.5.2：像 Ada 中那样，我们假设每个表达式必须具有唯一的类型，但是我们根据一个子表达式本身只能推导出一个可能类型的集合。也就是说，将函数 E_1 应用于参数 E_2(其文法产生式为 $E \rightarrow E_1(E_2)$)有如下规则：

$$E.type = \{t \mid \text{对 } E_2.type \text{ 中的某个 } s,\ s \rightarrow t \text{ 在 } E_1.type \text{ 中}\}$$

描述一个可以确定每个子表达式的唯一类型的语法制导定义(SDD)。它首先使用属性 *type*，按照自底向上的方式综合得到一个可能类型的集合。在确定了整个表达式的唯一类型之后，自顶向下地确定属性 *unique* 的值，这个属性表示各个子表达式的类型。

6.6 控制流

if-else 语句、while 语句这类语句的翻译和对布尔表达式的翻译是结合在一起的。在程序设计语言中，布尔表达式经常用来：

1) *改变控制流*。布尔表达式被用作语句中改变控制流的条件表达式。这些布尔表达式的值由程序到达的某个位置隐含地指出。例如，在 **if**(E) S 中，如果运行到语句 S，就意味着表达式 E 的取值为真。

2) *计算逻辑值*。一个布尔表达式的值可以表示 *true* 或 *false*。这样的布尔表达式也可以像算术表达式一样，使用带有逻辑运算符的三地址指令进行求值。

布尔表达式的使用意图要根据其语法上下文来确定。例如，跟在关键字 **if** 后面的布尔表达式用来改变控制流，而一个赋值语句右部的表达式用来表示一个逻辑值。有多种方式可以描述

这样的上下文：我们可以使用两个不同的非终结符号，也可以使用继承属性，还可以在语法分析过程中设置一个标记。此外，我们还可以建立一棵语法分析树并调用不同的过程来处理布尔表达式的两种不同的使用。

本节将介绍用于改变控制流的布尔表达式。更清楚地说，我们为此引入一个新的非终结符号 B。在 6.6.6 节中，我们将考虑编译器如何使得布尔表达式表示逻辑值。

6.6.1 布尔表达式

布尔表达式是将由作用于布尔变量或关系表达式的布尔运算符而构成的。我们使用 C 语言的方法，用 &&、||、! 分别表示 AND、OR、NOT 运算符。关系表达式的形式为 E_1 **rel** E_2。其中，E_1 和 E_2 为算术表达式。在本节中，我们考虑的是由如下文法生成的布尔表达式：

$$B \rightarrow B \ \| \ B \mid B \ \&\& \ B \mid ! \ B \mid (B) \mid E \ \textbf{rel} \ E \mid \textbf{true} \mid \textbf{false}$$

我们通过属性 **rel**. *op* 来指明 **rel** 究竟表示 6 种比较运算符 <、<=、=、!=、> 和 >= 中的哪一种。按照惯例，假设 || 和 && 是左结合的，|| 的优先级最低，其次为 &&，再其次为!。

给定表达式 $B_1 \| B_2$，如果我们已经确定 B_1 为真，那么不用再计算 B_2 就可以断定整个表达式为真。同样的，给定 $B_1 \&\& B_2$，如果 B_1 为假，则整个表达式为假。

程序设计语言的语义定义决定了是否需要对一个布尔表达式的各个部分都进行求值。如果语言的定义允许(或要求)不对布尔表达式的某个部分求值，那么编译器就可以优化布尔表达式的求值过程，只要已经求值的部分足以确定整个表达式值就可以了。因此，在表达式 $B_1 \| B_2$ 中，B_1 和 B_2 都不一定要完全地求值。如果 B_1 或 B_2 是具有副作用的表达式(比如它包含了改变一个全局变量的函数)，那么这么做就可能会得到意料之外的结果。

6.6.2 短路代码

在短路(跳转)代码中，布尔运算符 &&、|| 和! 被翻译成跳转指令。运算符本身不出现在代码中，布尔表达式的值是通过代码序列中的位置来表示的。

例 6.21 语句

```
if (x<100 || x>200 && x!=y) x=0;
```

可以被翻译成图 6-34 所示的代码。在这个翻译中，如果程序的控制流到达 L_2，就表示这个布尔表达式为真。如果表达式为假，则程序控制流将跳过 L_2 和赋值语句 $x=0$，直接转到 L_1。 □

```
      if x < 100 goto L2
      ifFalse x > 200 goto L1
      ifFalse x != y goto L1
L2:   x = 0
L1:
```

图 6-34 跳转代码

6.6.3 控制流语句

现在我们考虑在按下列文法生成的语句的上下文中，如何把布尔表达式翻译成为三地址代码。

$$S \rightarrow \textbf{if} \ (B) \ S_1$$

$$S \rightarrow \textbf{if} \ (B) \ S_1 \ \textbf{else} \ S_2$$

$$S \rightarrow \textbf{while} \ (B) \ S_1$$

在这些产生式中，非终结符号 B 表示一个布尔表达式，非终结符号 S 表示一个语句。

这个文法将例 5.19 中介绍的关于 while 表达式的连续使用的例子进行了推广。和那个例子一样，B 和 S 有综合属性 *code*，该属性给出了翻译得到的三地址指令。为简单起见，我们使用语法制导定义来构造得到翻译结果 B. *code* 和 S. *code*，结果值是字符串。定义了 *code* 属性的语义规则还可以按照下面的方法实现：首先构造语法树，并在遍历树的过程中产生目标代码。这些规则还可以通过 5.5 节中列出的任何方法来实现。

如图 6-35a 所示，对 **if**(B) S_1 的翻译结果中包含了 B. *code*，其后是 S_1. *code*。B. *code* 中存在基于 B 值的跳转。如果 B 为真，控制流转向 S_1. *code* 的第一条指令；如果 B 为假，控制流立即转向

紧跟在 S_1. *code* 之后的指令。

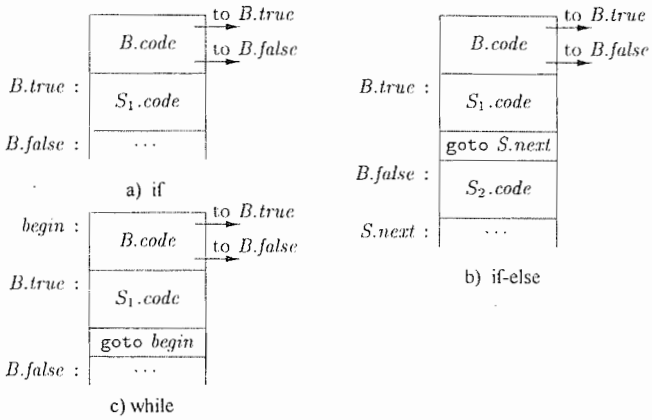


图 6-35 if、if-else、while 语句的代码

B. code 和 *S. code* 中的跳转标号使用继承属性来处理。我们将布尔表达式 B 和两个标号：*B. true* 和 *B. false* 相关联。当 B 为真时控制流转到 *B. true*；当 B 为假时控制流转到 *B. false*。我们将语句 S 和继承属性 *S. next* 相关联，这个属性表示紧跟在 S 代码之后的指令的标号。在某些情况下，紧跟在 *S. code* 之后的指令是一个跳转到某个标号 L 的跳转指令。使用 *S. next* 可以避免在 *S. code* 中出现这样的一个跳转指令，它的目标又是一个以 L 为目标的跳转指令。

图 6-36 和图 6-37 给出的语法制导定义可以为在 if、if-else 及 while 语句的上下文中的布尔表达式生成三地址代码。

产生式	语义规则
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

图 6-36 控制流语句的语法制导定义

我们假定每次调用 *newlabel*( )都会产生一个新的标号，并假设 *label*(*L*)将标号 *L* 附加到即将生成的下一条三地址指令上[⊖]。

一个程序包含一条由产生式 $P \rightarrow S$ 生成的语句。和这个产生式关联的语义规则将 *S. next* 初始化为一个新标号。*P. code* 包含 *S. code*，*S. code* 之后是新标号 *S. next*。产生式 $S \rightarrow$ **assign** 中的词法单元 **assign** 是一个表示赋值语句的占位符。赋值语句的翻译和 6.4 节中讨论的方法相同。在这里对控制流的讨论中，*S. code* 就是 **assign**. *code*。

在翻译 $S \rightarrow$ **if** (B) S_1 时，图 6-36 中的语义规则创建一个新的标号 *B. true*，并将其关联到为语句 S_1 生成的第一条三地址指令中，如图 6-35a 所示。因此，*B* 的代码中跳转到 *B. true* 的指令将跳转到语句 S_1 对应的代码处。不仅如此，通过将 *B. false* 设为 *S. next*，我们保证了当 *B* 的值为假时，控制流将跳过 S_1 的代码。

在翻译 if-else 语句 $S \rightarrow$ **if** (B) S_1 **else** S_2 时，布尔表达式 *B* 的代码中有一些向外跳转的指令，它们在 *B* 为真时跳转到 S_1 的代码的第一条指令；在 *B* 为假时跳转到 S_2 的代码的第一条指令，如图 6-35b 所示。然后，控制流从 S_1 或 S_2 转到紧跟在 *S* 的代码之后的三地址指令——该指令的标号由继承属性 *S. next* 指定。在 S_1 的代码之后有一条 *goto S. next* 指令，使得控制流越过 S_2 的代码。S_2 的代码之后不需要 *goto* 语句，因为 S_2. *next* 就是 *S. next*。

如图 6-35c 所示，$S \rightarrow$ **while**(B) S_1 的代码由 *B. code* 和 S_1. *code* 组成。我们使用一个局部变量 *begin* 来存放附加在这个 while 语句的第一条指令上的标号。这个 while 语句的第一条指令也是 *B* 的第一条指令。我们在这里使用变量而不是属性，是因为 *begin* 对于这个产生式的语义规则而言是局部的。继承属性 *S. next* 标记了当 *B* 为假时控制流必须转向的标号。因此，*B. false* 被设置为 *S. next*。在 S_1 的第一条指令上附加了一个新标号 *B. true*。*B* 的指令中的跳转指令在 *B* 为真时跳转到这个标号。我们在 S_1 的代码之后放置了一条指令 `goto` *begin*，它跳回到布尔表达式的代码的开始处。请注意，S_1. *next* 被设置为标号 *begin*，因此从 S_1. *code* 中跳出的指令可以直接跳转到 *begin*。

$S \rightarrow S_1 S_2$ 的代码包含了 S_1 的代码，然后是 S_2 的代码。相应的语义规则主要处理标号。S_1 的代码之后的第一条指令就是 S_2 的代码的起始指令。紧跟在 S_2 的代码之后的指令也是跟在 *S* 的代码之后的指令。

我们将在 6.7 节中进一步讨论控制流语句的翻译。在那里我们将使用另一种被称为回填的方法，它可以在一次扫描中生成各个语句的代码。

6.6.4 布尔表达式的控制流翻译

图 6-37 中针对布尔表达式的语义规则是图 6-36 中语句的语义规则的一个补充。如图 6-35 中的代码布局方案所示，一个布尔表达式 *B* 被翻译为一个三地址指令，它将使用条件或无条件跳转指令来对 *B* 求值。这些跳转指令的目标是两个标号之一：当 *B* 为真时是 *B. true*；当 *B* 为假时是 *B. false*。

图 6-37 中的第四个产生式，即 $B \rightarrow E_1$ **rel** E_2，直接被翻译成三地址比较指令，跳转到正确的位置。例如，$a < b$ 被翻译成：

```
if a < b goto B.true
goto B.false
```

⊖ 如果严格地按照上面的语义规则来实现，这些语义规则将产生很多标号，并可能在一个三地址指令上附加多个标号。6.7 节中介绍的回填技术只在必要的时候创建标号。处理这个问题的另一种方法是在后续的优化步骤中消除不必要的标号。

产生式	语义规则
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \,\&\&\, B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow$ **true**	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow$ **false**	$B.code = gen('\texttt{goto}'\ B.false)$

图 6-37 为布尔表达式生成三地址代码

B 的其余产生式按照下面的方法翻译：

1）假定 B 形如 $B_1 \parallel B_2$。如果 B_1 为真，那么我们立刻知道 B 本身也为真，因此 $B_1.true$ 和 $B.true$ 相同。如果 B_1 为假，那么就必须对 B_2 求值，因此我们将 $B_1.false$ 设置为 B_2 的代码的第一条指令的标号。B_2 的真假出口分别等于 B 的真假出口。

2）$B_1 \&\& B_1$ 的翻译方法类似于 1。

3）不需要为 $B \rightarrow ! B_1$ 产生新的代码，只需要将 B 中的真假出口对换，就可分别得到 B_1 的真假出口。

4）将常量 **true** 和 **false** 分别翻译成目标为 $B.true$ 和 $B.false$ 的跳转指令。

例 6.22 重新考虑例 6.21 中的下列语句：

```
if (x<100 || x>200 && x!=y) x=0;        (6.13)
```

使用图 6-36 和图 6-37 中的语法制导定义，我们可以得到图 6-38 中的代码。

```
      if x < 100 goto L2
      goto L3
L3:   if x > 200 goto L4
      goto L1
L4:   if x != y goto L2
      goto L1
L2:   x = 0
L1:
```

图 6-38 一个简单的 if 语句的控制流翻译结果

语句(6.13)是图 6-36 中的产生式 $P \rightarrow S$ 生成的一个程序。这个产生式的语义规则生成了 S 的代码之后的第一条指令的新标号 L_1。语句 S 的形式为 **if**(B) S_1，其中 S_1 是 $x=0$。因此，图 6-36 中的规则生成了一个新标号 L_2，并将它附加到 $S_1.code$ 的第一条(在这个例子中也是唯一的)指令，即 $x=0$ 处。

因为 $\parallel$ 的优先级低于 &&，所以式(6.13)中的布尔表达式的形式为 $B_1 \parallel B_2$，其中 B_1 是 $x<100$。按照图 6-37 中的规则，$B_1.true$ 是 L_2，即语句 x = 0 的标号；$B_1.false$ 是一个新的标号 L_3，它附加在 B_2 的代码的第一条指令上。

值得注意的是，生成的代码不是最优的，因为这个翻译结果比例 6.21 中的代码多三条(goto)指令。指令 goto L_3 是冗余的，因为 L_3 恰巧就是下一条指令的标号。如果像例 6.21 中那样使用

ifFalse 指令，而不使用 if 指令，那么两条 goto L_1 指令也可以被消除。 □

6.6.5 避免生成冗余的 goto 指令

在例 6.22 中，比较表达式 $x > 200$ 被翻译成如下代码片段：

```
    if x > 200 goto L4
    goto L1
L1: ···
```

可以将上面的指令替换为如下指令：

```
    ifFalse x > 200 goto L1
L4: ···
```

ifFlase 指令利用了控制流在指令序列中会从一个指令自然流动到下一个指令的性质，因此当 $x > 200$ 时，控制流直接“穿越”到标号 L_4，从而减少了一个跳转指令。

在图 6-35 中所示的 if 和 while 语句的代码布局中，S_1 的代码紧跟在布尔表达式 B 的代码之后。通过使用一个特殊标号“fall”（即“不要生成任何跳转指令”），我们可以修改图 6-36 和图 6-37中的语义规则，支持控制流从 B 的代码直接穿越到 S_1 的代码。图 6-36 中的产生式 $S \rightarrow \textbf{if}(B)S_1$ 的新语义规则将 $B.true$ 设为 *fall*：

$$
\begin{aligned}
&B.true = fall\\
&B.false = S_1.next = S.next\\
&S.code = B.code \,||\, S_1.code
\end{aligned}
$$

类似地，if-else 和 while 语句的规则也将 $B.true$ 设为 *fall*。

现在我们将修改布尔表达式的语义规则，使之尽可能地允许控制流穿越。在 $B.true$ 和 $B.false$ 都是显式的标号时，也就是说它们都不等于 *fall* 时，图 6-39 中的 $B \rightarrow E_1$ **rel** E_2 的新规则将产生两条指令（和图 6-37 一样）。否则，如果 $B.true$ 是显式的标号，那么 $B.false$ 一定是 *fall*，因此它们产生一条 if 指令，使得当条件为假时控制流穿越到下一条指令。反过来，如果 $B.false$ 是显式的标号，那么它们产生一条 ifFalse 指令。在其余情况中，$B.true$ 和 $B.false$ 都是 *fall*，因此不产生任何跳转指令[⊖]。

$$
\begin{aligned}
&test = E_1.addr\ \textbf{rel}.op\ E_2.addr\\
&\quad s = \textbf{if}\ B.true \neq fall\ \textbf{and}\ B.false \neq fall\ \textbf{then}\\
&\qquad\quad gen(\text{'if'}\ test\ \text{'goto'}\ B.true) \,||\, gen(\text{'goto'}\ B.false)\\
&\qquad \textbf{else if}\ B.true \neq fall\ \textbf{then}\ gen(\text{'if'}\ test\ \text{'goto'}\ B.true)\\
&\qquad \textbf{else if}\ B.false \neq fall\ \textbf{then}\ gen(\text{'ifFalse'}\ test\ \text{'goto'}\ B.false)\\
&\qquad \textbf{else}\ ''\\
&B.code = E_1.code \,||\, E_2.code \,||\, s
\end{aligned}
$$

图 6-39 $B \rightarrow E_1$ **rel** E_2 的语义规则

在图 6-40 中显示的 $B \rightarrow B_1 \parallel B_2$ 的新规则中，请注意 B 的 *fall* 标号和 B_1 的 *fall* 标号具有不同的含义。假定 $B.true$ 为 *fall*，即如果 B 为真时控制流穿越 B。虽然当 B_1 为真时 B 的值必然为真，但 $B_1.true$ 必须保证控制流跳过 B_2 的代码，直接到达 B 之后的下一条指令。

$$
\begin{aligned}
&B_1.true = \textbf{if}\ B.true \neq fall\ \textbf{then}\ B.true\ \textbf{else}\ newlabel()\\
&B_1.false = fall\\
&B_2.true = B.true\\
&B_2.false = B.false\\
&B.code = \textbf{if}\ B.true \neq fall\ \textbf{then}\ B_1.code \,||\, B_2.code\\
&\qquad\quad \textbf{else}\ B_1.code \,||\, B_2.code \,||\, label(B_1.true)
\end{aligned}
$$

图 6-40 $B \rightarrow B_1 \parallel B_2$ 的语义规则

另一方面，如果 B_1 的值为假，B 的真假值就由 B_2 的值决定。因此，图 6-40 中的规则保证

⊖ 在 C 和 Java 中，表达式中可能包含赋值语句，因此即使 $B.true$ 和 $B.false$ 都为 *fall*，也必须为子表达式 E_1 和 E_2 生成代码。如果必要，无用代码可以在优化阶段被清除。

$B_1.false$ 对应于控制流穿越 B_1 直接到达 B_2 的代码的情况。

$B \rightarrow B_1 \&\& B_2$ 的语义规则和图 6-40 中的语义规则类似，我们将其留作练习。

例 6.23 使用了特殊标号 *fall* 的语义规则将例 6.21 中的程序(6.13)

```
if (x<100 || x>200 && x!=y) x=0;
```

翻译成图 6-41 所示的代码。

```
      if x < 100 goto L2
      ifFalse x > 200 goto L1
      ifFalse x != y goto L1
L2:   x = 0
L1:
```

图 6-41 使用控制流穿越技术翻译的 if 语句

和例 6.22 一样，产生式 $P \rightarrow S$ 的语义规则创建标号 L_1。和例 6.22 不同的是，当应用 $B \rightarrow B_1 \| B_2$ 的语义规则时，继承属性 $B.true$ 是 $fall$（$B.false$ 为 L_1）。图 6-40 中的规则创建一个新标号 L_2，使得当 B_1 为真时有一个跳转指令可以跳过 B_2 的代码。因此，$B_1.true$ 为 L_2 而 $B_1.false$ 为 $fall$，因为 B_1 为假时必须计算 B_2 的值。

当开始处理生成了表达式 $x<100$ 的产生式 $B \rightarrow E_1$ **rel** E_2 时，B. true = L_2 且 B. false = fall。图 6-39 中的规则使用这些继承到的标号生成了一条指令 `if x<100 goto` L_2。 □

6.6.6 布尔值和跳转代码

本节讨论的重点是用于改变语句中控制流的布尔表达式。一个布尔表达式的目的可能就是要求出它的值，如 `x=true;` 或 `x=a<b;` 的语句中的布尔表达式就是这样。

处理布尔表达式的这两种角色的一种简单思路是首先建立表达式的抽象语法树，可以使用下面的两种方法之一：

1）*使用两趟处理的方法*。为输入构造出完整的抽象语法树，然后以深度优先顺序遍历这棵抽象语法树，依据语义规则的描述计算得到翻译结果。

2）*对语句进行一趟处理，但对表达式进行两趟处理*。使用这种方法时，我们将首先翻译语句 **while**(E) S_1 中的 E，然后再处理 S_1。然而，要对 E 进行翻译，需要首先建立它的抽象语法树，然后再遍历它。

在下列文法中，用单个非终结符号 E 来代表表达式：

$$S \rightarrow \textbf{id} = E; \mid \textbf{if}\ (E)\ S \mid \textbf{while}\ (E)\ S \mid S\ S$$

$$E \rightarrow E \| E \mid E \&\& E \mid E\ \textbf{rel}\ E \mid E + E \mid (E) \mid \textbf{id} \mid \textbf{true} \mid \textbf{false}$$

非终结符号 E 支配了 $S \rightarrow$ **while** $(E)\ S_1$ 的控制流。同一个非终结符号 E 在 $S \rightarrow$ **id** $= E$ 和 $E \rightarrow E + E$ 中则表示一个值。

我们可以使用不同的代码生成函数处理表达式的这两种角色。假定属性 $E.n$ 表示对应于表达式 E 的抽象语法树结点，并且抽象语法树中的结点都是对象。令方法 *jump* 产生一个表达式结点的跳转代码，并令方法 *rvalue* 产生计算结点的值的代码，该代码还把得到的值存储在一个临时变量中。

对于出现在 $S \rightarrow$ **while** $(E)\ S_1$ 中的 E，在结点 $E.n$ 上调用方法 *jump*。方法 *jump* 的实现是基于图 6-37 给出的关于布尔表达式的语义规则。确切地说，跳转代码是通过调用 $E.n.jump(t, f)$ 生成的，其中 t 是指向 $S_1.code$ 的第一条指令的新标号，而 f 就是标号 $S.next$。

对于出现在 $S \rightarrow$ **id** $= E;$ 中的 E，在结点 $E.n$ 上调用方法 *rvalue*。如果 E 形如 $E_1 + E_2$，方法调用 $E.n.rvalue()$ 按照 6.4 节中讨论的方法生成代码。如果 E 形如 $E_1 \&\& E_2$，我们首先为 E 生成跳转代码，然后在跳转代码的真假出口分别将 true 和 false 赋给一个新的临时变量 t。

例如，赋值语句 `x=a<b && c<d` 可以用图 6-42 中的代码来实现。

```
      ifFalse a < b goto L1
      ifFalse c < d goto L1
      t = true
      goto L2
L1:   t = false
L2:   x = t
```

图 6-42 通过计算一个临时变量的值来翻译一个布尔类型的赋值语句

6.6.7 6.6 节的练习

练习 6.6.1：在图 6-36 的语法制导定义中添加处理下列控制流构造的规则：

1）一个 repeat 语句，**repeat** S **while** B。

！2）一个 for 循环语句，**for** $(S_1; B; S_2)$ S_3。

练习 6.6.2：现代计算机试图在同一时刻执行多条指令，其中包括各种分支指令。因此，当计算机投机性地预先执行某个分支，但实际控制流却进入另一分支时（此时所有预先执行的投机工作将被抛弃），付出的代价是很大的。因此我们希望尽可能地减少分支数量。请注意，在图 6-35c 中while 循环语句的实现中，每个迭代有两个分支：一个是从条件 B 进入到循环体中，另一个分支跳转回 B 的代码。基于尽量减少分支的考虑，我们通常更倾向于将 **while**(B) S 当作 **if** (B) {**repeat** S **until** !(B)} 来实现。给出这种翻译方法的代码布局，并修改图 6-36 中 while 循环语句的规则。

！**练习 6.6.3**：假设 C 中存在一个异或运算（当且仅当两个分量恰有一个为真时，表达式为真）。按照图 6-37 的风格写出这个运算符的代码生成规则。

练习 6.6.4：使用 6.6.5 节中介绍的避免 goto 语句的翻译方案，翻译下列表达式：

1) `if (a==b && c==d || e==f) x == 1;`

2) `if (a==b || c==d || e==f) x == 1;`

3) `if (a==b && c==d && e==f) x == 1;`

练习 6.6.5：基于图 6-36 和图 6-37 中给出的语法制导定义，给出一个翻译方案。

练习 6.6.6：使用类似于图 6-39 和图 6-40 中的规则，修改图 6-36 和图 6-37 的语义规则，使之允许控制流穿越。

！**练习 6.6.7**：练习 6.6.6 中的语句的语义规则产生了一些不必要的标号。修改图 6-36 中语句的规则，使之只创建必要的标号。你可以使用特殊标号 *deferred* 来表示还没有创建一个标号。你的语义规则必须能够生成类似于例 6.21 的代码。

！！**练习 6.6.8**：6.6.5 节中讨论了如何使用穿越代码来尽可能减少生成的中间代码中跳转指令的数目。然而，它并没有充分考虑将一个条件替换为它的补的方法，例如将 `if a < b goto` L_1 `; goto` L_2 `;` 替换为 `if a >= b goto` L_2 `; goto` L_1。给出一个语法制导定义，它在需要时可以利用这种替换方法。

6.7 回填

为布尔表达式和控制流语句生成目标代码时，关键问题之一是将一个跳转指令和该指令的目标匹配起来。例如，对 `if` (B) S 中的布尔表达式 B 的翻译结果中包含一条跳转指令。当 B 为假时，该指令将跳转到紧跟在 S 的代码之后的指令处。在一趟式的翻译中，B 必须在处理 S 之前就翻译完毕。那么跳过 S 的 `goto` 指令的目标是什么呢？在 6.6 节中，我们解决这个问题的方法是将标号作为继承属性传递到生成相关跳转指令的地方。但是，这样的做法要求再进行一趟处理，将标号和具体地址绑定起来。

本节将介绍一种被称为回填（backpatching）的补充性技术，它把一个由跳转指令组成的列表以综合属性的形式进行传递。明确地讲，生成一个跳转指令时暂时不指定该跳转指令的目标。这样的指令都被放入一个由跳转指令组成的列表中。等到能够确定正确的目标标号时才去填充这些指令的目标标号。同一个列表中的所有跳转指令具有相同的目标标号。

6.7.1 使用回填技术的一趟式目标代码生成

回填技术可以用来在一趟扫描中完成对布尔表达式或控制流语句的目标代码生成。我们生

成的目标代码的形式和 6.6 节中的代码的形式相同，但是处理标号的方法不同。

在本节中，非终结符号 B 的综合属性 *truelist* 和 *falselist* 将用来管理布尔表达式的跳转代码中的标号。特别的，$B.truelist$ 将是一个包含跳转或条件跳转指令的列表，我们必须向这些指令中插入适当的标号，也就是当 B 为真时控制流应该转向的标号。类似地，$B.falselist$ 也是一个包含跳转指令的列表，这些指令最终获得的标号就是当 B 为假时控制流应该转向的标号。在生成 B 的代码时，跳转到真或假出口的跳转指令是不完整的，标号字段尚未填写。这些不完整的跳转指令被保存在 $B.truelist$ 和 $B.falselist$ 所指的列表中。类似地，语句 S 的综合属性 $S.nextlist$ 也是一个跳转指令列表，这些指令应该跳转到紧跟在 S 的代码之后的指令。

更明确地讲，我们将生成的指令放入一个指令数组中，而标号就是这个数组的下标。为了处理跳转指令的列表，我们使用下面三个函数：

1）$makelist(i)$ 创建一个只包含 i 的列表。这里 i 是指令数组的下标。函数 *makelist* 返回一个指向新创建的列表的指针。

2）$merge(p_1, p_2)$ 将 p_1 和 p_2 指向的列表进行合并，它返回的指针指向合并后的列表。

3）$backpatch(p, i)$ 将 i 作为目标标号插入到 p 所指列表中的各指令中。

6.7.2 布尔表达式的回填

现在我们构造一个可以在自底向上语法分析过程中为布尔表达式生成目标代码的翻译方案。这个文法中有一个标记非终结符号 M。它引发的语义动作在适当的时刻获取将要生成的下一条指令的下标。该文法如下：

$B \rightarrow B_1 \,||\, M\ B_2 \mid B_1\ \&\&\ M\ B_2 \mid !\ B_1 \mid (\ B_1\ ) \mid E_1\ \mathbf{rel}\ E_2 \mid \mathbf{true} \mid \mathbf{false}$

$M \rightarrow \epsilon$

翻译方案如图 6-43 所示。

1)	$B \rightarrow B_1 \,\|\|\, M\ B_2$	{ $backpatch(B_1.falselist, M.instr)$; $B.truelist = merge(B_1.truelist, B_2.truelist)$; $B.falselist = B_2.falselist$; }
2)	$B \rightarrow B_1\ \&\&\ M\ B_2$	{ $backpatch(B_1.truelist, M.instr)$; $B.truelist = B_2.truelist$; $B.falselist = merge(B_1.falselist, B_2.falselist)$; }
3)	$B \rightarrow !\ B_1$	{ $B.truelist = B_1.falselist$; $B.falselist = B_1.truelist$; }
4)	$B \rightarrow (\ B_1\ )$	{ $B.truelist = B_1.truelist$; $B.falselist = B_1.falselist$; }
5)	$B \rightarrow E_1\ \mathbf{rel}\ E_2$	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr+1)$; $gen('\texttt{if}'\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ '\texttt{goto}\ \_')$; $gen('\texttt{goto}\ \_')$; }
6)	$B \rightarrow \mathbf{true}$	{ $B.truelist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
7)	$B \rightarrow \mathbf{false}$	{ $B.falselist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
8)	$M \rightarrow \epsilon$	{ $M.instr = nextinstr$; }

图 6-43 布尔表达式的翻译方案

考虑上述文法中对应于规则 $B \rightarrow B_1 \,||\, MB_2$ 的语义动作(1)。如果 B_1 为真，那么 B 也为真，这样 $B_1.truelist$ 中的跳转指令就成为 $B.truelist$ 的一部分。然而，如果 B_1 为假，我们下一步必须测试 B_2。因此 $B_1.falselist$ 中的跳转指令的目标必定是 B_2 的代码的起始位置。这个位置使用标记非

终结符号 M 获得。在即将生成 B_2 代码之前，M 生成了下一条指令的序号，存放在综合属性 $M.instr$ 中。

为了获得指令序号，我们将产生式 $M \rightarrow \epsilon$ 和语义动作

$$\{M.instr = nextinstr;\}$$

关联起来。变量 *nextinstr* 保存了紧跟着的下一条指令的序号。当我们已经看到了产生式 $B \rightarrow B_1 \parallel M\ B_2$ 的余下部分时，这个值将被回填到 $B_1.falselist$ 中的指令上（即 $B_1.falselist$ 中的每条指令都把 $M.instr$ 当作目标标号）。

$B \rightarrow B_1 \&\&\ M\ B_2$ 的语义动作(2)和动作(1)类似。$B \rightarrow !\ B$ 的语义动作(3)对换真假列表。动作(4)只是忽略括号。

为简单起见，语义动作(5)生成了两条指令：一个条件转移指令 goto 和一个无条件转移指令。它们的目标标号都未填写。这两个指令被放入新的分别由 $B.truelist$ 和 $B.falselist$ 指向的列表中。

例 6.24 再次考虑表达式

$$x < 100 \parallel x > 200 \ \&\& \ x != y$$

它的一棵注释语法分析树如图 6-44 所示。为了增加可读性，属性 *truelist*、*falselist* 和 *instr* 分别用它们的第一个字母表示。在对这棵语法树进行深度优先遍历时执行语义动作。因为所有的动作都出现在规则右部的最后，因此它们可以和自底向上语法分析过程中的归约动作同时进行。在根据产生式(5)将 $x < 100$ 归约为 B 时，语义动作相应地产生两条指令：

```
100:  if x < 100 goto _
101:  goto _
```

我们任意地从 100 开始为指令编号。产生式

$$B \rightarrow B_1 \parallel M\ B_2$$

中的标记非终结符号 M 记录了 *nextinstr* 的值，此时这个值为 102。使用产生式(5)将 $x > 200$ 归约为 B 产生下面两条指令

```
102:  if x > 200 goto _
103:  goto _
```

子表达式 $x > 200$ 对应于下面产生式中的 B_1：

$$B \rightarrow B_1 \ \&\&\ M\ B_2$$

标记非终结符号 M 记录了 *nextinstr* 的当前值，现在是 104。使用产生式(5)将 $x != y$ 归约为 B 产生下列指令

```
104:  if x != y goto _
105:  goto _
```

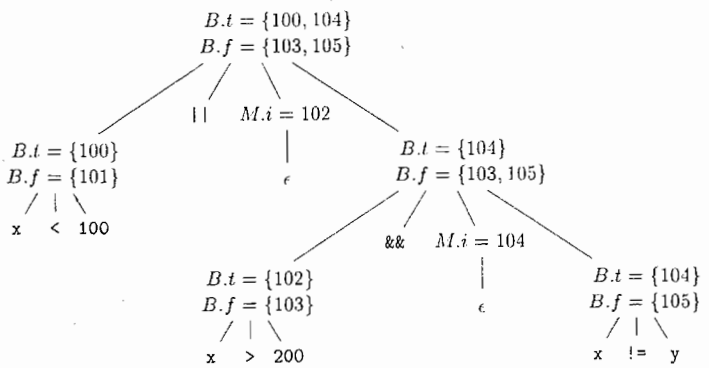


图 6-44 $x < 100 \parallel x > 200 \ \&\& \ x != y$ 的注释语法分析树

我们现在使用 $B \rightarrow B_1$ &&M B_2 进行归约。相应的语义动作调用 *backpatch*(B_1. *truelist*, M. *instr*) 将 B_1 的真值出口绑定到 B_2 的第一条指令处。因为 B_1. *truelist* 是{102}, M. *instr* 是104，这次对 *backpatch* 的调用将序号 104 填写到 102 指令中。至今为止产生的六条指令如图 6-45a 所示。

和最后一次归约使用的产生式 $B \rightarrow B_1 \parallel M\ B_2$ 相关联的语义动作调用 *backpatch*({101}, 102)，得到的指令如图 6-45b 所示。

整个表达式为真当且仅当控制流到达 100 和 104 位置上的跳转指令；表达式为假当且仅当控制流到达103 和 105 位置上的跳转指令。在后续的编译过程中，当已知表达式为真或假时分别应该做什么的时候，这些指令的目标将会被填写完整。 □

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

a) 将 104 回填到指令 102 中之后

```
100:  if x < 100 goto _
101:  goto 102
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

b) 将 102 回填到指令 101 中之后

图 6-45 回填的步骤

6.7.3 控制转移语句

现在我们使用回填技术在一趟扫描中完成控制流语句的翻译。考虑由下列文法产生的语句：

$$S \rightarrow \textbf{if}(B)\ S \mid \textbf{if}(B)\ S\ \textbf{else}\ S \mid \textbf{while}(B)\ S \mid \{L\} \mid A\ ;$$
$$L \rightarrow L\ S \mid S$$

这里 S 表示一个语句，L 是一个语句的列表，A 是一个赋值语句，B 是一个布尔表达式。请注意，一定还存在一些其他的产生式，比如那些关于赋值语句的产生式。然而，这里给出的这些产生式已经足以用来说明在控制流语句的翻译中用到的技术。

语句 if、if-else 和 while 的代码布局和 6.6 节中的描述一样。我们给出一个隐含的假设，即指令数组中的代码顺序反映了控制流的自然流动，即控制从一条语句到达下一条语句。假如没有这个假设，那么我们就必须明确插入跳转指令来实现自然的顺序控制流。

图 6-46 中的翻译方案保留了多个跳转指令的列表，当确定了这些跳转指令的目标序号后就会回填列表。如图 6-43 所示，由非终结符号 B 生成的布尔表达式有两个跳转指令列表：B. *truelist* 和 B. *falselist*。它们分别对应于 B 的代码的真假出口。由非终结符号 S 和 L 生成的语句也有一个待回填的跳转指令列表，由属性 *nextlist* 表示。列表 S. *nextlist* 中包含了所有跳转到按照运行顺序紧跟在 S 代码之后的指令的条件或无条件转移指令。L. *nextlist* 的定义与此类似。

考虑图 6-46 中的语义动作(3)。产生式 $S \rightarrow$ **while** $(B)\ S_1$ 的代码布局如图 6-35c 所示。标记非终结符号 M 在产生式

$$S \rightarrow \textbf{while}\ M_1(B)\ M_2\ S_1$$

中的两次出现分别记录了 B 的代码和 S_1 的代码的开始处的指令编号。它们分别对应于图 6-35c 中的标号 *begin* 和 B. *true*。

M 还是只有唯一的产生式 $M \rightarrow \epsilon$。图 6-46 中的动作(6)将属性 M. *instr* 的值设为下一条指令的序号。在 while 语句的循环体 S_1 执行之后，控制流回到此语句的起始位置。因此，在将 **while** $M_1(B)\ M_2\ S_1$ 归约为 S 的时候，我们对 S_1. *nextlist* 中的所有跳转指令进行回填，使得该列表中所有指令的目标为序号 M_1. *instr*。在 S_1 的代码之后显式地插入了一条跳转到 B 的代码的开始处的指令，这是因为控制流也有可能“穿越底部”。通过将 B. *truelist* 中的指令设置为转向 M_2. *instr*，我们将 B. *truelist* 回填为 S_1 代码的起始位置。

在为条件语句 **if**$(B)\ S_1$ **else** S_2 生成代码时，我们可以看到更加有说服力的使用 S. *nextlist* 和 L. *nextlist* 的理由。如果控制流“穿越”了 S_1 的代码的底部，比如当 S_1 是一个赋值语句时就会发生

这样的事情，我们必须在 S_1 的代码之后增加一条越过 S_2 代码的跳转指令。我们使用位于 S_1 之后的另一个标记非终结符号来生成这个跳转指令。假定这个标记非终结符号为 N，且其产生式为 $N \rightarrow \epsilon$。N 有属性 $N.nextlist$，它是一个由 N 的语义动作(7)生成的跳转指令 goto _的序号组成的列表。

1)	$S \rightarrow$ **if** $(B)\ M\ S_1$	{ $backpatch(B.truelist,\ M.instr)$; $S.nextlist = merge(B.falselist,\ S_1.nextlist)$; }
2)	$S \rightarrow$ **if** $(B)\ M_1\ S_1\ N$ **else** $M_2\ S_2$	{ $backpatch(B.truelist,\ M_1.instr)$; $backpatch(B.falselist,\ M_2.instr)$; $temp = merge(S_1.nextlist,\ N.nextlist)$; $S.nextlist = merge(temp,\ S_2.nextlist)$; }
3)	$S \rightarrow$ **while** $M_1\ (B)\ M_2\ S_1$	{ $backpatch(S_1.nextlist,\ M_1.instr)$; $backpatch(B.truelist,\ M_2.instr)$; $S.nextlist = B.falselist$; $gen('\texttt{goto}'\ M_1.instr)$; }
4)	$S \rightarrow \{\ L\ \}$	{ $S.nextlist = L.nextlist$; }
5)	$S \rightarrow A\ ;$	{ $S.nextlist =$ **null**; }
6)	$M \rightarrow \epsilon$	{ $M.instr = nextinstr$; }
7)	$N \rightarrow \epsilon$	{ $N.nextlist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
8)	$L \rightarrow L_1\ M\ S$	{ $backpatch(L_1.nextlist,\ M.instr)$; $L.nextlist = S.nextlist$; }
9)	$L \rightarrow S$	{ $L.nextlist = S.nextlist$; }

图 6-46 语句的翻译

图 6-46 中的语义动作(2)处理满足下列语法的 if-else 语句：

$$S \rightarrow \textbf{if}\ (B)\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$$

我们将对应于 B 为真的跳转指令回填为 $M_1.instr$，也就是 S_1 的代码的开始位置。类似地，我们将回填那些对应于 B 为假的跳转指令，使它们跳转到 S_2 的代码的开始位置。列表 $S.nextlist$ 包含了所有从 S_1 和 S_2 中跳出的指令，也包括由 N 产生的跳转指令。(变量 *temp* 是仅用于合并列表的临时变量。)

语义动作(8)和(9)处理语句序列。在

$$L \rightarrow L_1\ M\ S$$

中，按照执行顺序紧跟在 L_1 的代码之后的是 S 的开始指令。因此，列表 $L_1.nextlist$ 被回填为 S 代码的开始位置，该位置由 $M.instr$ 给出。在 $L \rightarrow S$ 中，$L.nextlist$ 和 $S.nextlist$ 相同。

请注意，除了语义规则(3)和(7)之外，这些语义规则中的任何地方都没有产生新的指令。其他所有的代码都是由赋值语句和表达式相关的语义动作产生的。我们根据控制流进行了正确的回填，因此赋值语句和布尔表达式的求值过程被正确地连接了起来。

6.7.4 break 语句、continue 语句和 goto 语句

用于改变程序控制流的最基本的程序设计语言结构是 goto 语句。在 C 语言中，像 goto L 这

样的语句将控制流转到标号为 L 的指令——在相应作用域内必须恰好存在一条标号为 L 的语句。在实现 goto 语句时，可以为每个标号维护一个未完成跳转指令的列表，然后在知道这些指令的目标之后进行回填。

Java 废除了 goto 语句。但是 Java 支持一种规范化的跳转语句，即 break 语句。它使控制流跳出外围的语言结构。Java 中还可以使用 continue 语句。这个语句的作用是触发外围循环的下一轮迭代。下面的代码摘自一个语法分析器，它说明了简单的 break 语句和 continue 语句。

```
1)  for ( ; ; readch() ) {
2)      if( peek == ' ' || peek == '\t' ) continue;
3)      else if( peek == '\n' ) line = line + 1;
4)      else break;
5)  }
```

控制流会从第 4 行中的 break 语句跳出到外围 for-循环之后的下一个语句。控制流也会从第 2 行中的 continue 语句跳转到计算 *reach*( ) 的代码，然后再转到第 2 行中的 if 语句。

如果 S 表示外围的循环结构，那么一条 break 语句就是跳转到 S 代码之后第一条指令处的跳转指令。我们可以按照下面的步骤为 break 生成代码：①跟踪外围循环语句 S，②为该 break 语句生成未完成的跳转指令，③将这些指令放到 S. *nextlist* 中，其中 *nextlist* 就是 6.7.3 节中讨论的列表。

在一个通过两趟扫描构建抽象语法树的编译器前端中，S. *nextlist* 可以被实现为对应于语句 S 的结点的一个字段。我们可以在符号表中将一个特殊的标识符 **break** 映射为表示外围循环语句 S 的结点，以此来跟踪 S。这种方法同样可以处理 java 中带标号的 break 语句，因为同样可以用符号表来将这个标号映射为对应于标号所指的结构的语法树结点。

如果不使用符号表来访问 S 的结点，我们还可以在符号表中设置一个指向 S. *nextlist* 的指针。现在当遇到一个 break 语句时，我们生成一个未完成的跳转指令，并通过符号表查找到 *nextlist*，然后把这个跳转指令加入到这个列表中。这个 *nextlist* 将按照 6.7.3 节中讨论的方法进行回填。

continue 语句的处理方法和 break 语句的处理方法类似。两者之间的主要区别在于生成的跳转指令的目标不同。

6.7.5 6.7 节的练习

练习 6.7.1：使用图 6-43 中的翻译方案翻译下列表达式。给出每个子表达式的 *truelist* 和 *falselist*。你可以假设第一条被生成的指令的地址是 100。

1) `a==b && (c==d || e==f)`

2) `(a==b || c==d) || e==f`

3) `(a==b && c==d) && e==f`

练习 6.7.2：图 6-47a 中给出了一个程序的摘要。6-47b 概述了使用图 6-46 中的回填翻译方案生成的三地址代码的结构。这里，$i_1 \sim i_8$ 是每个 code 区域的第一条被生成指令的标号。当我们实现这个翻译时，我们为每个布尔表达式 E 维护了两个列表，表中给出 E 的代码中的一些位置。我们分别用 E. *true* 和 E. *false* 来表示这两个列表。对于 E. *true* 列表中的那些指令位置，我们最终要加入当 E 为真时控制流应该到达的语句的标号。E. *false* 是类似的存放特定位置号的列表，我们要在这些位置上加入当发现 E 为假时控制流应该到达的标号。同时，我们还为语句 S 维护了一个位置的列表。我们必须在这些位置上加入当 S 执行完毕之后控制流应该到达的标号。请给出最终将代替下列各个列表中的位置的值(即 $i_1 \sim i_8$ 中的某个标号)。

(1) E_3. *false* (2) S_2. *next* (3) E_4. *false* (4) S_1. *next* (5) E_2. *true*

练习 6.7.3：当使用图 6-46 中的翻译方案对图 6-47 进行翻译时，我们为每条语句创建 S. *next* 列表。一开始是赋值语句 S_1、S_2、S_3，然后逐步处理越来越大的 if 语句、if-else 语句、while 语句和语句块。在图 6-47 中有 5 个这种类型的结构语句：

S_4：**while**(E_3) S_1。

S_5：**if**(E_4) S_2。

S_6：包含 S_5 和 S_3 的语句块。

S_7：语句 **if** (E_2)S_4 **else** S_6。

S_8：整个程序。

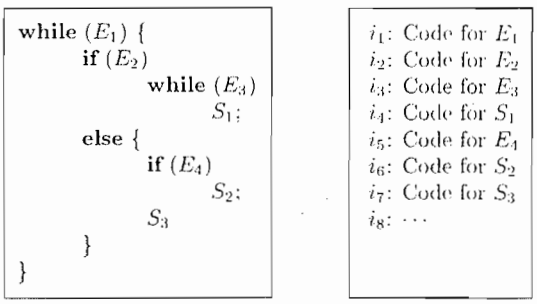


图 6-47 练习 6.7.2 的程序的控制流结构

对于这些结构语句，我们可以通过一个规则用其他的 S_j. *next* 列表以及程序中的表达式的列表 E_k. *true* 和 E_k. *false* 构造出 S_i. *next*。给出计算下列 *next* 列表的规则：

（1）S_4. *next* （2）S_5. *next* （3）S_6. *next* （4）S_7. *next* （5）S_8. *next*

6.8 switch 语句

很多语言都使用“switch”或“case”语句。我们的 switch 语句的语法如图 6-48 所示。语句中包含一个待求值的选择表达式 E，后面是该表达式可能取的 n 个常量值 V_1，V_2，…，V_n。语句中也可能包含一个默认“值”，当其他值都不和选择表达式的值匹配时，就用这个默认值来匹配。

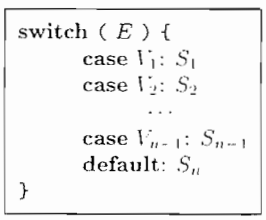


图 6-48 Switch 语句的语法

6.8.1 switch 语句的翻译

一个 switch 语句的预期翻译结果是完成如下工作的代码：

1）计算表达式 E 的值。

2）在 case 列表中寻找与表达式值相同的值 V_j。回顾一下，当在 case 列表中明确列出的值都不和表达式匹配时，就用默认值和表达式匹配。

3）执行和匹配值关联的语句 S_j。

步骤(2)是一个 n 路分支，它可以采取多种方法实现。如果 case 的数目较少，比如不多于 10 个，那么可以使用一个条件跳转指令序列来实现。每一个条件跳转指令都测试一个常量值，并跳转到这个值对应的语句的代码。

实现这个条件跳转指令序列的一个简洁的方法是创建一个对照关系表。表中的每一个关系都包含了一个常量值和相应语句代码的标号。在运行时刻，表达式自身的值以及默认语句的标号被放在对照表的末端。编译器生成一个简单循环，把表达式的值和表中的每个值进行比较。我们已经保证了当找不到其他匹配时，最后一个条目(默认值条目)一定会匹配。

如果值的个数超过 10 个或更多，那么更高效的方式是为这些值构造一个散列表。这个表的条目是各个分支语句的标号。如果没有找到对应于 switch 表达式的值的条目，就会有一条跳转指令转到默认语句。

还有一种常见的特殊情况，它的实现可以比 n 路分支更加高效。如果表达式的值位于某个较小的范围内，比如从 min 到 max，并且不同常量值的总数接近 max - min。那么我们可以构造一

个包含 max - min 个“桶”的数组，其中桶 j - min 包含了对应于值 j 的语句的标号；任何没有被填入对应标号的“桶”中包含了默认标号。

执行 switch 语句时，首先计算表达式并获得值 j；检查它是否在 min 到 max 的范围之内，如是则间接跳转到偏移量为 j - min 的条目中的标号。例如，如果表达式的类型是字符型，我们可以创建一个包含 128 个条目(根据具体的字符集，条目个数可有不同)的表，并且不进行范围检查直接进行控制流跳转。

6.8.2 switch 语句的语法制导翻译

图 6-49 中的中间代码是图 6-48 中的 switch 语句的一个近似翻译结果。所有的测试都出现在代码的末端，因此一个简单的代码生成器就可以识别出多路分支，并使用本节开始时介绍的多种实现方法中最合适的实现方法来生成高效的代码。

图 6-50 中显示的是一个更直接的代码序列。它要求编译器进行更加深入的分析，才能找到最高效的实现。值得注意的是，在一趟式编译器中，将分支语句放在开始的位置会造成不便，因为编译器此时还没有碰到各个语句 S_i，无法生成转向各个语句的代码。

为了翻译成如图 6-49 所示的形式，当我们看到关键字 **switch** 的时候，我们生成两个新标号 `test` 和 `next` 以及一个临时变量 t。然后，当我们对表达式 E 进行语法分析的时候，生成计算 E 值并将其保存到 t 的代码。处理完 E 之后，产生跳转指令 `goto test`。

当我们看见各个 **case** 关键字时，就创建一个新的标号 L_i，并将其加入符号表。我们将在一个仅用于存放 case 分支的队列中放入一个值 - 标号对。这个值 - 标号对由常量值 V_i 和 L_i(或者是指向符号表中 L_i 的条目的指针)组成。我们逐个处理语句 **case** V_i : S_i，生成附加于 S_i 的代码上的标号 L_i。最后生成跳转指令 `goto next`。

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        ...
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```

图 6-49 一个 switch 语句的翻译结果

```
        code to evaluate E into t
        if t != V1 goto L1
        code for S1
        goto next
L1:     if t != V2 goto L2
        code for S2
        goto next
L2:
        ...
Ln-2:   if t != Vn-1 goto Ln-1
        code for Sn-1
        goto next
Ln-1:   code for Sn
next:
```

图 6-50 一个 switch 语句的另一种翻译

当编译器到达 switch 语句的末端时，我们已经可以生成 n 路分支的代码了。读取值 - 标号对的队列，我们就可以生成形如图 6-51 所示的三地址语句序列。其中 t 是一个保存选择表达式 E 的值的临时变量，L_n 为默认语句的标号。

```
case t V1 L1
case t V2 L2
...
case t Vn-1 Ln-1
case t t Ln
next:
```

图 6-51 用来翻译 switch 语句的 case 三地址代码指令

指令 `case t` V_i L_i 和图 6-49 中的 `if t =` V_i `goto` L_i 含义相同，但是 `case` 指令更加容易被最终的代码生成器探测到，从而对这些指令进行某种特殊处理。在代码生成阶段，

根据分支的个数以及这些值是否在一个较小的范围内，这些 case 语句的序列可以被翻译成最高效的 n 路分支。

6.8.3 6.8 节的练习

! 练习 6.8.1：为将 switch 语句翻译成一个如图 6-51 所示的 case 语句序列，翻译器需要在处理 switch 语句的源代码时创建一个由值 - 标号对组成的列表。我们可以使用一个附加的翻译方案来做到这一点，这个方案只搜集这些值 - 标号对。给出一个语法制导定义的概要描述。该 SDD 可以生成值 - 标号对照表，同时还为各个语句 S_i 生成代码。这里的 S_i 是各个 case 对应的动作。

6.9 过程的中间代码

过程及其实现将在第 7 章中与运行时刻的变量存储管理一并详细地讨论。本节我们使用术语“函数”来表示带有返回值的过程。我们将简单讨论函数声明以及函数调用的三地址代码。在三地址代码中，函数调用被拆分为准备进行调用时的参数求值，然后是调用本身。为简单起见，我们假定参数使用值传递的方式。1.6.6 节中曾讨论过参数传递方法。

例 6.25 假定 a 是一个整数数组，并且 f 是一个从整数到整数的函数。那么赋值语句

```
n = f(a[i]);
```

可以被翻译成如下的三地址代码。

```
1)  t1 = i * 4
2)  t2 = a [ t1 ]
3)  param t2
4)  t3 = call f, 1
5)  n = t3
```

如 6.4 节中讨论的，前两行计算表达式 a[i] 的值，并将结果存放到临时变量 t_2 中。第 3 行将 t_2 作为实在参数用于第 4 行中对 f 的调用。这个调用只带有一个参数。第 4 行中函数调用的返回值被赋给 t_3。第 5 行将返回值赋给 n。 □

图 6-52 中的产生式可以生成函数定义和函数调用。（这个文法会在最后一个参数之后生成一个不必要的逗号，但是它已经足以说明翻译的方法了。）如 6.3 节所述，非终结符号 D 和 T 分别生成声明和类型。由 D 生成的函数定义包含了关键字 **define**、返回类型、函数名、括号中的形式参数以及由一个位于花括号中的语句组成的函数体。非终结符号 F 生成 0 个或多个形式参数，每个形式参数包括一个类型和一个标识符。非终结符号 S 和 E 分别生成语句和表达式。S 的产生式增加了一条返回表达式值的语句。E 的产生式中增加了函数调用，调用中的实在参数由 A 生成。一个实在参数就是一个表达式。

D	$\rightarrow$	**define** T **id** (F) { S }
F	$\rightarrow$	ϵ | T **id** , F
S	$\rightarrow$	**return** E ;
E	$\rightarrow$	**id** (A)
A	$\rightarrow$	ϵ | E , A

图 6-52 在源语言中加入函数

函数定义和函数调用可以用本章中已经介绍过的概念进行翻译。

- *函数类型*。一个函数类型必须包含它的返回值类型和形式参数类型。令 *void* 是一个表示没有参数或没有返回值的特殊类型。因此，返回一个整数的函数 *pop*( ) 的类型是“从 *void* 到 *integer* 的函数”。函数类型可以在返回值类型和有序的参数类型列表上应用构造算子 *fun* 来表示。
- *符号表*。设编译器处理到一个函数定义时，最上层的符号表为 s。函数名被放入 s，以便在程序的其他部分使用。函数的形式参数可以用类似于记录字段名的方式来处理（见图 6-18）。在 D 的产生式中，在看到关键字 **define** 和函数名之后，我们将 s 压栈并建立新的符号表

Env.push(*top*); *top* = **new** *Env*(*top*);

这个新符号表被称为 t。注意，*top* 被作为参数传递到 **new** *Env*(*top*)，因此新的符号表 t 可以被链接到先前的符号表 s。新的符号表 t 用于这个函数的函数体的翻译。在这个函数体被翻译完成之后，我们恢复到先前的符号表 s。

- *类型检查*。在表达式中，一个函数和运算符的处理方法相同。因此在 6.5.2 节中讨论的类型检查规则(包括自动类型转换)仍然可用。例如，如果 f 是一个带有一个实数型参数的函数，那么在函数调用 $f(2)$ 时，整数 2 将被转换成实型数。
- *函数调用*。当为一个函数调用 **id**(E, E,…, E)生成三地址指令的时候，只需要生成对各个参数 E 求值的三地址指令，或者生成将各个参数 E 归约为地址的三地址指令，然后再为每个参数生成一条 `param` 指令即可。如果我们不愿将参数计算指令和 `param` 指令混在一起，可以将每个表达式 E 的属性 $E.addr$ 存放到一个数据结构(比如队列)中。一旦所有的表达式都翻译完成，我们就可以在清空队列的同时生成 `param` 指令。

过程是程序设计语言中重要且常用的编程结构，因此编译器必须为过程调用和返回生成良好的代码。用于处理过程的参数传递、调用和返回的运行时刻例程是运行时刻支持系统的一部分。运行时刻支持机制将在第 7 章中讨论。

6.10 第 6 章总结

本章中介绍的技术可以被综合起来，构造一个简单的编译器前端，比如附录 A 中的那个编译器前端。编译器的前端可以增量式地进行构造：

- *选择一个中间表示形式*：中间表示形式通常是一个图形表示方法和三地址代码的组合。比如在语法树中，图中的结点表示一个程序构造；而各个子结点表示其子构造。三地址代码的名字源于它的 $x = y$ **op** z 的形式。每条指令至多有一个运算符。另外还有一些用于控制流的三地址指令。
- *翻译表达式*：通过在各个形如 $E \rightarrow E_1$ **op** E_2 的产生式中加入语义动作，带有复杂运算的表达式可以被分解成一个由单一运算组成的序列。这些动作或者创建一个 E 的结点，此结点的子结点为 E_1 和 E_2；或者生成一条三地址指令，该指令对 E_1 和 E_2 的地址应用运算符 **op**，并将其运算结果放入一个临时变量中。这个临时变量就成了 E 的地址。
- *检查类型*：一个表达式 E_1 **op** E_2 的类型是由运算符 **op** 以及 E_1 和 E_2 的类型决定的。自动类型转换(coercion)是指隐式的类型转换，例如从 *integer* 转换到 *float*。中间代码中还包含了显式的类型转换，以保证运算分量的类型和运算符的期待类型精确匹配。
- *使用符号表来实现声明*：一个声明指定了一个名字的类型。一个类型的宽度是指存放该类型的变量所需要的存储空间。使用宽度，一个变量在运行时刻的相对地址可以计算为相对于某个数据区域的开始地址的偏移量。每个声明都会将一个名字的类型和相对地址放入符号表，这样当这个名字后来出现在一个表达式中时，翻译器就可以获取这些信息。
- *将数组扁平化*：为实现快速访问，数组元素被存放在一段连续的空间内。数组的数组可以被扁平化，当作各个元素的一维数组进行处理。数组的类型用于计算一个数组元素相对于数组基地址的偏移量。
- *为布尔表达式产生跳转代码*：在短路(或者说跳转)代码中，布尔表达式的值被隐含在代码所到达的位置中。因为布尔表达式 B 常常被用于决定控制流，例如在 **if**(B)S 中就是这样，因此跳转指令是有用的。只要使得程序正确地跳转到代码 `t = true` 或 `t = flase` 处，就可以计算出布尔值，其中的 t 是一个临时变量。使用跳转标号，通过继承对应于一个布尔表达式的真假出口的标号，就可以对布尔表达式进行翻译。常量 *true* 和 *false* 分别

被翻译成跳转到真值出口和假值出口的指令。

- 用控制流实现语句：通过继承 *next* 标号就可以实现语句的翻译，其中 *next* 标记了这个语句的代码之后的第一条指令。翻译条件语句 $S \rightarrow \textbf{if}\ (B)\ S_1$ 时，只需要将一个标记了 S_1 的代码起始位置的新标号和 $S.next$ 分别作为 B 的真值出口和假值出口传递给其他处理程序。
- 可以选择使用回填技术：回填是一种为布尔表达式和语句进行一趟式代码生成的技术。其基本思想是维护多个由不完整跳转指令组成的列表，在同一列表中的指令具有同样的跳转目标。当目标位置已知时，将为相应列表中的所有指令填入这个目标。
- 实现记录：记录或类中的字段名可以当作声明序列进行处理。一个记录类型包含了关于它的各个域的类型和相对地址的信息。可以使用一个符号表对象来实现这个目的。

6.11 第6章参考文献

本章中的大部分技术来自于围绕 Algol60 进行的设计和实现活动。在 Pascal[11]和 C[6, 9]产生的时候，生成中间代码的语法制导翻译技术已经很成熟了。

从 20 世纪 50 年代开始，人们就开始寻求一种虚构的中间语言 ——UNCOL(面向所有编译器的语言)。如果有一个 UNCOL，我们可以把针对一种给定的源语言的前端和针对一种给定目标语言的后端连接起来，构建出一个编译器[10]。报告[10]中的指导性技术常常用于将编译器重定向。

人们用很多种方法来实现 UNCOL 思想，即将多个前端和后端混合并相互匹配。一个可重定目标的编译器包括一个前端，该前端可以和不同的后端结合起来，以便在不同机器上实现同一种给定的语言。Neliac 语言是一个带有重定目标编译器[5]的早期例子，这个编译器是使用 Neliac 本身编写的。另一种方法是为一个新的语言建立一个前端，将其翻译到一个已有的编译器上。Fedman[2]描述了在 C 编译器上加入 Fortran 的前端的方法[6][9]。GCC，即 GNU 的编译器集合[3]，支持包括 C、C++、Objective-C、Fortran、Java、Ada 等语言的前端。

值编码方法及其基于散列技术的实现来自于 Ershov[1]。

在 Java 字节码中使用类型信息来提高安全性的技术由 Gosling[4]描述。

使用合一方法求解方程组的类型推导技术被人们多次重复发现；它在 ML 上的应用由 Milner[7]描述。要对类型进行更全面的处理，可参见 Pierce[8]。

1. Ershov, A. P., "On programming of arithmetic operations," *Comm. ACM* **1**:8 (1958), pp. 3–6. See also *Comm. ACM* **1**:9 (1958), p. 16.

2. Feldman, S. I., "Implementation of a portable Fortran 77 compiler using modern tools," *ACM SIGPLAN Notices* **14**:8 (1979), pp. 98–106

3. GCC home page `http://gcc.gnu.org/`, Free Software Foundation.

4. Gosling, J., "Java intermediate bytecodes," *Proc. ACM SIGPLAN Workshop on Intermediate Representations* (1995), pp. 111–118.

5. Huskey, H. D., M. H. Halstead, and R. McArthur, "Neliac — a dialect of Algol," *Comm. ACM* **3**:8 (1960), pp. 463–468.

6. Johnson, S. C., "A tour through the portable C compiler," Bell Telephone Laboratories, Inc., Murray Hill, N. J., 1979.

7. Milner, R., "A theory of type polymorphism in programming," *J. Computer and System Sciences* **17**:3 (1978), pp. 348–375.

8. Pierce, B. C., *Types and Programming Languages*, MIT Press, Cambridge, Mass., 2002.

9. Ritchie, D. M., "A tour through the UNIX C compiler," Bell Telephone Laboratories, Inc., Murray Hill, N. J., 1979.

10. Strong, J., J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel, "The problem of programming communication with changing machines: a proposed solution," *Comm. ACM* **1**:8 (1958), pp. 12–18. Part 2: **1**:9 (1958), pp. 9–15. Report of the SHARE Ad-Hoc **Committee** on Universal Languages.

11. Wirth, N. "The design of a Pascal compiler," *Software—Practice and Experience* **1**:4 (1971), pp. 309–333.