第8章 代码生成

我们的编译器模型的最后一个步骤是代码生成器。如图 8-1 所示，它以编译器前端生成的中间表示(IR)和相关的符号表信息作为输入，输出语义等价的目标程序。

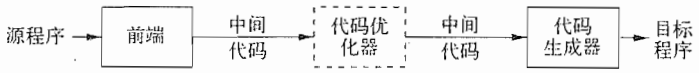


图 8-1 代码生成器的位置

对代码生成器的要求是很严格的。目标程序必须保持源程序的语义含义，还必须具有很高的质量。也就是说，它必须有效地利用目标机器上的可用资源。此外，代码生成器本身必须能够高效运行。

具有挑战性的是，从数学上讲，为给定源程序生成一个最优的目标程序是不可判定问题，在代码生成中碰到的很多子问题(比如寄存器分配)都具有难以处理的计算复杂性。在实践中，我们必须使用那些能够产生良好但不一定最优的代码的启发性技术。幸运的是，启发性技术已经非常成熟，一个精心设计的代码生成器所产生的代码要比那些由简单的生成器生成的代码快好几倍。

要产生高效目标程序的编译器都会在代码生成之前包含一个优化步骤。优化器把一个 IR 映射为另一个可用于产生高效代码的 IR。编译器的代码优化和代码生成步骤通常被称为编译器的*后端*(back end)。它们可能在生成目标程序之前对 IR 作多趟处理。代码优化将在第 9 章中详细讨论。不论代码生成之前有没有优化步骤，都可以使用本章所讨论的技术。

代码生成器有三个主要任务：指令选择、寄存器分配和指派、以及指令排序。这些任务的重要性将在 8.1 节中概述。指令选择考虑的问题是选择适当的目标机指令来实现 IR 语句。寄存器分配和指派考虑的问题是把哪个值放在哪个寄存器中。指令排序考虑的问题是按照什么顺序来安排指令的执行。

本章给出了一些和代码生成相关的算法，代码生成器可以使用这些算法把输入的 IR 翻译成简单寄存器机器的目标语言指令序列。这些算法将使用 8.2 节中的机器模型来解释。第 10 章讨论了复杂的现代机器的代码生成问题，这些现代机器支持在单一指令中的大量并行性。

在讨论了代码生成器设计中的众多难题之后，我们给出了一个编译器需要生成什么样的目标代码，以支持常见源语言中所包含的抽象机制。在 8.3 节，我们概述了静态和栈式数据区分配的实现方法，并说明如何把 IR 中的名字转换成为目标代码中的地址。

很多代码生成器把 IR 指令分成“基本块”，每个基本块由一组总是一起执行的指令组成。把 IR 划分成基本块是 8.4 节的主题。接下来介绍了针对基本块的一些简单的局部转换方法。从转换得到的基本块出发可以生成更加高效的代码。虽然要到第 9 章才开始考虑更加深入的代码优化理论，但这种转换已经是代码优化的初步形式。一个有用的局部转换的例子是在中间代码的层次上寻找公共子表达式，然后相应地把算术运算替换为更简单的拷贝运算。

8.6 节给出了一个简单的代码生成算法。它依次为每个语句生成代码，并把运算分量尽可能长时间地保留在寄存器中。这种代码生成器的输出可以很容易地使用窥孔优化技术进行优化。接下来的 8.7 节中将讨论窥孔优化技术。

其余的部分将研究指令选择和寄存器分配。

8.1 代码生成器设计中的问题

虽然代码生成器设计依赖于中间表示形式、目标语言和运行时刻系统的特定细节，但指令选择、寄存器分配和指派以及指令排序等任务会在几乎所有的代码生成器设计中碰到。

代码生成器的最重要的标准是生成正确的代码。正确性问题非常突出的原因是代码生成器会碰到很多种特殊情况。在优先考虑正确性的情况下，另一个重要的设计目标是把代码生成器设计得易于实现、测试和维护。

8.1.1 代码生成器的输入

代码生成器的输入是由前端生成的源程序的中间表示形式以及符号表中的信息组成的。这些信息用来确定 IR 中的名字所指的数据对象的运行时刻地址。

IR 的中间表示形式的选择有很多，包括诸如四元式、三元式、间接三元式等三地址表示方式；也包括诸如字节代码和堆栈机代码的虚拟机表示方式；包括诸如后缀表示的线性表示方式；还包括诸如语法树和 DAG 的图形表示方式。本章中的多个算法都是根据第 6 章中所考虑的表示方法来表示的。这些表示方法包括：三地址代码、树和 DAG。然而，我们讨论的技术也可以用于其他的中间表示形式。

在本章中，我们假设前端已经扫描、分析了源程序，并把它转换成为相对低层次的中间表示形式，因此在 IR 中出现的名字的值可以用能被目标机直接处理的量来表示。这些量可以是整数、浮点数等。我们还假设所有的语法和静态语义错误都已经被检测出来，必要的类型检查都已经完成，而类型转换运算已经被插入到必要的地方。因此，代码生成器可以在工作过程中假设它的输入已经排除了这些错误。

8.1.2 目标程序

构造一个能够产生高质量机器代码的代码生成器的难度会受到目标机器的指令集体系结构的极大影响。最常见的目标机体系结构是 RISC(精简指令集计算机)、CISC(复杂指令集计算机)和基于堆栈的结构。

RISC 机通常有很多寄存器、三地址指令、简单的寻址方式和一个相对简单的指令集体系结构。相反，CISC 机通常具有较少寄存器、两地址指令、多种寻址方式、多种类型的寄存器、可变长度的指令和具有副作用的指令。

在基于栈的机器中，运算是通过把运算分量压入一个栈，然后再对栈顶的运算分量进行运算而完成的。为了获得高性能，栈顶元素通常保存在寄存器中。因为人们觉得堆栈组织的限制太多，并且需要太多的交换和拷贝操作，所以基于堆栈的机器几乎已经消失了。

但是，基于堆栈的体系结构随着 Java 虚拟机(JVM)的出现又复活了。JVM 是一个 Java 字节码的软件解释器。字节码是由 Java 编译器生成的一种中间语言。这个解释器提供了跨平台的软件兼容性。这是 Java 成功的一个重要因素。

解释执行会引起很高的性能损失，有时可能达到 10 倍的数量级。为了克服这个问题，人们创造了即时(Just-In-Time，JIT)Java 编译器。这些即时编译器在运行时刻把字节码翻译成目标机上的本地硬件指令集。另一个提高 Java 程序性能的方法是建立一个编译器，把 Java 程序直接编译成目标机器指令，彻底绕过字节码。

输出一个使用绝对地址的机器语言程序的优点是程序可以放在内存中的某个固定位置上，并立即执行。程序可以很快地进行编译和执行。

输出可重定位的机器语言程序(通常称为目标模块，object module)可以使各个子程序能够被

分别编译。一组可重定位的目标模块可以被一个链接加载器链接到一起并加载运行。如果我们要生成可重定位的目标模块，我们就必须为链接和加载付出代价。但是这样做可以使我们得到很多的灵活性。我们可以把子程序分开编译，并能够从一个目标模块中调用其他已经编译好的程序。如果目标机没有自动处理重定位，编译器就必须向加载器提供明确的重定位信息，以便把分开编译的程序模块链接起来。

输出一个汇编程序使代码生成过程变得稍微容易一些。我们可以生成符号指令，并使用汇编器的宏机制来帮助生成代码。这么做的代价是代码生成之后还需要增加一个汇编步骤。

在本章中，我们将使用一个非常简单的类 RISC 计算机作为目标机。我们在这个机器上加入了一些类 CISC 的寻址方式。这样我们就可以讨论 CISC 机器的代码生成技术了。为了增加可读性，我们把汇编代码用作目标语言。只要变量地址可以通过偏移量和存放于符号表中的其他信息计算出来，代码生成器就可以为源程序中的名字生成可重定位地址或绝对地址。这和生成符号地址一样，都是很简单的事情。

8.1.3 指令选择

代码生成器必须把 IR 程序映射成为可以在目标机上运行的代码序列。完成这个映射的复杂性由如下的因素决定：

- IR 的层次。
- 指令集体系结构本身的特性。
- 想要达到的生成代码的质量。

如果 IR 是高层次的，代码生成器就要使用代码模板把每个 IR 语句翻译成为机器指令序列。但是，这种逐个语句生成代码的方式通常会产生质量不佳的代码。这些代码需要进一步优化。如果 IR 中反映了相关计算机的某些低层次细节，那么代码生成器就可以使用这些信息来生成更加高效的代码序列。

目标机指令集本身的特性对指令选择的难度有很大的影响。比如，指令集的统一性和完整性是两个很重要的因素。如果目标机没有以统一的方式支持每种数据类型，那么总体规则的每个例外都需要进行特别处理。比如，在某些机器上，浮点数运算使用单独的寄存器完成。

指令速度和机器的特有用法是另外一些重要因素。如果我们不考虑目标程序的效率，那么指令选择是很简单的。对于每一种三地址语句，我们可以生成一个代码骨架。此骨架定义了对这个构造生成什么样的目标代码。比如，每一个形如 x = y + z 的三地址语句(其中 x、y 和 z 都是静态分配的)可以被翻译成如下的代码序列：

```
LD  R0, y       // R0 = y          (把y装载到寄存器R0)
ADD R0, R0, z   // R0 = R0 + z     (把z加到R0)
ST  x, R0       // x = R0          (把R0保存到x)
```

这种策略常常会产生冗余的加载和存储运算。比如，下面的三地址语句序列

```
a = b + c
d = a + e
```

会被翻译成

```
LD  R0, b       // R0 = b
ADD R0, R0, c   // R0 = R0 + c
ST  a, R0       // a = R0
LD  R0, a       // R0 = a
ADD R0, R0, e   // R0 = R0 + e
ST  d, R0       // d = R0
```

这里的第四个语句是冗余的，因为它加载了一个刚刚保存到内存的值。并且如果 a 以后不再被使用，那么第三个语句也是冗余的。

生成代码的质量通常是由它的运行速度和大小来确定的。在大多数机器上，一个给定的 IR 程序可以用很多种不同的代码序列来实现。这些不同实现之间在代价上有着显著的差别。因此，对中间代码的简单翻译虽然能产生正确的目标代码，但是这些代码却可能过于低效而让人不可接受。

比如，如果目标机有一个“加一”指令(INC)，那么三地址语句 a = a + 1 可以用一个指令 INC a 来实现。这个指令要比如下的代码序列更加高效：把 a 加载进一个寄存器，对寄存器加 1，然后把结果保存回 a。

```
LD  R0, a       // R0 = a
ADD R0, R0, #1  // R0 = R0 + 1
ST  a, R0       // a = R0
```

要设计出良好的代码序列，我们就必须知道指令的代价。遗憾的是，我们经常难以得到精确的代价信息。对于一个给定的三地址构造，可能还需要有关该构造所在上下文的信息才能决定哪个是最好的机器代码序列。

在 8.9 节，我们将看到指令选择可以用树模式匹配过程来建模。在这个过程中，我们把 IR 和机器指令表示为树结构。然后，我们尝试着用一组对应于机器指令的子树覆盖一棵 IR 树。如果我们把每棵机器指令子树和一个代价值相关联，我们就可以用动态规划的方法来生成最优化的代码序列。动态规划将在 8.11 节中讨论。

8.1.4 寄存器分配

代码生成的关键问题之一是决定哪个值放在哪个寄存器里面。寄存器是目标机上运行速度最快的计算单元，但是我们通常没有足够的寄存器来存放所有的值。没有存放在寄存器中的值必须存放在内存中。使用寄存器运算分量的指令总是要比那些运算分量在内存中的指令短并且快。因此，有效利用寄存器非常重要。

寄存器的使用经常被分解为两个子问题：

1）*寄存器分配*：对于源程序中的每个点，我们选择一组将被存放在寄存器中的变量。

2）*寄存器指派*：我们指定一个变量被存放在哪个寄存器中。

即使对于单寄存器机器，找到一个从寄存器到变量的最优指派也是很困难的。从数学上讲，这个问题是 NP 完全的。而且，目标机的硬件和/或操作系统可能要求代码遵守特定的寄存器使用规则，从而使这个问题变得更加复杂。

例 8.1 有些机器要求为某些运算分量和结果使用*寄存器对*（即一个偶数号寄存器和相邻的奇数号寄存器）。比如，在某些机器上，整数乘法和整数除法就涉及寄存器对。乘法指令的形式如下：

```
M x, y
```

其中被乘数 x 是偶数/奇数寄存器对中的奇数号寄存器，而乘数 y 则可以存放在任意位置。乘法结果占据了整个偶数/奇数寄存器对。除法指令的形式如下：

```
D x, y
```

其中，被除数占据了整个偶数/奇数寄存器对，x 是其中的偶数号寄存器；而除数是 y。相除之后，偶数号寄存器保存余数，而奇数号寄存器保存商。

现在，考虑图 8-2 中的两个三地址代码序列。图 8-2a 和图 8-2b 之间的唯一差别是第二个语句的运算符。图 8-2a 和图 8-2b 对应的最短汇编代码序列如图 8-3 所示。

```
t = a + b
t = t * c
t = t / d
```

a)

```
t = a + b
t = t + c
t = t / d
```

b)

图 8-2 两个三地址代码序列

```
L   R1,a
A   R1,b
M   R0,c
D   R0,d
ST  R1,t
```

a)

```
L    R0, a
A    R0, b
A    R0, c
SRDA R0, 32
D    R0, d
ST   R1, t
```

b)

图 8-3 最优机器代码序列

Ri 表示第 i 号寄存器。SRDA 表示双算术右移(Shift-Right-Double-Arithmetic)，而 SRDA R0 32 把被除数从 R0 中移入 R1 并把 R0 清空，使得所有位都等于被除数的正负号位。L，ST 和 A 分别表示加载、保存和相加。需要注意的是，把 a 加载到哪个寄存器的最优选择依赖于最终会对 t 做什么样的运算。 □

寄存器的分配和指派的策略将在 8.8 节讨论。8.10 节将给出对某些类型的机器，我们可以构造出使用最少的寄存器来完成表达式求值的代码序列。

8.1.5 求值顺序

计算执行的顺序会影响目标代码的效率。我们即将看到，相比其他的计算顺序而言，某些计算顺序对用于存放中间结果的寄存器的需求更少。但是在一般情况下，找到最好的顺序是一个困难的 NP 完全问题。一开始，我们将按照中间代码生成器生成代码的顺序为三地址语句生成代码，从而暂时避开这个问题。在第 10 章，我们将研究对流水线计算机的代码排序。这种流水线计算机可以在一个时钟周期内执行多个运算。

8.2 目标语言

熟悉目标计算机及其指令集是设计一个优秀代码生成器的前提。为了给某个目标机器上的一个完整的源语言生成高质量的代码，我们需要了解该目标机的许多细节。遗憾的是，在对代码生成的一般性讨论中不可能描述出全部的细节。在本章中，我们将使用一个简单计算机的汇编代码作为目标语言。这个计算机是很多寄存器机器的代表。然而，本章中描述的很多代码生成技术也可以用于很多其他类型的机器。

8.2.1 一个简单的目标机模型

我们的目标计算机是一个三地址机器的模型。它具有加载和保存操作、计算操作、跳转操作和条件跳转。这个计算机的内存按照字节寻址，它具有 n 个通用寄存器 R0，R1，…，R$n-1$。一个完整的汇编语言具有几十到上百个指令。为了避免因为过多的细节而妨碍对概念的解释，我们将只使用一个很有限的指令集合，并假设所有的运算分量都是整数。大部分指令包含一个运算符，然后是一个目标地址，最后是一个源运算分量的列表。指令之前可能有一个标号。我们假设有如下种类的指令可用：

- *加载运算*：指令 LD *dst*，*addr* 把位置 *addr* 上的值加载到位置 *dst*。这个指令表示赋值 *dst* = *addr*。这个指令最常见的形式是 LD r，x。它把位置 x 中的值加载到寄存器 r 中。形如 LD r_1，r_2 的指令是一个寄存器到寄存器的拷贝运算。它把寄存器 r_2 的内容拷贝到寄存器 r_1 中。
- *保存运算*：指令 ST x，r 把寄存器 r 中的值保存到位置 x。这个指令表示赋值 $x = r$。
- *计算运算*：形如 *OP dst*，src_1，src_2，其中 *OP* 是一个诸如 ADD 或 SUB 的运算符，而 *dst*、src_1 和 src_2 是内存位置。这些位置不一定要相互不同。这个机器指令的作用是把 *OP* 所代表的运算作用在位置 src_1 和 src_2 中的值上，然后把这次运算的结果放到位置 *dst* 中。比如，SUB r_1，r_2，r_3 计算了 $r_1 = r_2 - r_3$。原先存放在 r_1 中的值丢失了，但是如果 r_1 等于 r_2 或者 r_3，计算机会首先读出原来的值。只需要一个运算分量的单目运算符没有 src_2。
- *无条件跳转*：指令 BR L 使得控制流转向标号为 L 的机器指令。(BR 表示产生分支)。
- *条件跳转*：该指令的形式为 Bcond r，L，其中 r 是一个寄存器，L 是一个标号，而 *cond* 代表了对寄存器 r 中的值所做的某个常见测试。比如，当寄存器 r 中的值小于 0 时，BLTZ r，L 使得控制流跳转到标号 L；否则，控制流传递到下一个机器指令。

我们假设目标机具有多种寻址模式：

- 在指令中，一个位置可以是一个变量名 x，它指向分配给 x 的内存位置(即 x 的左值)。
- 一个位置也可以是一个带有下标的形如 $a(r)$ 的地址，其中 a 是一个变量，而 r 是一个寄存器。$a(r)$ 所表示的内存位置按照如下方式计算得到：a 的左值加上存放在寄存器 r 中的值。比如，指令 LD R1, a(R2) 的效果是 R1 = *contents*(a + *contents*(R2))，其中 *contents*(x) 表示 x 所代表的寄存器或内存位置中存放的内容。这个寻址方式对于数组访问是很有用的，其中 a 是数组的基地址(即第一个元素的地址)，而 r 中存放了从基地址到数组 a 的某个元素所要经过的字节数。
- 一个内存位置可以是一个以寄存器作为下标的整数。比如，LD R1, 100(R2)的效果就是使得 R1 = *contents*(100 + *contents*(R2))。也就是说，首先计算寄存器 R2 中的值加上 100 得到的和，然后把这个和所指向的位置中的值加载到 R1 中。正如我们在下面的例子中将看到的那样，这个寻址方式可以用于沿指针取值。
- 我们还支持另外两种间接寻址模式：$*r$ 表示在寄存器 r 的内容所表示的位置上存放的内存位置。而 *100(r)表示在 r 中内容加上 100 的和所代表的位置上的内容所代表的位置。比如，LD R1, *100(R2)的效果是把 R1 设置为 *contents*(*contents*(100 + *contents*(R2)))。也就是说，首先计算寄存器 R2 中的内容加上 100 的和，取出和值所指的位置中的内容，再把这个内容代表的位置中的值加载到 R1 中。
- 最后，我们支持一个直接常数寻址模式。在常数前面有一个前缀#。指令 LD R1, #100 把整数 100 加载到 R1 中，而 ADD R1, R1, #100 则把 100 加到寄存器 R1 中去。

在指令之后的注解由//开头。

例 8.2 三地址语句 x = y - z 可以使用下面的机器指令序列实现：

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2     // R1 = R1 - R2
ST  x, R1          // x = R1
```

也许我们能做得更好。一个优秀的代码生成算法的目标之一是尽可能地避免使用上面的全部四个指令。比如，y 和/或 z 可能已经被计算出来并存放在一个寄存器中。如果是这样，我们就可以避免相应的 LD 步骤。类似地，如果 x 的值被使用时都存放在寄存器中，并且之后不会再被用到，我们就不需要把这个值保存回 x。

假设 a 是一个元素为 8 字节值(比如实数)的数组。再假设 a 的元素的下标从 0 开始。我们可以通过下面的指令序列来执行三地址指令 b = a[i]：

```
LD  R1, i          // R1 = i
MUL R1, R1, 8      // R1 = R1 * 8
LD  R2, a(R1)      // R2 = contents(a + contents(R1))
ST  b, R2          // b = R2
```

这里的第二步计算 $8i$；而第三步把 a 的第 i 个元素的值放到 R2 中，这个元素位于离数组 a 的基地址 $8i$ 个字节的地方。

类似地，三地址指令 a[j] = c 所代表的对数组 a 的赋值可以实现为：

```
LD  R1, c          // R1 = c
LD  R2, j          // R2 = j
MUL R2, R2, 8      // R2 = R2 * 8
ST  a(R2), R1      // contents(a +  contents(R2)) = R1
```

为了实现一个简单的指针间接存取，比如三地址语句 x = *p，我们可以使用如下的机器指令序列：

```
LD  R1, p          // R1 = p
LD  R2, 0(R1)      // R2 = contents(0 + contents(R1))
ST  x, R2          // x = R2
```

通过指针的赋值语句 * p = y 可以类似地用如下的机器代码实现：

```
LD  R1, p          // R1 = p
LD  R2, y          // R2 = y
ST  0(R1), R2      // contents(0 + contents(R1)) = R2
```

最后考虑一个带条件跳转的三地址指令：

```
if x < y goto L
```

它的等价的机器代码如下：

```
LD   R1, x         // R1 = x
LD   R2, y         // R2 = y
SUB  R1, R1, R2    // R1 = R1 - R2
BLTZ R1, M         // if R1 < 0 jump to M
```

这里的 M 是从标号为 L 的三地址指令所产生的机器指令序列中的第一个指令的标号。对于任意一个三地址指令，我们希望可以省略这些指令中的某些指令。省略的原因可能是所需的运算分量已经在寄存器中了，也可能因为结果不需要存放回内存。 □

8.2.2 程序和指令的代价

我们经常会指出编译及运行一个程序所需的代价。根据我们在优化一个程序时感兴趣的方面，我们会使用不同的度量。常用的度量包括编译时间的长短，以及目标程序的大小、运行时间和能耗。

确定编译和运行一个程序的实际代价是一个复杂的问题。总的来说，为一个给定的源程序找到一个最优的目标程序是一个不可判定问题，而很多相关的子问题都是 NP 困难的。正如我们已经指出的，在代码生成时，我们通常必须满足于那些能够生成优良代码但不一定是最优目标程序的启发式技术。

在本章的其余部分，我们将假设每个目标语言指令都有相应的代价。为简单起见，我们把一个指令的代价设定为 1 加上与运算分量寻址模式相关的代价。这个代价对应于指令中字的长度。寄存器寻址模式具有的附加代价为 0，而涉及内存位置或常数的寻址方式的附加代价为 1。下面是一些例子：

- 指令 LD R0，R1 把寄存器 R1 中的内容拷贝到寄存器 R0 中。因为不要求附加的内存字，所以这个指令的代价是 1。
- 指令 LD R0，M 把内存位置 M 中的内容加载到寄存器 R0 中。指令的代价是 2，因为内存位置 M 的地址在紧跟着指令的字中。
- 指令 LD R1，*100(R2) 把值 *contents*(*contents*(100 + *contents*(R2))) 加载到寄存器 R1 中。这个指令的代价是 2，因为常数 100 存放在紧跟着指令的内存字中。

在本章中，我们假设对于一个指定的输入，目标语言程序的代价是当此程序在该输入上运行时所执行的所有指令的代价总和。优秀的代码生成算法的目标是使得程序在典型输入上运行时所执行指令的代价总和最小。我们将会看到，在某些情况下，我们真的能够在某些类型的寄存器机器上为表达式生成最优的代码。

8.2.3 8.2 节的练习

练习 8.2.1：假设所有的变量都存放在内存中，为下面的三地址语句生成代码：

1) x = 1
2) x = a
3) x = a + 1
4) x = a + b
5) 两个语句的序列

```
x = b * c
y = a + x
```

练习 8.2.2：假设 a 和 b 是元素为 4 字节值的数组，为下面的三地址语句序列生成代码。

1）四个语句的序列

```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

2）三个语句的序列

```
x = a[i]
y = b[i]
z = x * y
```

3）三个语句的序列

```
x = a[i]
y = b[x]
a[i] = y
```

练习 8.2.3：假设 p 和 q 存放在内存位置中，为下面的三地址语句序列生成代码：

```
y = *q
q = q + 4
*p = y
p = p + 4
```

练习 8.2.4：假设 x、y 和 z 存放在内存位置中，为下面的语句序列生成代码：

```
    if x < y goto L1
    z = 0
    goto L2
L1: z = 1
```

练习 8.2.5：假设 n 在一个内存位置中，为下面的语句序列生成代码：

```
    s = 0
    i = 0
L1: if i > n goto L2
    s = s + i
    i = i + 1
    goto L1
L2:
```

练习 8.2.6：确定下列指令序列的代价。

1)
```
LD  R0, y
LD  R1, z
ADD R0, R0, R1
ST  x, R0
```

2)
```
LD  R0, i
MUL R0, R0, 8
LD  R1, a(R0)
ST  b, R1
```

3)
```
LD  R0, c
LD  R1, i
MUL R1, R1, 8
ST  a(R1), R0
```

4)
```
LD R0, p
LD R1, 0(R0)
ST x, R1
```

5)
```
LD R0, p
LD R1, x
ST 0(R0), R1
```

6)
```
LD   R0, x
LD   R1, y
SUB  R0, R0, R1
BLTZ *R3, R0
```

8.3 目标代码中的地址

在本节中，我们将说明如何使用静态和栈式内存分配为简单的过程调用和返回生成代码，以此将 IR 中的名字转换成为目标代码中的地址。在 7.1 节中，我们描述了每个正在执行的程序是如何在它的逻辑地址空间上运行的。这个空间被划分成为四个代码及数据区域：

1）一个静态确定的代码区 *Code*。这个区存放可执行的目标代码。目标代码的大小可以在编译时刻确定。

2）一个静态确定的静态数据区 *Static*。这个区存放全局常量和编译器生成的其他数据。全局常量和编译器数据的大小也可以在编译时刻确定。

3）一个动态管理的堆区 *Heap*。这个区存放程序运行时刻分配和释放的数据对象。*Heap* 的大小不能在编译时刻静态确定。

4）一个动态管理的栈区 *Stack*。这个区存放过程的活动记录。活动记录会随着过程的调用和返回被创建和消除。和堆区一样，栈区的大小也不能在编译时刻确定。

8.3.1 静态分配

为了说明简化的过程调用和返回的代码生成，我们关注下面的三地址语句：

- `call` *callee*
- `return`
- `halt`
- `action`，这是代表其他三地址语句的占位符。

活动记录的大小和布局是由代码生成器通过存放于符号表中的名字的信息来确定的。我们将首先说明如何在过程调用时在一个活动记录中存放返回地址，以及如何在过程调用结束后把控制返回到这个地址。为方便起见，我们假设活动记录的第一个位置存放返回地址。

我们首先考虑实现最简单情况(即静态分配)时的代码。这里，中间代码中的 `call` *callee* 语句可以用包含两个目标机指令的序列来实现：

```
ST   callee.staticArea, #here + 20
BR   callee.codeArea
```

ST 指令把返回地址保存到 *callee* 的活动记录的开始处，而 BR 把控制传递到被调用过程 *callee* 的目标代码上。属性 *callee. staticArea* 是一个常量，给出了 *callee* 的活动记录的开始处的地址，而属性 *callee. codeArea* 也是一个常量，指向运行时刻内存中 *Code* 区中被调用过程 *callee* 的第一个指令的地址。

ST 指令中的运算分量*#here* + 20 是返回地址的文字表示，它是紧跟在 BR 指令之后的指令的地址。我们假设*#here* 是当前指令的地址，而调用序列中的三个常量加上两个指令的长度为 5 个字，即 20 个字节。

过程代码的结尾处是一个返回到调用者过程的指令。但是没有调用者的第一个过程例外，它的最后一个指令是 HALT。这个指令把控制返回给操作系统。一个`return`语句可以使用一个简单的跳转语句实现：

```
BR   *callee.staticArea
```

它把控制流转到保存在 *callee* 的活动记录开始位置的地址上。

例 8.3 假设我们有下面的三地址代码：

```
                    // c 的代码
action1
call p
```

```
action2
halt
                    // p的代码
action3
return
```

图 8-4 给出了这个三地址代码的目标程序。我们使用伪指令 ACTION 来代表执行语句 action 的机器指令序列。这些 action 语句代表了和本次讨论无关的三地址代码。我们假定过程 c 的代码从地址 100 开始，而过程 p 从地址 200 开始。我们假定每个 ACTION 伪指令占用 20 个字节。我们还假定这些过程的活动记录以静态方式分配，其位置分别是 300 和 364。

```
                                // c的代码
100:  ACTION1                   // action1的代码
120:  ST 364, #140              // 在位置364上存放返回地址140
132:  BR 200                    // 调用p
140:  ACTION2
160:  HALT                      // 返回操作系统
      ...
                                // p的代码
200:  ACTION3
220:  BR *364                   // 返回在位置364保存的地址处
      ...
                                // 300-363存放c的活动记录
300:                            // 返回地址
304:                            // c的局部数据
      ...
                                // 364-451存放p的活动记录
364:                            // 返回地址
368:                            // p的局部数据
```

图 8-4 静态分配的目标代码

从地址 100 开始的指令实现了过程 c 的语句：

```
action1; call p; action2; halt
```

因此程序的运行从地址 100 上的指令 $ACTION_1$ 开始。在地址 120 上的 ST 指令把返回地址 140 存放在机器状态字段中，也就是 p 的活动记录的第一个字中。在地址 132 上的 BR 指令把控制转移到被调用过程 p 的目标代码的第一个指令。

执行了 $ACTION_3$ 之后，位于地址 220 的跳转指令被执行。因为上面的调用代码序列把位置 140 存放在地址 364 中，因此当位于地址 220 的 BR 语句执行时，*364 代表 140。所以当过程 p 结束时，控制流返回到地址 140，过程 c 继续执行。 □

8.3.2 栈分配

如果在保存活动记录时使用相对地址，静态分配就可以变成栈分配。但是在栈分配方式中，只有等到运行时刻才能知道一个过程的活动记录的位置。这个位置通常存放在一个寄存器里面，因此活动记录中的字可以通过相对于寄存器中值的偏移量来访问。我们的目标机的下标地址模式可以方便地完成这种访问。

正如我们在第 7 章中已经看到的，活动记录的相对地址可以用相对于活动记录中的任一已知位置的偏移量来表示。为方便起见，我们将在寄存器 SP 中维护一个指向栈顶的活动记录的开始处的指针，这样就可以使所有的偏移量都是正数。当发生过程调用时，调用过程增加 SP 的值，并把控制传递到被调用过程。在控制返回到调用者时，我们减少 SP 的值，从而释放被调用过程的活动记录。

第一个过程的代码把 SP 设置成内存中栈区的开始位置，完成对栈的初始化：

```
LD   SP, #stackStart              // 初始化栈
code for the first procedure
HALT                              // 结束执行
```

一个过程调用指令序列增加 SP 的值，保存返回地址，并把控制传递到被调用过程：

```
ADD  SP, SP, #caller.recordSize   // 增加栈指针
ST   0(SP), #here + 16            // 保存返回地址
BR   callee.codeArea              // 转移到被调用过程
```

运算分量*#caller. recordSize* 表示一个活动记录的大小，因此 ADD 指令使得 SP 指向下一个活动记录。在 ST 指令中的运算分量*#here* + 16 是跟随在 BR 之后的指令的地址，它被存放在 SP 所指向的地址中。

返回指令序列包含两个部分。被调用过程使用下面的指令把控制传递到返回地址：

```
BR   *0(SP)                       // 返回给调用者
```

在 BR 中使用 *0(SP)的原因是我们需要两层间接寻址：0(SP)是活动记录的第一个字所在的位置，而 *0(SP)是存放在那里的返回地址。

返回指令序列的第二部分在调用者中，这个序列减少 SP 的值，因此把 SP 恢复为以前的值。也就是说，在减法运算之后，SP 指向调用者的活动记录的开始处：

```
SUB  SP, SP, #caller.recordSize   // 栈指针减 l
```

第 7 章中包含了有关调用指令序列以及在调用过程和被调用过程之间进行任务分配的折衷方案的更广泛的讨论。

例 8.4 图 8-5 中的程序是前一章中的快速排序程序的一个抽象。过程 q 是递归的，因此在同一时刻可能有多个活跃的 q 的活动记录。

```
                    // m 的代码
action1
call q
action2
halt
                    // p的代码
action3
return
                    // q 的代码
action4
call p
action5
call q
action6
call q
return
```

图 8-5 例 8.4 的代码

假设过程 m、p 和 q 的活动记录的大小已经确定，分别是 *msize*、*psize* 和 *qsize*。每个活动记录的第一个字存放返回地址。我们随意地假设这些过程的代码分别从地址 100、200 和 300 处开始，并假设栈区在地址 600 处开始。目标程序在图 8-6 中显示。

```
                        // m的代码
100:  LD SP, #600       // 初始化栈
108:  ACTION1           // action1的代码
128:  ADD SP, SP, #msize // 调用指令序列的开始
136:  ST 0(SP), #152    // 将返回地址压入栈
144:  BR 300            // 调用q
152:  SUB SP, SP, #msize // 恢复SP的值
160:  ACTION2
180:  HALT
      ...
                        // p的代码
200:  ACTION3
220:  BR *0(SP)         // 返回
      ...
                        // q的代码
300:  ACTION4           // 包含有跳转到456的条件转移指令
320:  ADD SP, SP, #qsize
328:  ST 0(SP), #344    // 将返回地址压入栈
336:  BR 200            // 调用p
344:  SUB SP, SP, #qsize
352:  ACTION5
```

图 8-6 栈式分配时的目标代码

```
372:  ADD SP, SP, #qsize
380:  BR 0(SP), #396          // 将返回地址压入栈
388:  BR 300                  // 调用q
396:  SUB SP, SP, #qsize
404:  ACTION6
424:  ADD SP, SP, #qsize
432:  ST 0(SP), #440          // 将返回地址压入栈
440:  BR 300                  // 调用q
448:  SUB SP, SP, #qsize
456:  BR *0(SP)               // 返回
      ...
600:                          // 栈区的开始处
```

图 8-6 （续）

我们假设 $\mathtt{ACTION}_4$ 包含了一个条件跳转指令，跳转到 q 的返回代码序列开始地址 456；否则，递归过程 q 将不得不永远调用自己。

令 *msize*、*psize* 和 *qsize* 分别是 20、40 和 60。在地址 100 处的第一个指令把 SP 初始化为 600，即栈区的开始地址。在控制从 m 转向 q 的前一刻，SP 中的值是 620（因为 *msize* 为 20）。随后当 q 调用 p 时，在地址 320 处的指令把 SP 增加到 680，即 p 的活动记录的开始处；当控制返回到 q 的时候，SP 回复到 620。如果接下来的两个对 q 的递归调用立刻返回，那么执行过程中 SP 的最大值就是 680。但是请注意，栈区中被使用的最后的位置是 739，因为从位置 680 开始的 q 的活动记录总共有 60 个字节。 □

8.3.3 名字的运行时刻地址

存储分配策略以及过程的活动记录中局部数据的布局决定了如何访问名字对应的内存位置。在第 6 章，我们假设一个三地址语句中的名字实际上是一个指向该名字的符号表条目的指针。这个方法有一个极大的好处，它使得编译器更加易于移植，因为即使当编译器被移植到使用不同运行时刻组织方式的其他机器时，其前端也不需要修改。但是从另一个方面来看，在生成中间代码时生成特定的访问步骤对于一个优化编译器也有极大的好处，因为这使得优化器能够利用原本在简单的三地址语句中不可见的细节。

在任何一种情况下，名字最终必须被替代为访问存储位置的代码。在这里，我们考虑简单的三地址拷贝语句 x = 0 的一些细节。假设在处理完一个过程的声明部分后，x 的符号表条目包含了 x 的相对地址 12。如果 x 被分配在一个从地址 *static* 开始的静态分配区域中，那么 x 的实际运行时刻地址是 *static* + 12。虽然编译器最终可以在编译时刻确定 *static* + 12 的值，但是在生成访问该名字的中间代码时可能还不知道静态区域的位置。在这种情况下，生成“计算”*static* + 12 的三地址代码是有意义的。当然我们要理解，这个计算在程序运行之前就会完成：它或者在代码生成阶段完成，或者由加载器完成。那么，赋值语句 x = 0 被翻译成

```
static[12] = 0
```

如果静态区从地址 100 开始，这个语句的目标代码是

```
LD 112, #0
```

8.3.4 8.3 节的练习

练习 8.3.1：假设使用栈式分配而寄存器 SP 指向栈的顶端，为下列的三地址语句生成代码。

```
call p
call q
return
call r
return
return
```

练习 8.3.2：假设使用栈式分配而寄存器 SP 指向栈的顶端，为下列的三地址语句生成代码。

1) `x = 1`

2) `x = a`

3) `x = a + 1`

4) `x = a + b`

5) 两个语句的序列

```
x = b * c
y = a + x
```

练习 8.3.3：假设使用栈式分配，且假设 a 和 b 都是元素大小为 4 字节的数组，再次为下面的三地址语句生成代码。

1) 四个语句的序列

```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

2) 三个语句的序列

```
x = a[i]
y = b[i]
z = x * y
```

3) 三个语句的序列

```
x = a[i]
y = b[x]
a[i] = y
```

8.4 基本块和流图

本节介绍一种用图来表示中间代码的方法。即使这个图没有显式地被代码生成算法生成，它对于讨论代码生成也是有帮助的。上下文信息有助于更好地生成代码。正如我们将在 8.8 节看到的，如果我们知道程序中的值是如何被定值和使用的，我们就可以更好地分配寄存器。我们还将在 8.9 节看到，通过检查三地址语句序列，我们可以更好地完成指令选择工作。

这个表示方法可以按照如下方法构造：

1) 把中间代码划分成为*基本块*(basic block)。每个基本块是满足下列条件的最大的连续三地址指令序列。

① 控制流只能从基本块中的第一个指令进入该块。也就是说，没有跳转到基本块中间的转移指令。

② 除了基本块的最后一个指令，控制流在离开基本块之前不会停机或者跳转。

2) 基本块形成了*流图*(flow graph)的结点。而流图的边指明了哪些基本块可能紧随一个基本块之后运行。

从第 9 章开始，我们将讨论在流图上的多种转换。这些转换把原有的中间代码转换成为“优化后”的中间代码，而从“优化后”的中间代码可以生成更好的目标代码。将“优化后”的中间代码转换为目标机器代码的工作将使用本章中的代码生成技术完成。

中断的影响

有人认为，只要控制流到达基本块的开始处就必然会继续执行到基本块结束处，但是这个说法需要一些仔细的考虑。有很多原因会导致一个中断使得控制流离开基本块，甚至可能不再返回，但这些中断并没有在代码中显式地反映出来。比如，一个像 `x = y/z` 这样的指令看起来不影响控制流。但是如果 z 是 0，此指令实际上可能使程序异常中止。

我们用不着担心这种可能性。理由如下：构造基本块的目的是优化代码。一般来说，当一个中断发生时，它要么被适当处理然后将控制返回到引起中断的指令，就好像控制流从来没有离开过；要么程序会中止并报错。在后一种情况下，即使我们在优化时假设控制流会一直到达基本块的结尾，优化的结果也不会有错，因为程序本来就不会给出预计的结果。

8.4.1 基本块

我们的第一项工作是把一个三地址指令序列分割成为基本块。我们以第一个指令作为一个新基本块的开始，然后不断把后续的指令加进去，直到我们碰到一个无条件跳转、条件跳转指令或者下一个指令前面的标号为止。当没有跳转和标号时，控制流直接从一个指令到达下一个指令。这个想法在下面的算法中形式化地表示出来。

算法 8.5 把三地址指令序列划分成为基本块。

输入：一个三地址指令序列。

输出：输入序列对应的一个基本块列表，其中每个指令恰好被分配给一个基本块。

方法：首先，我们确定中间代码序列中哪些指令是首指令(leader)，即某个基本块的第一个指令。跟在中间程序末端之后的指令的不包含在首指令集合中。选择首指令的规则如下：

1）中间代码的第一个三地址指令是一个首指令。

2）任意一个条件或无条件转移指令的目标指令是一个首指令。

3）紧跟在一个条件或无条件转移指令之后的指令是一个首指令。

然后，每个首指令对应的基本块包括了从它自己开始，直到下一个首指令(不含)或者中间程序的结尾指令之间的所有指令。 □

```
 1)  i = 1
 2)  j = 1
 3)  t1 = 10 * i
 4)  t2 = t1 + j
 5)  t3 = 8 * t2
 6)  t4 = t3 - 88
 7)  a[t4] = 0.0
 8)  j = j + 1
 9)  if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

图 8-7 把一个 10×10 的矩阵设置成单位矩阵的中间代码

例 8.6 图 8-7 中的中间代码把一个 10×10 的矩阵 a 设置成一个单位矩阵。这段代码来自哪里并不重要，它也许是从图 8-8 的伪代码中翻译得到的。在生成这个中间代码的时候，我们假设每一个实数值的数组元素占 8 个字节，且矩阵 a 按行存放。

```
for i from 1 to 10 do
        for j from 1 to 10 do
                a[i, j] = 0.0;
for i from 1 to 10 do
        a[i, i] = 1.0;
```

图 8-8 图 8-7 的源代码

首先，根据算法 8.5 的规则(1)可知第一个指令是一个首指令。为了找到其他的首指令，我们要找到跳转指令。在这个例子中有三个跳转指令(全部是条件跳转指令)，即指令 9、11 和 17。根据规则(2)，这些跳转指令的目标是首指令，它们分别是指令 3、2 和 13。然后，根据规则(3)，跟在一个跳转指令后面的每个指令都是首指令，即指令 10 和 12。注意，在这段代码里没有跟在指令 17 后面的指令。假如有的话，那么第 18 个指令也是一个首指令。

我们可以得出结论：指令 1、2、3、10、12 和 13 是首指令。每个首指令对应的基本块包括了从它开始直到下一个首指令之前的所有指令。因此，指令 1 的基本块就是指令 1，指令 2 的基本块是指令 2。但首指令 3 的基本块包含了从指令 3 到指令 9 的所有指令。指令 10 的基本块是 10 和 11；指令 12 的基本块仅仅包含指令 12，而指令 13 的基本块是指令 13 到 17。 □

8.4.2 后续使用信息

知道一个变量的值接下来会在什么时候使用对于生成良好的代码是非常重要的。如果一个变量的值当前存放在一个寄存器中，且之后一直不会被使用，那么这个寄存器就可以被分派给另一个变量。

在一个三地址语句中对一个名字的使用(use)的定义如下。假设三地址语句 i 给 x 赋了一个值。如果语句 j 的一个运算分量为 x，并且从语句 i 开始可以通过未对 x 进行赋值的路径到达语句 j，那么我们说语句 j 使用了在语句 i 处计算得到的 x 的值。我们可以进一步说 x 在语句 i 处活跃(live)。

对每个类似于 $x=y+z$ 的三地址语句，我们希望确定对 x、y 和 z 的下一次使用是什么。当前我们不考虑在包含本三地址语句的基本块之外的使用。

我们用来确定活跃性和后续使用信息的算法对每个基本块进行一次反向的遍历。我们把得到的信息存放到符号表中。使用算法 8.5 中给出的方法，我们可以很容易地通过扫描一个三地址语句流找到各个基本块的结尾。因为过程可能有副作用，为方便起见，我们假设每一个过程调用指令是一个新的基本块的开始。

算法 8.7 对一个基本块中的每一个语句确定活跃性与后续使用信息。

输入： 一个三地址语句的基本块 B，我们假设在开始的时候符号表显示 B 中的所有非临时变量都是活跃的。

输出： 对于 B 的每一个语句 i：$x=y+z$，我们将 x、y 及 z 的活跃性信息及后续使用信息关联到 i。

方法： 我们从 B 的最后一个语句开始，反向扫描到 B 的开始处。对于每个语句 i：$x=y+z$，我们做下面的处理：

1) 把在符号表中找到的有关 x、y 和 z 的当前后续使用和活跃性信息与语句 i 关联起来。

2) 在符号表中，设置 x 为“不活跃”和“无后续使用”。

3) 在符号表中，设置 y 与 z 为“活跃”，并把它们的下一次使用设置为语句 i。

在这里，我们使用 + 作为代表任意运算符的符号。如果三地址语句 i 形如 $x=+y$ 或者 $x=y$，那么处理步骤依然和上面相同，只是忽略了对 z 的处理。注意，步骤(2)和步骤(3)的顺序不能颠倒，因为 x 可能就是 y 或者 z。 □

8.4.3 流图

当将一个中间代码程序划分成为基本块之后，我们用一个流图来表示它们之间的控制流。流图的结点就是这些基本块。从基本块 B 到基本块 C 之间有一条边当且仅当基本块 C 的第一个指令可能紧跟在 B 的最后一个指令之后执行。存在这样一条边的原因有两种：

- 有一个从 B 的结尾跳转到 C 的开头的条件或无条件跳转语句。
- 按照原来的三地址语句序列中的顺序，C 紧跟在 B 之后，且 B 的结尾不存在无条件跳转语句。

我们说 B 是 C 的前驱(predecessor)，而 C 是 B 的一个后继(successor)。

我们通常会增加两个分别称为“入口”(entry)和“出口”(exit)的结点。它们不和任何可执行的中间指令对应。从入口到流图的第一个可执行结点(即包含了中间代码的第一个指令的基本块)有一条边。从任何包含了可能是程序的最后执行指令的基本块到出口有一条边。如果程序的最后指令不是一个无条件转移指令，那么包含了程序的最后一条指令的基本块是出口结点的一个前驱。但任何包含了跳转到程序之外的跳转指令的基本块也是出口结点的前驱。

例 8.8 从例 8.6 中构造出的基本块可以生成图 8-9 中所示的流图。入口结点指向基本块 B_1，因为 B_1 包含了这个程序的第一个指令。B_1 的唯一后继是 B_2，因为 B_1 的结尾不是一个无条件跳转指令，且 B_2 的首指令紧跟在 B_1 的结尾指令之后。

基本块 B_3 有两个后继。其中的一个是它本身，因为 B_3 的首指令(即指令 3)是 B_3 结尾处的条件跳转指令(即指令 9)的目标。另一个后继是 B_4，因为控制流可能穿越 B_3 结尾处的条件跳转指令而到达 B_4 的首指令。

只有 B_6 指向流图的出口结点，因为到达紧跟在流图对应的程序之后的代码的唯一方式是穿越 B_6 结尾处的条件跳转指令。□

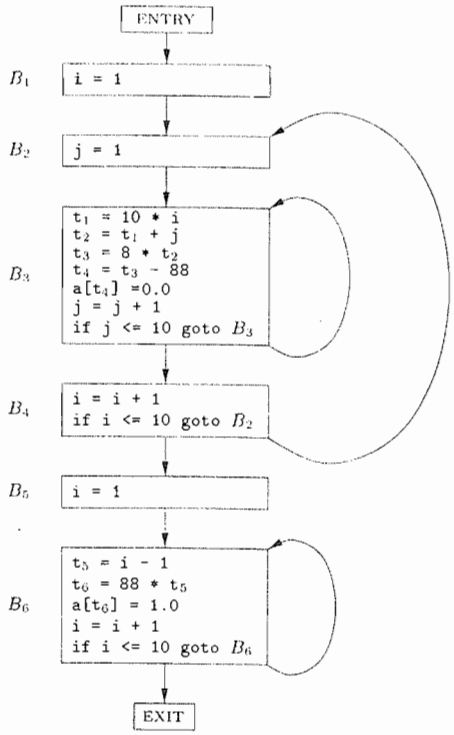


图 8-9 基于图 8-7 构造的流图

8.4.4 流图的表示方式

首先，从图 8-9 中可以看出，在流图里面把到达指令的序号或标号的跳转指令替换为到达基本块的跳转，这么做是很正常的。回忆一下，所有条件或无条件跳转指令总是跳转到某些基本块的首指令，而现在这些跳转指令指向了相应的基本块。这么做的原因是，在流图构造完成之后经常会对多个基本块中的指令做出实质性的改变。如果跳转的目标是指令，我们将不得不在每次改变了某个目标指令之后修正跳转指令的目标。

流图就是通常的图，它可以用任何适合表示图的数据结构来表示。结点(即基本块)的内容需要有它们自己的表示方式。我们可以用一个指向该基本块在三地址指令数组中的首指令的指针，再加上基本块的指令数量或一个指向结尾指令的指针来表示结点的内容。但是，因为我们可能会频繁改变一个基本块中的指令数量，所以为每个基本块创建一个指令链表是一种高效的表示方法。

8.4.5 循环

像 while 语句、do-while 语句和 for 语句这样的程序设计语言构造自然地把循环引入到程序中。因为事实上每个程序会花很多时间执行循环，所以对于一个编译器来说，为循环生成优良的代码就变得非常重要。很多代码转换依赖于对流图中“循环”的识别。如果下列条件成立，我们就说流图中的一个结点集合 L 是一个循环。

1) 在 L 中有一个被称为*循环入口*(loop entry)的结点，它是唯一的其前驱可能在 L 之外的结点。也就是说，从整个流图的入口结点开始到 L 中的任何结点的路径都必然经过循环入口结点，并且这个循环入口结点不是整个流图的入口结点本身。

2) L 中的每个结点都有一个到达 L 的入口结点的非空路径，并且该路径全部在 L 中。

例 8.9 图 8-9 中的流图有三个循环：

1) B_3 自身

2) B_6 自身

3) $\{B_2, B_3, B_4\}$

其中的前两个循环都由单一结点组成，这些结点都有到其自身的边。比如，B_3 形成一个以 B_3 本身为入口结点的循环。请注意，循环的第二个条件要求有一个从 B_3 到本身的非空路径。因此，像 B_2 这样的单一结点(它没有一条 $B_2 \rightarrow B_2$ 的边)不是循环，因为没有从 B_2 到其自身，且在集合 $\{B_2\}$ 中的非空路径。

第三个循环 $L = \{B_2, B_3, B_4\}$ 的循环入口结点是 B_2。请注意，这三个结点中只有 B_2 有一个不在 L 中的前驱 B_1。而且，这三个结点中都有在 L 中且到达 B_2 的非空路径。比如，从 B_2 开始就有路径 $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2$。 □

8.4.6 8.4 节的练习

练习 8.4.1：图 8-10 是一个简单的矩阵乘法程序。

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0.0;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

图 8-10 一个矩阵相乘算法

1) 假设矩阵的元素是需要 8 个字节的数值，而且矩阵按行存放。把程序翻译成为我们在本节中一直使用的那种三地址语句。

2) 为(1)中得到的代码构造流图。

3) 找出在(2)中得到的流图的循环。

练习 8.4.2：图 8-11 中是计算从 2 ~ n 之间素数个数的代码。它在一个适当大小的数组 a 上使用筛法来完成计算。也就是说，最后 $a[i]$ 为真仅当没有小于等于 $\sqrt{i}$ 的质数可以整除 i。我们一开始把所有的 $a[i]$ 初始化为 TRUE；如果我们找到了 j 的一个因子，就把 $a[j]$ 设置为 FALSE。

1) 把程序翻译成为我们在本节中使用的那种三地址语句序列。这里假设一个整数需要 4 个字节存放。

2) 为在(1)中得到的代码构造流图。

3) 找出在(2)中得到的流图的循环。

```
for (i=2; i<=n; i++)
    a[i] = TRUE;
count = 0;
s = sqrt(n);
for (i=2; i<=s; i++)
    if (a[i]) /* 已知i是一个素数*/ {
        count++;
        for (j=2*i; j<=n; j = j+i)
            a[j] = FALSE; /* i的倍数都不是素数*/
    }
```

图 8-11 筛法选取素数的代码

8.5 基本块的优化

仅仅通过对各个基本块本身进行局部优化，我们就常常可以实质性地降低代码运行所需的时间。更加彻底的全局优化将从第 9 章开始讨论。全局优化将检查信息是如何在一个程序的多个基本块之间流动的。全局优化是一个很复杂的主题，它将考虑很多不同的技术。

8.5.1 基本块的 DAG 表示

很多重要的局部优化技术首先把一个基本块转换成为一个 DAG(有向无环图)。在 6.11 节

中，我们介绍了用于表示简单表达式的 DAG。这个想法被自然地扩展到在一个基本块中创建的表达式的集合。我们按照如下方式为一个基本块构造 DAG：

1）基本块中出现的每个变量有一个对应的 DAG 的结点表示其初始值。

2）基本块中的每个语句 s 都有一个相关的结点 N。N 的子结点是基本块中的其他语句的对应结点。这些语句是在 s 之前、最后一个对 s 所使用的某个运算分量进行定值的语句。⊖

3）结点 N 的标号是 s 中的运算符；同时还有一组变量被关联到 N，表示 s 是在此基本块内最晚对这些变量进行定值的语句。

4）某些结点被指明为*输出结点*(output node)。这些结点的变量在基本块的*出口处活跃*。也就是说，这些变量的值可能以后会在流图的另一个基本块中被使用到。计算得到这些“活跃变量”是全局数据流分析的问题，将在 9.2.5 节中讨论。

基本块的 DAG 表示使我们可以对基本块所代表的代码进行一些转换，以改进代码的质量。

1）我们可以消除*局部公共子表达式*(local common subexpression)。所谓公共子表达式就是重复计算一个已经计算得到的值的指令。

2）我们可以消除*死代码*(dead code)，即计算得到的值不会被使用的指令。

3）我们可以对相互独立的语句进行重新排序，这样的重新排序可以降低一个临时值需要保持在寄存器中的时间。

4）我们可以使用代数规则来重新排列三地址指令的运算分量的顺序。这么做有时可以简化计算过程。

8.5.2 寻找局部公共子表达式

检测公共子表达式的方法是这样的。当一个新的结点 M 将被加入到 DAG 中时，我们检查是否存在一个结点 N，它和 M 具有同样的运算符和子结点，且子结点顺序相同。如果存在这样的结点，N 计算的值和 M 计算的值是一样的，因此可以用 N 替换 M。在 6.1.1 节中，这个技术被称为检测公共子表达式的“值编码”方法。

例 8.10 下面的基本块的 DAG 见图 8-12。

```
a = b + c
b = a - d
c = b + c
d = a - d
```

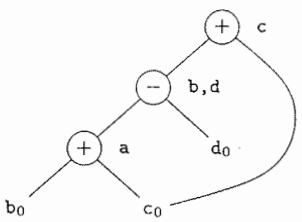


图 8-12 例 8.10 中的基本块的 DAG

当我们为第三个语句 c = b + c 构造结点的时候，我们知道 b + c 中 b 的使用指向图 8-12 中标号为 - 的结点。因为这个结点是 b 的最近的定值。因此，我们不会把语句 1 和语句 3 所计算的值混淆。

然而，对应于第四个语句 d = a - d 的结点的运算符是 -，且它的子结点是标记有变量 a 和 d_0 的结点。因为运算符和子结点都和语句 2 对应的结点相同，我们不需要创建这个结点，而是把 d 加到这个标记为 - 的结点的定值变量表中。 □

因为在图 8-12 的 DAG 中只有三个非叶子结点，看起来例 8.10 中的基本块可以替换为一个只有三个语句的基本块。实际上，假如 b 在这个基本块的出口点不活跃，我们不需要计算变量 b，可以使用 d 来存放图 8-12 中标号为 - 的结点所代表的值。这个基本块就变成了：

⊖ 原文如此。如果 s 的某个运算分量在基本块内没有在 s 之前被定值，那么这个运算分量对应的子结点就是代表该运算分量的初始值的结点。——译者注

```
a = b + c
d = a - d
c = d + c
```

但是，如果 b 和 d 都在出口处活跃，我们就必须使用第四个语句把值从一个变量复制到另一个。[⊖]

例 8.11 当我们寻找公共子表达式的时候，我们实际上是寻找不管如何计算一定能得到相同结果值的表达式。因此，DAG 方法不能看到下面的事实，即下面的语句序列

```
a = b + c
b = b - d
c = c + d
e = b + c
```

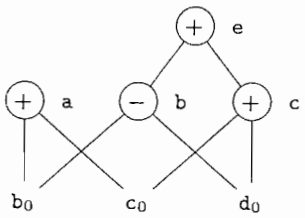


图 8-13 例 8.11 中的基本块的 DAG

中，第一和第四个语句实际上计算的是同一个表达式的值，即 $b_0 + c_0$。也就是说，虽然 b 和 c 在第一个和第四个语句之间改变了，但它们的和仍保持不变，因为 $b + c = (b - d) + (c + d)$。这个序列的 DAG 见图 8-13。它没有显示出任何公共子表达式。但是，如 8.5.4 节中将要讨论的，在 DAG 中应用代数恒等式可以揭示出这样的等值关系。 □

8.5.3 消除死代码

在 DAG 上消除死代码的操作可以按照如下方式实现。我们从一个 DAG 上删除所有没有附加活跃变量的根结点(即没有父结点的结点)。重复应用这样的处理过程就可以从 DAG 中消除所有对应于死代码的结点。

例 8.12 如果图 8-13 中的 a 和 b 是活跃变量，而 c 和 e 不是，我们可以立刻消除标记为 e 的根结点。然后标记为 c 的结点就变成根结点，也可以被删除。标记为 a 和 b 的结点被保留下来，因为它们都附有活跃变量。 □

8.5.4 代数恒等式的使用

代数恒等式表示基本块的另一类重要的优化方法。比如，我们可以使用诸如

$$x + 0 = 0 + x = x \qquad x - 0 = x$$
$$x \times 1 = 1 \times x = x \qquad x/1 = x$$

这样的恒等式来从一个基本块中消除计算步骤。

另一类代数优化是局部强度消减(reduction in strength)，就是把一个代价较高的运算替换为一个代价较低的运算。比如：

代价较高的		代价较低的
x^2	=	$x \times x$
$2 \times x$	=	$x + x$
$x/2$	=	$x \times 0.5$

第三种相关的优化是常量合并(constant folding)。使用这种方法时，我们在编译时刻对常量表达式求值，并把此常量表达式替换为求出的值[⊜]。因此，表达式 2 * 3.14 可以被替换为 6.28。

⊖ 总的来说，在从 DAG 生成代码时我们必须非常小心地处理变量的名字。如果变量 x 被定值两次，或者虽然只赋值一次但初始值 x_0 被使用过，那么必须保证不会在原先存放 x 值的结点被全部使用之前改变 x 的值。

⊜ 在编译时刻对算术表达式求值时，必须使用和运行时刻相同的求值方法。K. Thompson 给出了一个很完美的解决方法：对常量表达式进行编译，在目标机上执行目标代码，然后把表达式替换为执行结果。按照这样的做法，编译器就不需要另带一个解析器。

在实践中，因为在程序中频繁使用符号常量，所以会出现常量表达式。

DAG的构造过程可以帮助我们使用这些转换，以及其他的通用代数转换规则，比如交换律和结合律等。比如，假设语言的参考手册确定 * 是可交换的，也就是说，$x*y=y*x$。在创建一个标记为 * 且左右子结点分别是 M 和 N 的新结点时，我们总是检查这样的结点是否已经存在。然而，因为 * 是可交换的，所以我们还应该检查是否存在一个标记为 * 且左右子结点分别是 N 和 M 的结点。

< 和 = 这样的关系运算符有时会产生意料之外的公共子表达式。比如，条件表达式 $x>y$ 也可以通过将参数相减并测试由减法运算设置的条件代码来测试。因此，对 $x-y$ 和 $x>y$，只需要生成一个DAG结点[⊖]。

结合律也可以用于揭示公共子表达式。比如，如果源程序中包含如下的赋值语句：

```
a = b + c;
e = c + d + b;
```

则可能生成下面的中间代码：

```
a = b + c
t = c + d
e = t + b
```

如果 t 没有在基本块之外使用，通过应用 + 的交换律和结合律，我们可以把这个序列改为：

```
a = b + c
e = a + d
```

编译器的设计者应该仔细阅读语言的参考手册，以决定可以重新排列哪些计算。因为计算机算术(因为上溢或下溢等原因)可能不一定遵守数学上的代数恒等式。比如，Fortran语言标准说，编译器可以通过任意数学上等价的表达式来求值，前提是不能违反原来表达式的括号的一致性[⊜]。因此，编译器可以用 $x*(y-z)$ 的方式来计算 $x*y-x*z$，但是它不能以 $(a+b)-c$ 的方式计算 $a+(b-c)$。因此，如果一个Fortran编译器想按照语言的定义来优化程序，它必须跟踪源语言表达式中哪些地方有括号。

8.5.5 数组引用的表示

初看上去，数组下标指令似乎可以像其他的运算那样处理。比如，考虑下列的三地址指令序列：

```
x = a[i]
a[j] = y
z = a[i]
```

如果我们把 a[i] 当作是一个和 $a+i$ 类似的关于 a 和 i 的普通运算，那么 a[i] 的两次使用看起来好像是一个公共子表达式。在这种情况下，我们可能会把第三个指令 z = a[i] 优化为 z = x。然而，因为 j 可能等于 i，中间的语句可能实际上改变了 a[i] 的值。因此，这种优化是不合法的。

在DAG中，表示数组访问的正确方法如下。

1) 从一个数组取值并赋给其他变量的运算(比如 x = a[i])用一个新创建的运算符为 =[] 的结点表示。这个结点的左右子结点分别代表数组初始值(本例中是 a_0)和下标 i。变量 x 是这个结点的标号之一。

2) 对数组的赋值(比如 a[j] = y)用一个新创建的运算符为[] = 的结点来表示。这个结点的三个子结点分别表示 a_0、j 和 y。没有变量用这个结点标号。不同之处在于此结点的创建杀

⊖ 然而，减法运算可能引起上溢或下溢，而比较指令不会引起这个问题。

⊜ 即不能跨越括号求值——译者注。

死了所有当前已经建立的，其值依赖于 a_0 的结点。一个被杀死的结点不可能再获得任何标号。也就是说，它不可能成为一个公共子表达式。

例 8.13 基本块

```
x = a[i]
a[j] = y
z = a[i]
```

的 DAG 见图 8-14。对应于 x 的结点 N 首先被创建，但是当标号为[ ] = 的结点被创建时，N 就被杀死了。因此当 z 的结点被建立时，它不会被认为和 N 等同，而是必须创建一个具有同样的运算分量 a_0 和 i_0 的新结点。 □

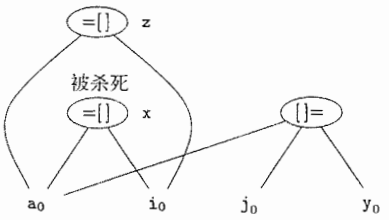


图 8-14 一个数组赋值序列的 DAG

例 8.14 有时即使某个结点的所有子结点都没有像例 8.13 中的 a_0 那样的附加数组变量，它也必须被杀死。类似地，如果一个结点具有数组后代，即使它的子结点都不是数组结点，它也可以杀死别的结点。例如考虑下面的三地址代码

```
b = 12 + a
x = b[i]
b[j] = y
```

这里的情况是，为了效率方面的原因，b 被定值为数组 a 中的一个位置。例如，如果 a 的元素长度是 4 个字节，那么 b 代表了 a 的第四个元素。如果 j 和 i 表示同一个值，那么 b[i] 和 b[j] 代表了同一个位置。因此，很重要的一件事情就是让第三个指令 b[j] = y 杀死带有附加变量 x 的结点。然而，正如我们在图 8-15 中看到的，被杀的结点和杀死被杀结点的结点都把 a_0 作为孙结点，而不是子结点。 □

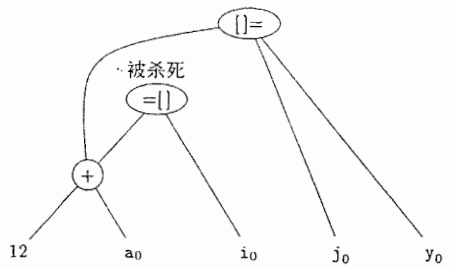


图 8-15 即使没有把一个数组作为子结点，一个结点也可能杀死对该数组的使用

8.5.6 指针赋值和过程调用

当我们像下面的赋值语句

```
x = *p
*q = y
```

那样，通过指针进行间接赋值时，我们并不知道 p 和 q 指向哪里。从效果看，x = *p 是对任意变量的使用，而 *q = y 可能对任意一个变量赋值。其结果是，运算符 =* 必须把当前所有带有附加标识符的结点当作其参数。但是这么做会影响死代码的消除过程。更加重要的是，*= 运算符会把至今为止构造出来的 DAG 中的其他结点全部杀死。

我们可以进行一些全局指针分析，以便把一个指针在代码中某个位置上可能指向的变量限制在一个较小的子集内。即使是局部分析也可以限制一个指针指向的范围。比如，对于下面的序列

```
p = &x
*p = y
```

我们知道是 x（而不是其他变量）被赋予 y 的值。因此，我们只需要杀死以 x 为附加变量的结点，不需要杀死其他结点。

过程调用和通过指针赋值很相似。在缺乏全局数据流信息的情况下，我们必须假设一个过程调用使用和改变了它访问的所有数据。因此，如果变量 x 在一个过程 P 的访问范围之内，对 P 的调用不仅使用了以 x 为附加变量的结点，还杀死了这个结点。

8.5.7 从 DAG 到基本块的重组

对 DAG 的各种优化处理可以在生成 DAG 图时进行，也可以在 DAG 构造完成后通过对 DAG 的运算完成。在完成这些优化处理之后，我们就可以根据优化得到的 DAG 重组生成相应基本块的三地址代码。对每个具有一个或多个附加变量的结点，我们构造一个三地址语句来计算其中某个变量的值。我们倾向于把计算得到的结果赋给一个在基本块出口处活跃的变量。但是，如果我们没有全局活跃变量的信息作为依据，就要假设程序的所有变量都在基本块出口处活跃(但是不包含编译器为了处理表达式而生成的临时变量)。

如果结点有多个附加的活跃变量，我们就必须引入复制语句，以便给每一个变量都赋予正确的值。有时我们可以通过全局优化技术，设法用其中的一两个变量来替代其他变量，从而消除这些复制语句。

例 8.15 回顾一下图 8-12 中的 DAG。在例 8.10 后面的讨论中，我们确定如果 b 在基本块的出口处不活跃，那么下面的三个语句

```
a = b + c
d = a - d
c = d + c
```

就足以重建那个基本块了。第三个指令 c = d + c 必须使用 d 而不是 b 作为运算分量，因为经过优化的基本块不会计算 b 的值。

如果 b 和 d 都在出口处活跃，或者我们不能够确定它们是否在出口处活跃，那么我们还是需要计算 d 和 b 的值。我们可以用下面的序列来完成这个计算：

```
a = b + c
d = a - d
b = d
c = d + c
```

这个基本块仍然比原来的基本块高效。虽然指令数目相同，但我们已经把一个减法替换为一个复制运算。在大多数机器上，复制运算要比减法更加高效。不仅如此，我们还有可能通过全局分析把此基本块外对 b 的使用全部替换为对 d 的使用，从而消除在基本块外对 b 的使用。在这种情况下，我们就可以再次回到这个基本块并消除 b = d。直观地讲，如果在任何使用 b 的这个值的时刻，d 中的值仍然和 b 一样，那么我们就可以消除这个复制运算。这种情况是否成立依赖于程序如何重新计算 d 的值。 □

当从 DAG 重构基本块时，我们不仅要关心用哪些变量来存放 DAG 中的结点的值，还要关心计算不同结点值的指令的顺序。应记住如下规则：

1) 指令的顺序必须遵守 DAG 中的结点的顺序。也就是说，只有在计算出一个结点的各个子结点的值之后，才可以计算这个结点的值。

2) 对数组的赋值必须跟在所有(按照原基本块中的指令顺序)在它之前的对同一数组的赋值或求值运算之后。

3) 对数组元素的求值必须跟在所有(在原基本块中)在它之前的对同一数组的赋值指令之后。对同一数组的两个求值运算可以交换顺序，只要在交换时它们都没有越过某个对同一数组的赋值运算即可。

4) 一个变量的使用必须跟在所有(在原基本块中)在它之前的过程调用和指针间接赋值运算之后。

5) 任何过程调用或者指针间接赋值都必须跟在所有(在原基本块中)在它之前的对任何变量的求值运算之后。

也就是说，当重组代码的时候，没有一个语句可以跨越过程调用或指针间接赋值运算。只有在两个使用同一个数组的指令都是数组访问而不是对数组元素赋值时，它们才可以交换顺序。

8.5.8 8.5 节的练习

练习 8.5.1：为下面的基本块构造 DAG。

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

练习 8.5.2：分别按照下列两种假设简化练习 8.5.1 的三地址代码。

1）只有 a 在基本块的出口处活跃。

2）a、b、c 在基本块的出口处活跃。

练习 8.5.3：为图 8-9 中的块 B_6 的代码构造 DAG。请不要忘记包含比较指令 $i \leqslant 10$。

练习 8.5.4：为图 8-9 中的块 B_3 的代码构造 DAG。

练习 8.5.5：扩展算法 8.7，使之可以处理如下的三地址语句（原文为 three-statements——译者注）

1）`a[i] = b`

2）`a = b[i]`

3）`a = *b`

4）`*a = b`

练习 8.5.6：分别按照下面的两个假设，为基本块

```
a[i] = b
*p = c
d = a[j]
e = *p
*p = a[i]
```

构造 DAG 图。假设如下：

1）`p` 可以指向任何地方。

2）`p` 只能指向 `b` 或 `d`。

！**练习 8.5.7**：如果一个指针或数组表达式（比如 `a[i]`或者 `*p`）被赋值之后又被使用，且赋值和使用之间没有做任何修改，我们就可以利用这种情况来简化 DAG。比如，在练习 8.5.6 的代码中，因为 `p` 可能指向的所有位置在第二个和第四个语句之间没有被赋值，所以不管 `p` 指向哪里，语句 `e = *p` 都可以被替换为 `e = c`。请修正 DAG 构造算法以利用这种情况带来的好处，并把你的算法应用到练习 8.5.6 的代码中。

练习 8.5.8：假设一个基本块由下面的 C 语言赋值语句生成：

```
x = a + b + c + d + e + f;
y = a + c + e;
```

1）给出这个基本块的三地址语句（每个语句只做一次加法）。

2）假设 `x` 和 `y` 都在基本块的出口处活跃，利用加法的结合律和交换律来修改这个基本块，使得指令个数最少。

8.6 一个简单的代码生成器

在本节中，我们将考虑一个为单个基本块生成代码的算法。它依次考虑各个三地址指令，并跟踪记录哪个值存放在哪个寄存器中。这样可以避免生成不必要的加载和保存指令。

在代码生成中的主要问题之一是决定如何最大限度地利用寄存器。寄存器有如下四种主要使用方法：

- 在大部分机器的体系结构中，执行一个运算时该运算的部分或全部运算分量必须存放在寄存器中。

- 寄存器很适合做临时变量，即在计算一个大表达式时存放其子表达式的值。或者更一般地讲，寄存器适合用于存放只在单个基本块内使用的变量的值。
- 寄存器用来存放在一个基本块中计算而在另一个基本块中使用的(全局)值。比如，循环下标的值，每次循环都对该值作增量运算，并在循环体中多次被使用。
- 寄存器经常用来帮助进行运行时刻的存储管理。比如，管理运行时刻栈包括栈指针的维护，栈顶元素也可能被存放在寄存器中。

因为可用寄存器的数量是有限的，这些需求之间有相互竞争的关系。

本节的算法假设有一组寄存器可以用来存放在基本块内使用的值。通常情况下，这个寄存器集合不包括机器的所有寄存器，因为有些寄存器专门用于存放全局变量或者用于对栈进行管理。我们假设基本块已经通过诸如公共子表达式合并这样的转换而变成了我们希望的三地址指令序列。我们进一步假设对每个运算符有且只有一个对应的机器指令。这个指令对存放在寄存器中的所需的运算分量进行运算，并把结果存放在一个寄存器中。机器指令的形式如下：

- LD *reg*, *mem*
- ST *mem*, *reg*
- OP *reg*, *reg*, *reg*

8.6.1 寄存器和地址描述符

我们的代码生成算法依次考虑了各个三地址指令，并决定需要哪些加载指令来把必需的运算分量加载进寄存器。在生成加载指令之后，它开始生成运算代码。然后，如果有必要把结果存放入一个内存位置，它还会生成相应的保存指令。

为了做出这些必要的决定，我们需要一个数据结构来说明哪些程序变量的值当前被存放在哪个或哪些寄存器里面。我们还需要知道当前存放在一个给定变量的内存位置上的值是否就是这个变量的正确值。因为变量的新值可能已经在寄存器中计算出来但还没有存放到内存中。这个数据结构具有下列描述符：

1）每个可用的寄存器都有一个*寄存器描述符*(register descriptor)。它用来跟踪有哪些变量的当前值存放在此寄存器内。因为我们仅仅考虑那些用于存放一个基本块内的局部值的寄存器，我们可以假设在开始时所有的寄存器描述符都是空的。随着代码生成过程的进行，每个寄存器将存放零个或多个变量名字的值。

2）每一个程序变量都有一个*地址描述符*(address descriptor)。它用来跟踪记录在哪个或哪些位置上可以找到该变量的当前值。这个位置可以是一个寄存器、一个内存地址、一个栈中的位置，也可以是由这些位置组成的一个集合。这个信息可以存放在这个变量名字对应的符号表条目中。

8.6.2 代码生成算法

这个算法的一个重要部分是函数 *getReg*(*I*)。这个函数为每个与三地址指令 *I* 有关的内存位置选择寄存器。函数 *getReg* 可以访问这个基本块的所有变量对应的寄存器和地址描述符。这个函数还可能需要获取一些有用的数据流信息，比如哪些变量在基本块出口处活跃。我们将首先给出基本算法，然后再讨论 *getReg* 函数。我们不知道总共有多少个寄存器可用于存放基本块的局部数据，因此假设有足够的寄存器使得在把值存放回内存，释放了所有的可用寄存器之后，空闲的寄存器足以完成任何三地址运算。

在一个形如 `x = y + z` 的三地址指令中，我们将把 + 当作一般的运算符，而 ADD 当作等价的机器指令。因此，我们没有利用 + 的交换性。这样，当我们实现这个运算时，y 的值必须在 ADD 指令中给出的第二个寄存器中，而绝不会是第三个寄存器。可以按照下面的方法来改进算法：只

要 + 是一个满足交换律的运算符，算法同时为 x = y + z 和 x = z + y 生成代码；随后再选择一个比较好的代码序列。

运算的机器指令

对每个形如 $x = y + z$ 的三地址指令，完成下列步骤：

1）使用 *getReg*($x = y + z$)来为 x、y、z 选择寄存器。我们把这些寄存器称为 R_x、R_y 和 R_z。

2）如果(根据 R_y 的寄存器描述符)y 不在 R_y 中，那么生成一个指令“LD R_y, y'”，其中 y'是存放 y 的内存位置之一((y'可以根据 y 的地址描述符得到)。

3）类似地，如果 z 不在 R_z 内，生成一个指令“LD R_z, z'”，其中 z'是存放 z 的位置之一。

4）生成指令“ADD R_x, R_y, R_z”。

复制语句的机器指令

形如 $x = y$ 的三地址指令是一个重要的特例。我们假设 *getReg* 总是为 x 和 y 选择同一个寄存器。如果 y 没有在寄存器 R_y 中，那么生成机器指令 LD R_y, y。如果 y 已经在 R_y 中，我们不需要做任何事情。我们只需要修改 R_y 的寄存器描述符，表明 R_y 中也存放了 x 的值。

基本块的收尾处理

我们描述算法时表明，在代码结束的时候，基本块中使用的变量可能仅存放在某个寄存器中。如果这个变量是一个只在基本块内部使用的临时变量，那就没有问题；当基本块结束时，我们可以忘记这些临时变量的值并假设这些寄存器是空的。但如果一个变量在基本块的出口处活跃，或者我们不知道哪些变量在出口处活跃，那么就必须假设这个变量的值会在以后被用到。在那种情况下，对于每个变量 x，如果它的地址描述符表明它的值没有存放在 x 的内存位置上，我们必须生成指令 ST x, R，其中 R 是在基本块的结尾处存放 x 值的寄存器。

管理寄存器和地址描述符

当代码生成算法生成加载、保存和其他机器指令时，它必须同时更新寄存器和地址描述符。修改的规则如下：

1）对于指令“LD R, x”：

① 修改寄存器 R 的寄存器描述符，使之只包含 x。

② 修改 x 的地址描述符，把寄存器 R 作为新增位置加入到 x 的位置集合中。

③ 从任何不同于 x 的变量的地址描述符中删除 R。(原文缺一条——译者注。)

2）对于指令 ST x, R，修改 x 的地址描述符，使之包含自己的内存位置。

3）对于实现三地址指令 $x = y + z$ 的“ADD R_x, R_y, R_z”这样的运算而言：

① 改变 R_x 的寄存器描述符，使之只包含 x。

② 改变 x 的地址描述符使得它只包含位置 R_x。注意，现在 x 的地址描述符中不包含 x 的内存位置。

③ 从任何不同于 x 的变量的地址描述符中删除 R_x。

4）当我们处理复制语句 $x = y$ 时，如果有必要生成把 y 加载入 R_y 的加载指令，那么在生成加载指令并(按照规则 1)像处理所有的加载指令那样处理完各个描述符之后，再进行下面的处理：

① 把 x 加入到 R_y 的寄存器描述符中。

② 修改 x 的地址描述符，使得它只包含唯一的位置 R_y。

例 8.16 让我们把由下列三地址语句组成的基本块翻译成代码。

```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

这里，我们假设 t、u、v 都是基本块的局部临时变量，而变量 a、b、c、d 在基本块出口处活跃。因为我们还没有讨论函数 *getReg* 是如何工作的，所以将简单地假设当需要时总有足够的寄存器可用。但是当一个寄存器中存放的值不再有用时(比如，它只存放了一个临时变量的值，且对这个临时变量的所有使用都已经处理完了)，我们就复用这个寄存器。

图 8-16 显示了算法生成的所有机器代码指令。该图还显示了在翻译每个三地址指令之前和之后的寄存器和地址描述符的情况。

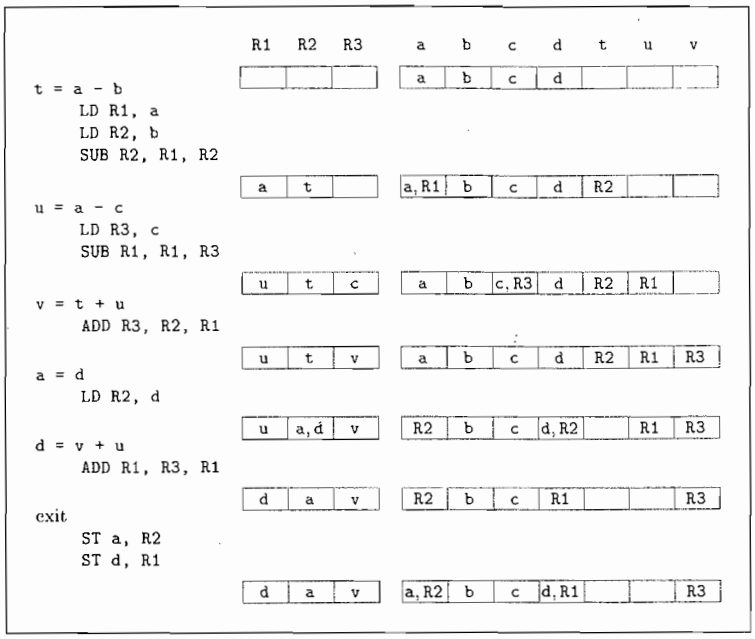


图 8-16　生成的指令以及寄存器和地址描述符的改变过程

因为最初寄存器中不保存任何值，我们需要为第一个三地址指令 t = a - b 生成三个指令。因此，我们看到 a 和 b 被加载到寄存器 R1 和 R2 中，而 t 的值生成后存放于寄存器 R2 中。注意，我们可以使用 R2 来存放 t 是因为原先存放于 R2 中的 b 的值在该基本块内不再被使用。因为预设了 b 在基本块的出口处活跃，假如(b 的地址描述符表明)b 不在它自己的内存位置上，那么我们将不得不先把 R2 中的值保存到 b。假如我们需要 R2，那么生成指令 ST b R2 的决定将由 getReg 做出。

第二个指令 u = a - c 不需要加载 a 的指令，因为 a 已经存放在寄存器 R1 中。原来存放在寄存器 R1 中的 a 的值在该基本块中不再被用到，而且如果在基本块之外需要使用 a 的值，可以从 a 的内存位置上获取(因为 a 的值也在它自己的内存位置上)。因此，我们还可以复用 R1 来存放结果 u。请注意，我们改变了 a 的地址描述符，以表明它已经不在 R1 中，但是还在称为 a 的内存位置中。

第三个指令 v = t + u 只需要一个加法指令。而且，我们可以用 R3 来存放结果 v，因为原先存放在该寄存器中的 c 的值在该基本块内不再使用，且 c 在自己的内存位置上也存放了这个值。

复制指令 a = d 需要一个指令来加载 d，因为 d 不在寄存器中。图中显示寄存器 R2 的描述符包含了 a 和 b。把 a 加入到寄存器描述符是我们处理这个复制语句的结果，而不是任何机器指令的结果。

第五个指令 d = v + u 使用两个存放在寄存器中的值。因为 u 是一个临时变量且它的值不再被使用，所以我们选择复用它的寄存器 R1 来存放 d 的新值。请注意，d 现在只存放在 R1 中，不在它自己的内存位置上。对于 a 也是同样的情况，a 的值只存放在 R2 中，而不在被称为 a 的内存位置上。因为这个原因，我们需要为基本块的机器代码增加一个“尾声”：它把在出口处活跃的变量 a 和 d 的值保存回它们的内存位置。这就是图中的最后两个指令的工作。 □

8.6.3 函数 getReg 的设计

最后，让我们考虑如何针对一个三地址指令 I 实现函数 $getReg(I)$。实现这个函数可以选择很多种方法，当然也存在一些绝对不可以选择的方法。这些错误方法会因丢失一个或多个活跃变量的值而导致生成错误代码。我们用处理一个运算指令的步骤来开始我们的讨论，还是用 $x = y + z$ 作为一般性的例子。首先，我们必须为 y 和 z 分别选择一个寄存器。这两次选择所面临的问题是相同的，因此我们将集中考虑为 y 选择寄存器 R_y 的方法。选择规则如下：

1）如果 y 当前就在一个寄存器中，则选择一个已经包含了 y 的寄存器作为 R_y。不需要生成一个机器指令来把 y 加载到这个寄存器。

2）如果 y 不在寄存器中，但是当前存在一个空寄存器，那么选择这个空寄存器作为 R_y。

3）比较困难的情况是 y 不在寄存器中且当前也没有空寄存器。无论如何，我们需要选择一个可行的寄存器，并且必须保证复用这个寄存器是安全的。设 R 是一个候选寄存器，且假设 v 是 R 的寄存器描述符表明的已位于 R 中的变量。我们需要保证要么 v 的值已经不会被再次使用，要么我们还可以到别的地方获取 v 的值。可能的情况包括：

① 如果 v 的地址描述符说 v 还保存在 R 之外的其他地方，我们就完成了任务。

② 如果 v 是 x，即由指令 I 计算的变量，且 x 不同时是指令 I 的运算分量之一（比如这个例子中的 z），那么我们就完成了任务。其原因是在这种情况下，我们知道 x 的当前值决不会再次被使用，因此我们可以忽略它。

③ 否则，如果 v 不会在此之后被使用（即在指令 I 之后不会再次使用 v，且如果 v 在基本块的出口处活跃，那么 v 的值必然在基本块中被重新计算），那么我们就完成了任务。

④ 如果前面的三个条件都不满足，我们就需要生成保存指令 ST v, R 来把 v 的值复制到它自己的内存位置上去。这个操作称为*溢出操作*（spill）。

因为在那个时刻 R 可能存放了多个变量的值，所以我们需要对每个这样的变量 v 重复上述步骤。最后，R 的“得分”是我们需要生成的保存指令的个数。选择一个具有最低得分的寄存器（或之一）。

现在考虑寄存器 R_x 的选择。其中的难点和可选项几乎和选择 R_y 时的一样，因此我们只给出其中的区别。

1）因为 x 的一个新值正在被计算，因此只存放了 x 的值的寄存器对 R_x 来说总是可接受。即使 x 就是 y 或 z 之一，这个语句仍然成立，因为我们的机器指令允许一个指令中的两个寄存器相同。

2）如果（像上面对变量 v 的描述那样）y 在指令 I 之后不再使用，且（在必要时加载 y 之后）R_y 仅仅保存了 y 的值，那么 R_y 同时也可以用作 R_x。对 z 和 R_z 也有类似选择。

需要特别考虑的最后一个问题是当 I 是复制指令 x = y 时的情况。我们用上面描述的方法选择 R_y，然后是让 $R_x = R_y$。

8.6.4 8.6 节的练习

练习 8.6.1：为下面的每个 C 语言赋值语句生成三地址代码

1) `x = a + b*c;`
2) `x = a/(b+c) - d*(e+f);`
3) `x = a[i] + 1;`
4) `a[i] = b[c[i]];`
5) `a[i][j] = b[i][k] + c[k][j];`
6) `*p++ = *q++;`

假设其中的所有数组元素都是整数，每个元素占四个字节。在4和5部分，假设a、b、c是常数。和在本章之前有关数组访问的例子中一样，它们给出了同名数组的第0个元素的位置。

！练习8.6.2：假设数组a、b、c分别通过指针pa、pb和pc定位。这些指针指向各自数组的首元素(第0个元素)。重复练习8.6.1的4和5部分。

练习8.6.3：把在练习8.6.1中得到的三地址代码转换为本节给出的机器模型的机器代码。假设你有任意多个寄存器可用。

练习8.6.4：假设有三个可用的寄存器，使用本节中的简单代码生成算法，把在练习8.6.1中得到的三地址代码转换为机器代码。请给出每一个步骤之后的寄存器和地址描述符。

练习8.6.5：重复练习8.6.4，但是假设只有两个可用的寄存器。

8.7 窥孔优化

虽然大部分编译器产品通过仔细的指令选择和寄存器分配来生成优质代码，但还有一些编译器使用另一种策略：它们先生成原始的代码，然后对目标代码进行“优化”转换，提高目标代码的质量。这里使用术语“优化”具有一定的误导性，因为不能保证得到的代码在任何数学度量之下都是最优的。不管怎么说，很多简单的转换可以有效地改善目标程序的运行时间和空间需求。

一个简单却有效的、用于局部改进目标代码的技术是窥孔优化(peephole optimization)。它在优化的时候检查目标指令的一个滑动窗口(即窥孔)，并且只要有可能就在窥孔内用更快或更短的指令来替换窗口中的指令序列。也可以在中间代码生成之后直接应用窥孔优化来提高中间表示形式的质量。

窥孔是程序上的一个小的滑动窗口。窥孔优化技术并不要求在窥孔中的代码一定是连续的，尽管有些实现要求代码连续。窥孔优化的特点是每一次改进又可能产生出新的优化机会。一般来说，为了获得最大的好处就需要多次扫描目标代码。在本节中，我们将给出下列具有窥孔优化特点的程序变换的例子。

- 冗余指令消除
- 控制流优化
- 代数化简
- 机器特有指令的使用

8.7.1 消除冗余的加载和保存指令

如果我们在目标程序中看到指令序列

```
LD R0, a
ST  a, R0
```

我们就可以删除其中的保存指令，因为不管这个保存指令何时执行，第一个指令将保证a的值已经被加载到寄存器R0中。请注意，假如保存指令有一个标号，我们就不能保证第一个指令总是在第二个指令之前执行，因此不能删除这个保存指令。换句话说，为了保证这样的转换是安全的，这两个指令必须在同一个基本块内。

这种类型的冗余加载/保存指令不会由前一节中的简单代码生成算法生成。但是，一个类似

于 8.1.3 节中的原始的代码生成器可能生成类似的冗余代码序列。

8.7.2 消除不可达代码

另一个窥孔优化的机会是消除不可达的指令。一个紧跟在无条件跳转之后的不带标号的指令可以被删除。通过重复这个运算，就可以删除一个指令序列。比如，为了调试的目的，一个大型程序中可能含有一些只有当变量 debug 等于 1 时才运行的代码片断。在中间表示形式中，这个代码看起来可能就像

```
        if debug == 1 goto L1
        goto L2
L1:     print debugging information
L2:
```

一个显而易见的窥孔优化方法是消除级联跳转指令。因此，不管 debug 的值是什么，上面的代码序列可以被替换为：

```
        if debug != 1 goto L2
        print debugging information
L2:
```

如果 debug 在程序开始的时候被设置为 0，常量传播优化将把这个序列转换为

```
        if 0 != 1 goto L2
        print debugging information
L2:
```

现在，第一个语句的条件值总是 true，因此这个语句可以被替换为 goto L2。替换之后，打印调试信息的所有语句都变成了不可达语句，因此可以被逐一消除。

8.7.3 控制流优化

简单的中间代码生成算法经常生成目标为无条件跳转指令的无条件跳转指令，到达条件跳转指令的无条件跳转指令，或者到达无条件跳转指令的条件跳转指令。这些不必要的跳转指令可以通过下面几种窥孔优化技术从中间代码或者目标代码中消除。我们可以把序列

```
        goto L1
        ...
L1:     goto L2
```

替换为

```
        goto L2
        ...
L1:     goto L2
```

如果没有跳转到 L1 的指令，并且语句 L1：goto L2 之前是一个无条件跳转指令，所以可以消除这个语句。

类似地，序列

```
        if a < b goto L1
        ...
L1:     goto L2
```

可以被替换为序列

```
        if a < b goto L2
        ...
L1:     goto L2
```

最后，假设只有一个到达 L1 的跳转指令，且 L1 之前是一个无条件跳转指令，那么序列

```
        goto L1
        . . .
L1:     if a < b goto L2
L3:
```

可以被替换为序列

```
        if a < b goto L2
        goto L3
        . . .
L3:
```

虽然两个序列中的指令个数相同，但是在第二个序列中我们有时可以跳过无条件跳转指令，而在第一个序列中却不可能。因此，第二个序列的运行时间要优于第一个序列的运行时间。

8.7.4 代数化简和强度消减

在8.5节，我们讨论了可以用于简化DAG的代数恒等式。这些代数恒等式也可以被窥孔优化器用于消除窥孔中类似于

```
x = x + 0
```

或者

```
x = x * 1
```

的三地址语句。

类似地，强度消减转换也可以应用到窥孔中，把代价比较高的运算替换为目标机器上代价较低的等价运算。有些机器指令和另一些指令相比其代价要低很多，它们经常被当作相应的高代价运算的特殊情况来使用。比如，用 $x*x$ 实现 x^2 的代价总是比通过调用求幂函数实现 x^2 的代价要低。对于乘数(除数)为2的幂的定点数乘法(除法)，用移位运算实现的代价要低一些。除数为常数的浮点除法可以通过乘数为该常量倒数的乘法来求近似值。后一种做法的代价要小一点。

8.7.5 使用机器特有的指令

目标机可能会有一些能够高效实现某些特定运算的硬件指令。检测允许使用这些指令的情况可以显著地降低运行时间。比如，有些机器具有自动增量和自动减量的寻址模式。这些指令在使用一个运算分量的值之前或之后，将运算分量的值自动加一或减一。在参数传递时的压栈或出栈运算中使用这个模式可以大大提高代码的质量。这个模式也可以在类似于 `x = x + 1` 的语句的代码中使用。

8.7.6 8.7节的练习

练习8.7.1：构造一个算法，它可以在目标机器代码上的滑动窥孔中进行冗余指令消除。

练习8.7.2：构造一个算法，它可以在目标机器代码上的滑动窥孔中进行控制流优化。

练习8.7.3：构造一个算法，它可以在目标机器代码上的滑动窥孔中进行简单的代数简化和强度消减。

8.8 寄存器分配和指派

只涉及寄存器运算分量的指令要比那些涉及内存运算分量的指令运行得快。在现代的机器上，处理器速度要比内存速度快一个数量级以上。因此，寄存器的有效利用对生成优质代码是非常重要的。本节将给出不同的策略，用于确定在程序的每个点上，哪个值应该存放在寄存器中(寄存器分配)以及各个值应该存放在哪个寄存器中(寄存器指派)。

寄存器分配和指派的方法之一是把目标程序中的特定值分配给特定的寄存器。比如，我们可以确定把基地址指派给一组寄存器，算术计算则使用另一组寄存器，栈顶指针指派给一个固定的寄存器，等等。

这个方法的优点是使代码生成器的设计变得简单。但因为它的应用有太多限制，所以寄存器的使用效率较低：有些被占用的寄存器在相当数量的代码运行中没有被使用到，同时却不得不生成很多不必要的其他寄存器的加载和保存运算指令。虽然如此，在大多数计算环境中还是要

保留一些寄存器。这些被保留的寄存器可以被用作基址寄存器、栈顶指针寄存器或其他类似的用途。其他寄存器则由代码生成器在它认为适当的时候使用。

8.8.1 全局寄存器分配

8.6 节中的代码生成算法在单个基本块的运行期间使用寄存器来存放值。但是，在每个基本块的结尾处，所有活跃变量的值都被保存到内存中。为了省略一部分这样的保存及相应的加载指令，我们可以把一些寄存器指派给频繁使用的变量，并且使得这些寄存器在不同基本块中的(即全局的)指派保持一致。因为程序的大部分时间花在它的内部循环上，所以一个自然的全局寄存器指派方法是试图在整个循环中把频繁使用的值存放在固定的寄存器中。从现在开始，假设我们知道一个流图的循环结构，并且我们知道在一个基本块中计算的哪些值会在该基本块外使用。下一个章将介绍用于计算这些信息的技术。

全局寄存器分配的策略之一是分配固定多个寄存器来存放每个内部循环中最活跃的值。在不同的循环中所选择的值也有所不同。没有被分配的寄存器可以如 8.6 节中说的那样用于存放一个基本块的局部值。这个方法的缺点是固定的寄存器个数并不总是恰好等于用于全局寄存器分配的最佳数量。但是这个方法实现起来很简单，它曾经被用在 Fortran H 中。这是 IBM 在 20 世纪 60 年代后期为 360 系列计算机开发的 Fortran 优化编译器。

在早期的 C 编译器中，程序员可以明确地参与某些寄存器分配过程。他们使用寄存器声明来使得某些值在一个过程运行期间都保存在寄存器中。明智地使用寄存器声明确实可以提高很多程序的运行速度，但是应该鼓励程序员在分配寄存器之前先获取程序的运行时刻特征并确定程序运行的热点代码。

8.8.2 使用计数

通过在循环 L 运行时把一个变量 x 保存在寄存器里面，我们可以节省从内存中加载 x 的开销。在本节我们假设，如果把 x 分配在寄存器中，对 x 的每一次引用可以节省一个单位的(用于加载的)成本。然而，如果 x 在一个基本块中被计算之后又在同一个基本块中被使用，那么当使用 8.6 节中的算法来生成基本块代码时，x 有很大的机会被仍然保存在寄存器中。(因此对 x 的使用很可能本来就不需要从内存中加载。——译者注)因此，只有当 x 在循环 L 的某个基本块内被使用，且在同一基本块中 x 没有被先行赋值时，我们才认为这次使用节约了一个单位的开销。如果我们能够避免在某个基本块的结尾把 x 保存回内存，我们也可以省略 2 个单位的开销：保存指令和之后的加载指令。因此，如果 x 被分配在某个寄存器中，对于每个向 x 赋值且 x 在其出口处活跃的基本块，我们节省了两个单位的开销。

在支出方面，如果 x 在循环头部的入口处活跃，我们必须在进入循环 L 之前把 x 加载到它的寄存器中。这个加载的成本是两个成本单元。类似地，对于循环 L 的每个出口基本块 B，如果 x 在 B 的某个 L 之外的后继的入口处活跃，我们必须以 2 个单位的代价把 x 保存起来。然而，假设循环将迭代多次，我们可以忽略这些支出。因为每次进入循环时，这些指令只会运行一次。因此，在循环 L 中把一个寄存器分配给 x 所得到的好处的一个估算公式是

$$\sum_{L\text{中的全部基本块}B} use(x,B) + 2 * live(x,B) \tag{8.1}$$

其中，$use(x, B)$ 是 x 在 B 中被定值之前被使用的次数。如果 x 在 B 的出口处活跃并在 B 中被赋予一个值，则 $live(x, B)$ 的取值为 1，否则 $live(x, B)$ 为 0。请注意，式 8.1 只是一个估算公式。这是因为一个循环中的各基本块的运行频率实际是不同的，也因为式(8.1)是基于循环被多次迭代的假设之上的。因此在特定的机器上，有可能需要设计一个与式(8.1)类似，但具有一定差异的公式。

例 8.17 考虑图 8-17 中所示的内部循环中的基本块。图中的跳转指令和条件跳转指令都被省略了。假设寄存器 R0、R1 和 R2 用于存放整个循环范围内的值。为方便起见，在图 8-17 中，各个基本块的入口处/出口处的活跃变量分别显示在基本块的上方和下方。我们将在下一章中讨论关于活跃变量的复杂问题。比如，请注意 e 和 f 都在 B_1 的结尾处活跃，但是只有 e 在 B_2 的入口处活跃，只有 f 在 B_3 的入口处活跃。一般来说，在一个基本块的结尾处活跃的变量集合是那些在该基本块的后继基本块的入口处活跃的变量的并集。

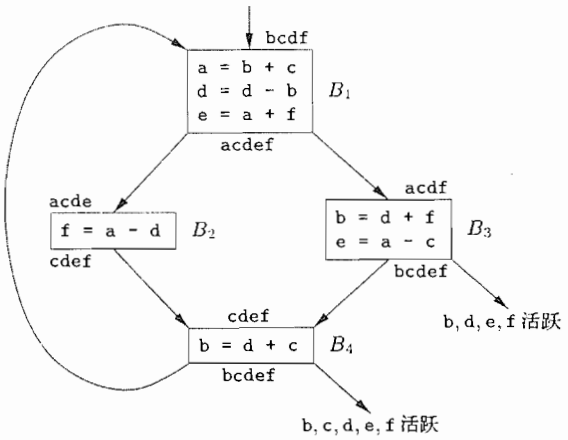


图 8-17 一个内层循环的流图

为了计算当 $x = a$ 时式(8.1)的值，我们观察到 a 在 B_1 的出口处活跃且在 B_1 中被赋值，但是它不在 B_2、B_3、B_4 的出口处活跃。因此，$\sum_{B\text{在循环}L\text{中}} use(a,B) = 2$。当 $x = a$ 时，式(8.1)的值是 4。也就是说，如果选择某个全局寄存器来存放 a 的值，可以节约的 4 个成本单位。对 b、c、d、e 和 f，式(8.1)的值分别是 5、3、6、4 和 4。因此，我们可以为 R0、R1、R2 分别选择 a、b、d。把 R0 用于存放 e 或 f 是另一种选择，显然这样做具有同样的收益。假设 8.6 节中介绍的策略用于生成各个基本块的代码，图 8-18 显示了根据图 8-17 生成的汇编代码。在图 8-17 中，我们没有为略去的各个基本块结尾处的条件或无条件跳转指令生成代码，因此我们没有像通常那样把代码显示成为一个序列。 □

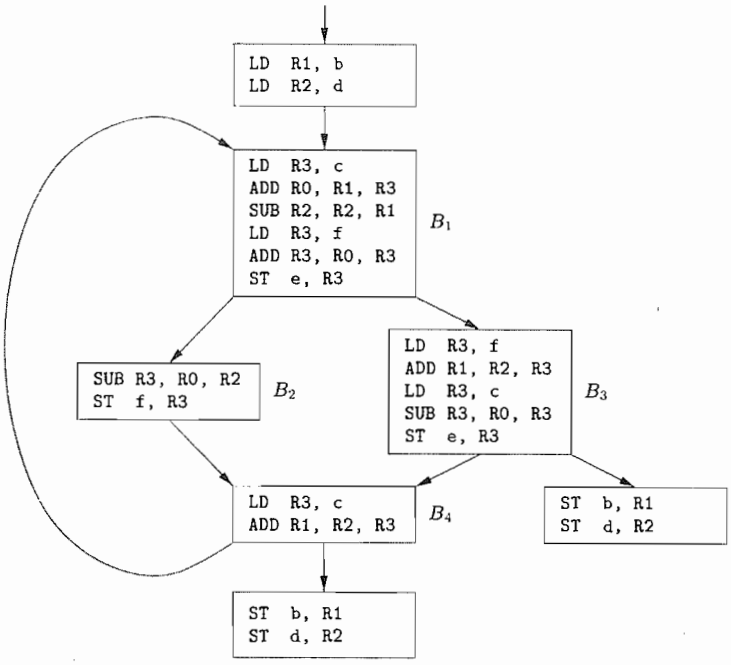


图 8-18 使用全局寄存器指派的代码序列

8.8.3 外层循环的寄存器指派

在为内层循环指派寄存器并生成代码之后，我们可以把同样的想法应用到更大的外围循环上去。如果一个外层循环 L_1 包含一个内层循环 L_2，在 L_2 中分配的寄存器的名字不一定要在 $L_1 - L_2$ 部分也分配到一个寄存器。然而，如果我们决定在 L_2 中（而不是在 L_1 中）为 x 分配一个寄存器，我们必须在 L_2 的入口处加载 x，而在 L_2 的出口处保存 x。我们把在外层循环 L 中选择为哪些名字分配寄存器的标准留作练习，在选择时假设已经为所有嵌套在 L 内部的循环完成了名字选择。

8.8.4 通过图着色方法进行寄存器分配

当计算中需要一个寄存器，但所有可用寄存器都在使用时，某个正被使用的寄存器的内容必须被保存（溢出）到一个内存位置上，以便释放出一个寄存器。图着色方法是一个可用于分配寄存器和管理寄存器溢出的简单且系统化的技术。

这个方法需要进行两趟处理。在第一趟处理中选择目标机器指令，处理时假设有无穷多个符号化寄存器。经过这次处理，中间代码中使用的名字变成了寄存器的名字，而三地址指令变成了机器指令。如果对变量的访问要求一些指令使用栈指针、显示表指针、基址寄存器或其他的量来辅助访问，我们就假设这些量存放在那些为相应目的而保留的寄存器中。通常情况下，它们的使用可以直接翻译成为机器指令中的一个地址所使用的某种访问模式。如果访问方式更加复杂，这个访问就必须被分解成为多个机器指令，并且需要创建一个或多个临时的符号化寄存器。

在选择好了指令之后，第二趟处理把物理寄存器指派给符号化寄存器。这一次处理的目标是寻找到一个溢出代价最小的指派方法。

在第二趟处理中，对每个过程都构造了一个*寄存器冲突图*（register-interference graph）。图中的结点是符号化寄存器。对于任意两个结点，如果一个结点在另一个被定值的地方是活跃的，那么这两个结点之间就有一条边。比如，图 8-17 对应的寄存器冲突图中有两个结点 a 和 b。在基本块 B_1 中，a 在对 b 定值的第二个语句上是活跃的，因此在图中结点 a 和 b 之间有一条边。

然后就可以尝试用 k 种颜色对寄存器冲突图进行着色，其中 k 是可指派的寄存器的个数。一个图被称为*已着色*（colored）当且仅当每个结点都被赋予了一个颜色，并且没有两个相邻的结点的颜色相同。一种颜色代表一个寄存器。着色方案保证不会把同一个物理寄存器指派给两个可能相互冲突的符号化寄存器。

一般来说，确定一个图是否 k-可着色是一个 NP 完全问题，但在实践中我们常常可以使用下面的启发式技术进行快速着色。假设图 G 中有一个结点 n，其邻居（即通过一条边连接到 n 的结点）个数少于 k 个。把 n 及和 n 相连的边从 G 中删除后得到一个图 G'。对图 G' 的一个 k-着色方案可以扩展成为一个对 G 的 k-着色方案：只要给 n 指派一个尚未指派给它的邻居的颜色就可以了。

通过不断地从寄存器冲突图中删除边数少于 k 的结点，要么最终我们得到一个空图，要么得到的图中每个结点都至少有 k 个相邻的结点。在第一种情况下，我们可以依照结点被删除的相反顺序对结点进行着色，从而得到一个原图的 k-着色方案。在第二种情况下已经不存在 k-着色方案了[⊖]。此时就需要通过引入保存和重新加载寄存器的代码，将某个结点溢出。Chaitin 设计了多个用来选择溢出结点的启发式规则。总的原则是避免在内部循环中引入溢出代码。

⊖ 实际并非如此，例如由 4 个结点组成的圈中，每个结点都有两条边，但是却存在 2-着色方案：奇数点为白色，而偶数点为黑色。作者的意思可能是指难以在适当的时间内找出 k-着色方案——译者注。

8.8.5 8.8节的练习

练习8.8.1：为图8-17中的程序构造寄存器冲突图。

练习8.8.2：假设我们在每个过程调用前在栈中自动保存所有的寄存器，并在该过程返回后重新从栈中恢复它们，请设计一个寄存器分配策略。

8.9 通过树重写来选择指令

指令选择可能是一个大型的排列组合任务。对于像CISC这样的具有丰富寻址模式的机器，或者具有某些特殊目的指令(比如信号处理指令)的机器尤其如此。即使我们假设求值的顺序已经给定，并且假设寄存器通过另一个独立的机制进行分配，指令选择——为实现中间表示形式中出现的运算符而选择目标语言指令的问题——仍然是一个规模很大的排列组合任务。

在本节中，我们把指令选择当作一个树重写问题来处理。目标指令的树形表示已经在代码生成器的生成器中得到有效使用。这种生成器可以依据目标机器的高层规约自动构造出一个代码生成器的指令选择阶段。对于某些机器，相对于使用树表示方法而言，使用DAG表示方法能够生成更好的代码。但是DAG匹配比树匹配更加复杂。

8.9.1 树翻译方案

在这一节中，代码生成过程的输入是一个由目标机器的语义层次上的树组成的序列。像8.3节讨论的那样在中间代码中插入运行时刻地址之后就可以得到这些树。另外，这些树的叶子包含有关它们的标号的存储类型的信息。

例8.18 图8-19包含了一个对应于赋值语句`a[i] = b + 1`的树，其中数组`a`存放在运行时刻栈中，而`b`是一个存放在内存位置M_b的全局变量。局部变量`a`和`i`的运行时刻地址是以相对于`SP`的常数偏移量C_a和C_i的方式给出的，其中`SP`是存放当前活动记录的起始位置的寄存器。

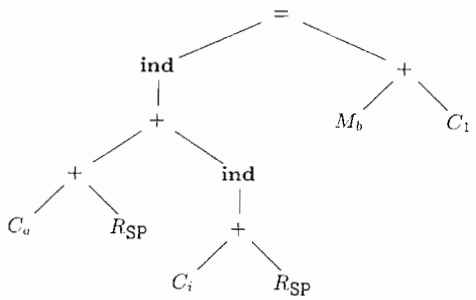


图8-19 `a[i] = b + i`的中间代码树

对`a[i]`的赋值是一个间接赋值，其中`a[i]`的位置上的右值被设置成表达式`b + 1`的右值。数组`a`和变量`i`的地址是通过分别把常量C_a和C_i的值加上寄存器`SP`的内容而得到的。为了简化数组地址的计算，我们假设每个元素值都是一个字节的字符(某些指令集中提供了特殊指令用于在地址计算中进行乘数为某些常数(比如2、4、8等)的乘法运算)。

在这棵树中，运算符**ind**把它的参数作为内存地址处理。作为一个赋值运算符的左子结点，**ind**结点指出了一个内存位置，该位置用来存放赋值运算符右部的右值。如果一个+或者**ind**运算符的某个参数是内存位置或寄存器，那么该内存位置或寄存器中的内容就是参数的值。这棵树的叶子结点的标号为属性，而下标表示属性的值。 □

目标代码是通过应用一个树重写规则序列来生成的，这些规则最终会把输入的树归约为单个结点。各个树重写规则形如

$$replacement \leftarrow template \ \{action\}$$

其中，*replacement*(被替换结点)是一个结点，*template*(模板)是一棵树，*action*(动作)是一个像语法制导翻译方案中那样的代码片断。

一组树重写规则被称为一个*树翻译方案*(tree-translation scheme)。

每个树重写规则表示了如何翻译由模板给出的输入树的一个片段。翻译中包含了一组可能为空的机器指令序列，该序列由与模板关联的动作发出。和输入树一样，模板的叶子是带有下标的属性。有时，会存在一些对于模板中的下标值的约束，这些约束通过语义断言来表示。只有满足这些约束才可以匹配模板。比如，一个断言可能规定某个常数的值必须位于某个区间内。

树翻译方案可以很方便地表示代码生成器的指令选择阶段。作为树重写规则的例子，考虑关于寄存器到寄存器加法指令的规则：

这个规则按照如下方法使用。如果输入树包含一个和上面的模板匹配的子树，也就是说，有一个子树的根结点的标号是运算符 +，且其左右子结点是寄存器 i 和 j 中的量，那么我们可以把这个子树替换为标号为 R_i 的单一结点，同时输出指令 ADD R_i, R_i, R_j。我们把这次替换称为对该子树的一次覆盖(tiling)。在一个给定时刻可能有多个模板与某个子树匹配，我们将简要描述在冲突情况下决定应用哪个规则的一些机制。

例 8.19 图 8-20 包含了我们的目标机上的一部分指令的树重写规则。这些规则将被用于一个贯穿本节的例子中。前面的两个规则对应于加载指令；接下来的两个规则对应于保存指令，其余的规则对应于带有下标的加载与加法运算。请注意，规则(8)要求常量的值必须是 1。这个条件将用一个语义断言来描述。 □

1)	R_i ← C_a	{ LD Ri, #a }
2)	R_i ← M_x	{ LD Ri, x }
3)	M ← = (M_x, R_i)	{ ST x, Ri }
4)	M ← = (ind(R_i), R_j)	{ ST *Ri, Rj }
5)	R_i ← ind(+ (C_a, R_j))	{ LD Ri, a(Rj) }
6)	R_i ← + (R_i, ind(+ (C_a, R_j)))	{ ADD Ri, Ri, a(Rj) }
7)	R_i ← + (R_i, R_j)	{ ADD Ri, Ri, Rj }
8)	R_i ← + (R_i, C_1)	{ INC Ri }

图 8-20 一些目标机指令的树重写规则

8.9.2 通过覆盖一个输入树来生成代码

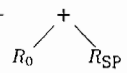
一个树翻译方案按照下面的方式工作。给定一个输入树，在这些树重写规则中的模板被用来覆盖输入树的子树。如果找到一个匹配的模板，那么输入树中匹配的子树将被替换为相应规则中的替换结点，并且执行规则的相关动作。如果这个动作包含了一个机器指令序列，那么就会生成这些指令。这个过程将一直重复，直到这个树被归约成单个结点，或找不到匹配的模板为止。在将一个输入树归约成单个结点的过程中生成的机器指令代码序列就是树翻译方案作用于给定输入树而得到的输出。

这样，描述一个代码生成器的过程就变得和使用语法制导翻译方案来描述翻译器的过程类似。我们写出一个树翻译方案来描述目标机的指令集合。在实践中，我们将试图找到一个能够对每个输入树生成代价最小的指令序列的树翻译方案。现在有很多工具可以帮助我们根据一个树翻译方案自动生成代码生成器。

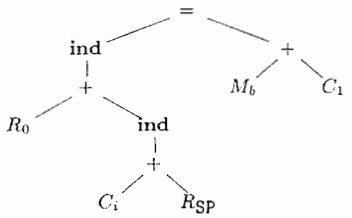
例 8.20 让我们用图 8-20 的树翻译方案来为图 8-19 中的输入树生成代码。假设第一个规则用于把常量 C_a 加载到寄存器 R_0 中：

1) $R_0 \leftarrow C_a$ { LD R0, #a }

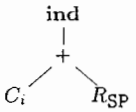
最左边叶子结点的标号就由 C_a 变成 R_0，同时生成了指令 LD R0, #a。现在，第七个规则和最左边的根标号为 + 的子树匹配：

7) $R_0 \leftarrow$  { ADD R0, R0, SP }

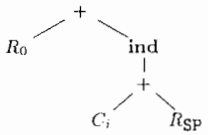
使用这个规则，我们把这棵子树重写为一个标号为 R_0 的单一结点，同时生成指令 ADD R0, R0, SP。现在这棵树如下所示：



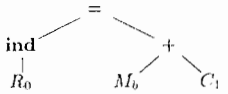
此时，我们可以应用规则(5)来把子树



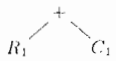
归约为单个结点，设其标号为 R_1。我们也可以使用规则(6)把较大的子树



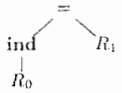
归约为单个结点 R_0，并生成指令 ADD R0, R0, i(SP)。假设用一个指令来计算较大的子树要比计算较小的子树更加高效，我们选择规则(6)得到下面的树：



在右边的子树中，可将规则(2)可应用于叶子结点 M_b，并产生一个把 b 加载到某个寄存器(比方说 R1)的指令。现在，使用规则(8)我们可以匹配子树



并生成增量指令 INC R1。至此，输入树已经被归约成为：



剩下的这棵树和规则(4)匹配，从而把这棵树归约为单个结点，并生成指令 ST *R0, R1。在把树归约成为单一结点的过程中，我们生成了下列代码序列：

```
LD  R0, #a
ADD R0, R0, SP
ADD R0, R0, i(SP)
LD  R1, b
INC R1
ST  *R0, R1
```

□

为了实现对例 8.18 中的树的归约过程，我们必须解决一些和树模式匹配相关的问题：

- 如何完成树模式匹配？代码生成过程(在编译时刻)的效率依赖于树匹配算法的效率。
- 如果在某个给定时刻有多个模板可以匹配，我们该做什么？生成的代码(在运行时刻)的效率依赖于模板被匹配的顺序，因为不同的匹配序列通常将产生不同的目标机代码，这些代码之间的效率是不同的。

如果没有匹配的模板，那么代码生成过程就无法继续了。在另一种极端情况下，我们要防止出现某个单个结点被重写无穷多次的可能性。这种情况会产生无穷多个寄存器之间的移动指令，或者无穷多个加载、保存指令。

为了避免阻塞，我们假设中间代码中的每个运算符都能够使用一个或多个目标机器的指令来实现。我们进一步假设存在足够多的寄存器用于计算树的每个结点。那么，不管树匹配过程如何进行，剩下的树总能够被翻译成为目标机器指令序列。

8.9.3 通过扫描进行模式匹配

在考虑通用的树匹配方法之前，我们先考虑一个特殊的匹配方法。这个方法使用 LR 语法分析器来完成模式匹配。输入树可以用前缀方式表示为一个串。比如，图 8-19 中的树的前缀表示为：

$$= \mathbf{ind} + + C_a\ R_{\mathrm{SP}}\ \mathbf{ind} + C_i\ R_{\mathrm{SP}} + M_b\ C_1$$

一个树翻译方案可以转换为一个语法制导的翻译方案，方法是把每个树重写规则替换为相应的上下文无关文法的产生式。对于一个树重写规则，相应的产生式的右部就是其指令模板的前缀表示方式。

例 8.21 图 8-21 中的语法制导翻译方案是基于图 8-20 中的树翻译方案构造的。

相应文法的非终结符号是 R 和 M。终结符号 **m** 表示特定的内存位置，比如例 8.18 中全局变量 b 的位置。可以这么理解规则(10)中的产生式 $M \rightarrow \mathbf{m}$：在使用涉及 M 的某个模板之前首先要把 **M** 和 **m** 匹配。类似地，我们为寄存器 SP 引入终结符 **sp**，并增加产生式 $R \rightarrow \mathbf{sp}$。最后，终结符 **c** 表示常量。

1)	$R_i \rightarrow \mathbf{c}_a$	{ LD Ri, #a }
2)	$R_i \rightarrow M_x$	{ LD Ri, x }
3)	$M \rightarrow = M_x\ R_i$	{ ST x, Ri }
4)	$M \rightarrow = \mathbf{ind}\ R_i\ R_j$	{ ST *Ri, Rj }
5)	$R_i \rightarrow \mathbf{ind} + \mathbf{c}_a\ R_j$	{ LD Ri, a(Rj) }
6)	$R_i \rightarrow + R_i\ \mathbf{ind} + \mathbf{c}_a\ R_j$	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \rightarrow + R_i\ R_j$	{ ADD Ri, Ri, Rj }
8)	$R_i \rightarrow + R_i\ \mathbf{c}_1$	{ INC Ri }
9)	$R \rightarrow \mathbf{sp}$	
10)	$M \rightarrow \mathbf{m}$	

图 8-21 由图 8-20 构造得到的语法制导翻译方案

使用这些终结符，图 8-19 中的输入树对应的串是：

$$= \mathbf{ind} + + \mathbf{c}_a \ \mathbf{sp} \ \mathbf{ind} + \mathbf{c}_i \ \mathbf{sp} + \mathbf{m}_b \ \mathbf{c}_1 \qquad \square$$

根据这个翻译方案的产生式，我们可以使用第 4 章中的某个 LR 语法分析器构造技术来构建一个 LR 语法分析器。目标代码通过每一步归约中发出的机器指令来生成。

一个用于代码生成的语法具有很大的二义性。在构造语法分析器的时候，对于如何处理语法分析动作冲突的问题要多加小心。在没有指令代价信息的时候，总体处理规则是偏向于执行较大的归约，而不是较小的规约。这意味着在一个归约 - 归约冲突中，优先选择较长的归约；在一个移入 - 归约冲突中，优先选择移入动作。这种“贪吃”的做法使得多个运算由一条机器指令完成。

在代码生成中使用 LR 语法分析方法有多个好处。第一，语法分析方法是高效的，并且容易被人们理解。因此，使用第 4 章中描述的算法可以构造出可靠和高效的代码生成器。第二，比较容易为所得代码生成器重新确定目标。只要写出描述新机器的指令集合的语法，就可以构造得到一个针对新机器的代码选择器。第三，可以通过增加特殊产生式来利用机器特有的指令，从而生成高效的代码。

但使用这个方法也存在着一些挑战。语法分析方法确定了求值过程必须是从左到右的。另外，对于某些具有很多种寻址模式的机器来说，描述机器的文法和由此得到的语法分析器可能变得异常庞大。其结果是人们不得不使用特殊技术对描述机器的文法进行编码和处理。我们还必须注意不要让得到的语法分析器在对表达式树进行语法分析的时候被阻塞（即无法进行下一步动作）。造成阻塞的原因可能是该文法不能处理某些运算符的模式，也可能是语法分析器在解决某些语法分析动作冲突的时候做出了错误的选择。我们必须保证语法分析器不会进入无限循环，不停地使用右部只有单个符号的产生式进行归约。无限循环问题可以在生成语法分析器表的时候通过状态分裂技术来解决。

8.9.4 用于语义检查的例程

在一个代码生成翻译方案中出现的属性和输入树中的属性是一样的。但是翻译方案中的属性常常带有关于该属性下标的取值的限制。比如，一个机器指令可能要求某个属性的值位于特定范围之内，或者两个属性的取值之间有一定关系。

这些关于属性值的限制可以用断言来描述。在进行归约之前需要判断相应的断言是否被满足。实际上，相对于纯文法描述的方式而言，语义动作和断言的普遍使用能够更加灵活、更加容易地对代码生成器加以描述。可以使用通用模板来描述各类指令，然后使用语义动作来为特定情况选择指令。比如，两种不同的加法指令可以用同一个模板来表示：

可以通过特定的断言来消除二义性，解决语法分析 - 动作的冲突问题。这些断言允许在不同的上下文中使用不同的选择策略。因为目标机体系结构的某些方面（比如寻址模式）可以用属性值来描述，所以对目标机器的描述可以变得更小。这种方法的复杂之处在于人们难以验证该翻译方案是否可靠地描述了目标机器。当然，所有的代码生成器都会或多或少地碰到这个问题。

8.9.5 通用的树匹配方法

基于前缀表示的用于模式匹配的 LR 语法分析方法优先处理双目运算符的左运算分量。在一个前缀表示 **op** E_1 E_2 中，有限向前看的 LR 语法分析方法中有关扫描动作的决定必须依据 E_1 的某个前缀做出。这是因为 E_1 可能具有任意长度。右运算分量可能会带来一些能够在目标指令集

中选择较好指令的机会。但是模式匹配方法可能会错失这些机会。

我们也可以弃用前缀表示方式而使用后缀表示。但是，一个用于模式匹配的 LR 语法分析方法会优先处理右运算分量。

对于一个手写的代码生成器，我们可以使用图 8-20 中所示的树模板作为指南，编写一个专门的匹配程序。比如，如果输入树的根的标号是 **ind**，那么唯一能够匹配的是规则 5 的模式；否则如果根的标号是 +，那么可能匹配的是规则 6 ~ 8 的模式。

对于一个可以生成代码生成器的生成器，我们需要一个通用的树匹配算法。通过扩展第 3 章中介绍的串模式匹配技术，我们可以开发出一个高效的自顶向下算法。其基本思想是把每个模板表示成一个串的集合，其中每个串对应于模板中的一条从根到某个叶结点的路径。通过在串中(从左到右地)为每个子结点加入位置编号，我们平等地处理每个运算分量。

例 8.22 在为一个指令集构建串集合的时候，我们将去掉下标。因为进行模式匹配时只考虑属性，而不考虑它们的值。

图 8-22 中的模板有如下的从根到叶子结点的串集合：

C
+ 1 *R*
+ 2 **ind** 1 + 1 *C*
+ 2 **ind** 1 + 2 *R*
+ 2 *R*

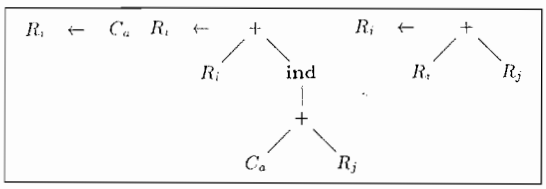


图 8-22 一个用于树匹配的指令集

串 *C* 表示以 *C* 为根的模板。串 + 1 *R* 表示以 + 为根的两个模板中的 + 号和它的左运算分量 *R*。 □

使用例 8.22 中的串集合可以构造出一个树模式匹配程序。该程序使用了可以高效地并行匹配多个串的技术。

在实践中，树重写过程可以按照如下方法实现：对输入树进行深度优先遍历的同时运行树模式匹配程序，并且在最后一次访问这个结点的时候进行归约。

如果要考虑指令代价的问题，可以给每个树重写规则关联一个代价值。这个值等于应用这个规则时所产生的代码序列的总代价。在 8.11 节中，我们将讨论一个可以和树模式匹配算法联合使用的动态规划算法。

通过并发地运行该动态规划算法，我们可以使用各个规则相关的代价信息来选择一个最优的匹配序列。我们要在各个候选序列的代价值都确定之后再决定使用哪个匹配序列。使用这个方法，可以根据一个树重写方案快速地构造出一个小而高效的代码生成器。不仅如此，动态规划算法使得代码生成器的设计者不需要再去解决匹配冲突的问题，或者决定求值的顺序。

8.9.6 8.9 节的练习

练习 8.9.1：为下面的语句构造抽象语法树。假设所有不是常量的运算分量都存放在内存中。

1) `x = a * b + c * d;`

2) `x[i] = y[j] * z[k];`

3) `x = x + 1;`

使用图 8-20 中的树重写方案来为每个语句生成代码。

练习 8.9.2：使用图 8-21 中的语法制导翻译方案来替代树翻译方案，重复练习 8.9.1。

! 练习 8.9.3：扩展图 8-20 中的树重写方案，使之可应用于 while 语句。

! 练习 8.9.4：扩展树重写技术使之应用于 DAG。

8.10 表达式的优化代码的生成

当一个基本块仅包含单一的表达式求值时，或者我们认为以逐次处理各个表达式的方式为基本块生成代码就已经足够了，那么我们就可以最佳地选择寄存器。在下面的算法中，我们引入对一个表达式树(即一个表达式的语法树)的结点添加数字标号的方案。在使用固定个数的寄存器来对一个表达式求值的情况下，该方案允许我们为表达式生成最优的代码。

8.10.1 Ershov 数

一开始，我们给一个表达式树的每个结点各赋予一个数值。该数表示如果我们不把任何临时值存放回内存的话，计算该表达式需要多少个寄存器。这些数有时被称为 *Ershov 数*(Ershov number)。这是根据 A. Ershov 命名的，他为只有一个算术寄存器的机器使用了类似的方案。对我们的机器模型而言，计算 Ershov 数的规则如下：

1) 所有叶子结点的标号为 1。

2) 只有一个子结点的内部结点的标号和其子结点的标号相同。

3) 具有两个子结点的内部结点的标号按照如下方式确定：

① 如果两个子结点的标号不同，那么选择较大的标号。

② 如果两个子结点的标号相同，那么它的标号就是子结点的标号值加一。

例 8.23 在图 8-23 中，我们可以看到一个表达式树(其中的运算符已经被省略)。这个树可能是表达式 $(a-b)+e\times(c+d)$ 的树，或者说是下面的三地址代码的树：

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

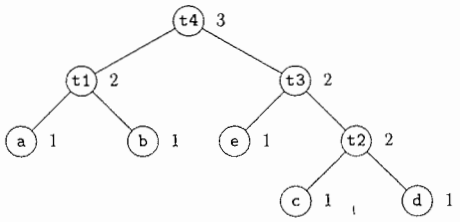


图 8-23 一个用 Ershov 数标号的树

根据规则(1)，该树的五个叶子结点的标号都是 1。然后，我们可以给对应于 `t1 = a - b` 的内部结点加上标号，因为它的两个子结点都已经被加上了标号。应用规则 3，该结点的标号是它的子结点的标号加上 1，也就是 2。对应于 `t2 = c + d` 的结点的标号的计算方式与此类似。

现在我们可以计算对应于 `t3 = e * t2` 的结点的标号。它的子结点的标号是 1 和 2，因此根据规则 3，`t3` 对应结点的标号是其中的较大值，即 2。最后计算根结点，即对应于 `t4 = t1 + t3` 的结点。它的两个子结点的标号都是 2，因此它的标号是 3。 □

8.10.2 从带标号的表达式树生成代码

假设在我们的机器模型中，所有的运算分量都必须在寄存器中，且寄存器可以同时用于存放某个运算的运算分量和结果。可以证明，如果在计算表达式的过程中不允许把中间结果保存回内存，那么一个结点的标号就等于计算该结点对应的表达式时需要的最少的寄存器个数。因为在这个机器模型中，我们必须把每个运算分量加载到寄存器中，且必须计算每个内部结点所对应的中间结果，所以，造成生成代码不是最优代码的唯一可能是我们使用了不必要的将临时结果存回内存的指令。对这个断言的证明包含在下面的算法中。这个算法生成的代码不包含将临时结果存回内存的指令，而这个代码所使用的寄存器数目就是根结点的标号。

算法 8.24 根据一个带标号的表达式树生成代码。

输入： 一个带有标号的表达式树，其中的每个运算分量只出现一次(即没有公共子表达式)。

输出：计算根结点对应的值并将该值存放在一个寄存器中的最优的机器指令序列。

方法：下面是一个用来生成机器代码的递归算法。从树的根结点开始应用下面的步骤。如果算法被应用于一个标号为 k 的结点，那么得到的代码只使用 k 个寄存器。然而，这些代码从某个基线 $b(b \geqslant 1)$ 开始使用寄存器，实际使用的寄存器是 R_b，R_{b+1}，…，R_{b+k-1}。计算结果总是存放在 R_{b+k-1} 中。

1）为一个标号为 k 且两个子结点的标号相同（它们的标号必然是 $k-1$）的内部结点生成代码时，做如下处理：

① 使用基线 $b+1$ 递归地为它的右子树生成代码。其右子树的结果将存放在寄存器 R_{b+k-1} 中。

② 使用基线 b，递归地为它的左子树生成代码。其左子树的结果将存放在寄存器 R_{b+k-2} 中。

③ 生成指令“OP R_{b+k-1}，R_{b+k-2}，R_{b+k-1}”，其中 OP 是标号为 k 的结点对应的运算。

2）假设我们有一个标号为 k 的内部结点，其子结点的标号不相等。那么，它必然有一个子结点的标号为 k，我们称之为“大子结点”；而另一个子结点的标号为某个 $m<k$，它被称为“小子结点”。使用基线 b，通过下列步骤为这个内部结点生成代码：

① 使用基线 b，递归地为大子结点生成代码，其结果存放在寄存器 R_{b+k-1} 中。

② 使用基线 b，递归地为小子结点生成代码，其结果存放在寄存器 R_{b+m-1} 中。请注意，因为 $m<k$，寄存器 R_{b+k-1} 和编号更高的寄存器都没有被使用。

③ 根据大子结点是该内部结点的右子结点还是左子结点，分别生成指令“OP R_{b+k-1}，R_{b+m-1}，R_{b+k-1}”或者“OP R_{b+k-1}，R_{b+k-1}，R_{b+m-1}”。

3）对于代表运算分量 x 的叶子结点，当基线为 b 时生成指令“LD R_b，x”。 □

例 8.25 让我们把算法 8.24 应用于图 8-23 中的树。因为根结点的标号是 3，其结果将存放在 R_3 中，并且只有寄存器 R_1、R_2、R_3 被使用。根结点的基线是 $b=1$。因为根结点的两个子结点的标号相同，我们首先以 2 为基线生成右子结点的代码。

当我们为根结点的标号为 t3 的右子结点生成代码时，我们发现该子结点的大子结点是其右子结点，而小子结点是其左子结点。这样，我们首先以 2 为基线生成右子结点的代码。应用针对具有相同标号子结点和叶子结点的规则，我们为标号 t2 的结点生成下列代码：

```
LD  R3, d
LD  R2, c
ADD R3, R2, R3
```

接下来，我们为根结点的右子结点的左子结点生成代码。这是一个标号为 e 的叶子结点。因为 $b=2$，正确的指令是

```
LD  R2, e
```

现在我们加上指令

```
MUL R3, R2, R3
```

就完整地生成了根结点的右子结点的代码。算法继续以 1 为基线生成根结点的左子结点的代码，并把结果放在 R_2 中。图 8-24 中显示了生成的全部指令序列。 □

```
LD  R3, d
LD  R2, c
ADD R3, R2, R3
LD  R2, e
MUL R3, R2, R3
LD  R2, b
LD  R1, a
SUB R2, R1, R2
ADD R3, R2, R3
```

图 8-24 图 8-23 中的树的最优的三地址代码

8.10.3 寄存器数量不足时的表达式求值

当可用寄存器的数量少于树的根结点的标号时，我们不能直接应用算法 8.24。此时需要引入一些保存指令，把某些子树的值溢出到内存中，然后在必要的时候生成加载指令把那些值再加载到寄存器中。下面是一个经过修改的代码生成算法，它考虑了寄存器数量的限制。

算法 8.26 根据一个带标号的表达式树生成代码。

输入：一个带有标号的表达式树和寄存器的数量 $r \geq 2$。表达式树的每个运算分量只出现一次(即没有公共子表达式)。

输出：计算根结点对应的值并将其存放到一个寄存器中的最优的机器指令序列。代码使用的寄存器的数量不大于 r。我们假设这些寄存器为 R_1，R_2，…，R_r。

方法：令基线 $b=1$，从根结点开始应用下面的递归算法。对于标号为 r 或者更小的结点 N，本算法和算法 8.24 完全一样，这里不再重复。但是，对于标号 $k>r$ 的内部结点，我们要分别处理该内部节点的各个子结点，并把较大子树的结果保存到内存中。该结果在对结点 N 求值之前才从内存重新加载，而最后的求值步骤将在 R_{r-1} 和 R_r 内进行。对于基本算法的改动如下：

1) 结点 N 至少有一个子结点的标号为 r 或者大于 r。选择较大的子结点(如果子结点标号相同则选择任意一个)作为“大”子结点，并把另外一个子结点作为“小”子结点。

2) 令基线 $b=1$，递归地为大子结点生成代码。这个求值的结果将存放在寄存器 R_r 中。

3) 生成机器指令“ST t_k，R_r”，其中 t_k 是一个用于存放中间结果的临时变量。这个变量用于对标号为 k 的结点求值。

4) 按照如下方式为小子结点生成代码。如果小子结点的标号大于或等于 r，选取基线 $b=1$。如果小子结点的标号为 $j<r$，选取基线 $b=r-j$。然后递归地把本算法应用于小子结点，其结果存放在 R_r 中。

5) 生成指令“LD R_{r-1}，t_k”。

6) 如果大子结点是 N 的右子结点，生成指令“OP R_r，R_r，R_{r-1}”。如果大子结点是 N 的左子结点，生成代码“OP R_r，R_{r-1}，R_r”。 □

例 8.27 现在假设 $r=2$，让我们重新回顾一下图 8-23 所代表的表达式。也就是说，只有寄存器 R1 和 R2 可以用来存放表达式求值过程中产生的临时结果。当我们把算法 8.26 应用到图 8-23 中时，我们看到根结点的标号(3)大于 $r=2$。这样，我们需要选择其中的一个子结点作为大子结点。因为子结点的标号相同，我们可以任选其中的一个。假设我们选择了右子结点作为大子结点。

因为根结点的大子结点的标号为 2，因此寄存器是够用的。我们把算法 8.24 应用到这个子树，其中基线 $b=1$，而寄存器个数为 2。最终的结果和我们在图 8-24 中生成的代码很相似，但原来的寄存器 R2 和 R3 被替换为 R1 和 R2。代码如下：

```
LD  R2, d
LD  R1, c
ADD R2, R1, R2
LD  R1, e
MUL R2, R1, R2
```

现在，因为我们要把这两个寄存器都用于根结点的左子树，我们需要生成指令

```
ST  t3, R2
```

接下来处理根结点的左子结点。同样，寄存器的数量足以处理这个子结点，代码如下：

```
LD  R2, b
LD  R1, a
SUB R2, R1, R2
```

最后，我们用指令

```
LD  R1, t3
```

把存放了根结点的右子结点的值的临时变量重新加载到寄存器中，并使用指令

```
ADD R2, R2, R1
```

执行树的根结点上的运算。完整的指令序列显示在图 8-25 中。 □

```
LD  R2, d
LD  R1, c
ADD R2, R1, R2
LD  R1, e
MUL R2, R1, R2
ST  t3, R2
LD  R2, b
LD  R1, a
SUB R2, R1, R2
LD  R1, t3
ADD R2, R2, R1
```

图 8-25 图 8-23 中的树的最优的三寄存器代码(只使用两个寄存器)

8.10.4 8.10 节的练习

练习 8.10.1：计算下列表达式的 Ershov 数。

1) $a/(b+c)-d*(e+f)$

2) $a+b*(c*(d+e))$

3) $(-a+*p)*((b-*q)/(-c+*r))$

练习 8.10.2：使用两个寄存器为练习 8.10.1 中的各个表达式生成最优的代码。

练习 8.10.3：使用三个寄存器为练习 8.10.1 中的各个表达式生成最优的代码。

! **练习 8.10.4**：将 Ershov 数的计算方法一般化，使之能够处理其中某些内部结点具有三个或更多的子结点的表达式树。

! **练习 8.10.5**：类似于 a[i]=x 的对数组元素的赋值看起来像一个具有三个运算分量（a、i 和 x）的运算符。你将如何修改给表达式树添加标号的方案，以便为这种机器模型生成最优的代码？

! **练习 8.10.6**：最初的 Ershov 数技术所应用的机器模型和书中的模型有所不同。该模型允许一个表达式的右运算分量存放在内存中，而不一定要存放在寄存器中。你将如何修改为表达式树添加标号的方案，使得它可以为这种机器模型生成最优代码？

! **练习 8.10.7**：某些机器要求使用两个寄存器来存放某些单精度值。假设单寄存器值的乘法的结果需要两个连续的寄存器，而当我们计算 a/b 时，a 的值必须存放在两个连续的寄存器中。你将如何修改为表达式树添加标号的方案，使得它可以为这种机器模型生成最优代码？

8.11 使用动态规划的代码生成

8.10 节中的算法 8.26 根据一个表达式树生成最优代码所需的时间是树的大小的线性函数。适合使用这个过程的机器要满足以下假设：所有的计算都在寄存器中完成，而指令中包含的运算符要么作用于两个寄存器，要么作用于一个寄存器和一个内存位置。

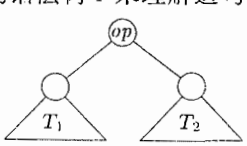
基于动态规划原理的算法可以应用到更多类型的机器上，使得人们可以在线性时间内为一个表达式树生成最优代码。动态规划算法可以被应用到具有复杂指令集的多种计算机上。

只要一个机器具有 r 个可互换的寄存器 R0，R1，…，Rr-1 以及加载、保存和运算指令，就可以应用基于动态规划的算法为这个机器生成代码。为简单起见，我们假设每个指令的代价是一个成本单位。然而，即使每个指令具有不同的代价值，人们也可以很容易地修改这个算法来处理这种情况。

8.11.1 连续求值

动态规划算法把为一个表达式生成最优代码的问题分解成为多个为该表达式的子表达式生成最优代码的子问题。作为一个简单的例子，考虑一个形如 E_1+E_2 的表达式 E。E 的一个最优程序由 E_1 和 E_2 的最优程序以某种顺序组合而成，然后是对 + 求值的代码。为 E_1 和 E_2 生成最优程序的子问题也以类似的方式解决。

由动态规划算法产生的最优程序有一个重要的性质。该代码以“连续”的方式计算表达式 $E=E_1$ **op** E_2。我们可以通过查看 E 的语法树 T 来理解这句话的含义。



这里，T_1 和 T_2 分别是 E_1 和 E_2 的语法树。

我们说一个程序 P 连续计算一棵树 T，如果它首先计算那些需要计算值并将其存放到内存中的 T 的子树。然后，它再计算 T 的其余部分，计算的顺序可以是 T_1，T_2，根结点，或者 T_2，T_1，根结点。无论在哪种情况下，作为非连续计算的一个例子，程序 P 可能先计算 T_1 的一部分并把结果存放在一个寄存器中(而不是内存中)，然后计算 T_2，然后再回过来计算 T_1 的其余部分。

对于本节中的寄存器机器，我们可以证明对于任何一个计算表达式树 T 的机器语言程序 P，我们都可以找到一个等价的程序 P'，使得

1) P'的代价不高于 P 的代价。

2) P'使用的寄存器不多于 P 使用的寄存器，而且

3) P'连续地对该树求值。

这个结果表明，每个表达式树可以用一个连续程序最优地求值。

相对而言，使用偶数 - 奇数寄存器对的计算机不一定总是具有最优的连续求值过程。x86 体系结构在乘法和除法中使用寄存器对。对于这样的机器，我们可以给出一些表达式树的例子。这些树的最优机器语言程序必须首先对根的左子树的一部分进行求值并把结果存放到寄存器中，然后处理右子树的一部分，再处理左子树的另一部分，如此往复。使用本节中的机器对任意一个表达式树进行最优求值时，没有必要进行这种类型的摆动。

上面定义的连续求值的性质保证了对于任何表达式树 T，总是存在一个最优程序。这个程序由根结点的子树的最优程序组成，最后是计算根结点值的指令。这个性质支持我们使用一个动态规划算法为 T 生成一个最优程序。

8.11.2 动态规划的算法

动态规划算法有三个步骤(假设目标机器具有 r 个寄存器)：

1) 对表达式树 T 的每个结点 n 自底向上地计算得到一个代价数组 C，其中 C 的第 i 个元素 $C[i]$是在假设有 $i(1 \leqslant i \leqslant r)$个可用寄存器的情况下对以 n 为根的子树 S 求值并将结果存放在一个寄存器中的最优代价。

2) 遍历 T，使用代价向量(数组)来决定 T 的哪棵子树应该被计算并保存到内存中。

3) 使用每个结点的代价向量和相关指令来遍历各棵子树并生成最终的目标代码。在这个过程中，首先为那些需要把结果值保存到内存的子树生成代码。

上述每一个步骤都可以高效地实现，运行所需时间与表达式树的大小成线性关系。

计算一个结点 n 的代价包括在给定寄存器数量的情况下对 S 求值时所需要的全部加载和保存运算，也包括了计算 S 的根结点处的运算符所需要的代价。代价向量的第 0 个元素存放的是把子树 S 的值计算出来并保存到内存的最优代价。只需要考虑 S 的根结点的各子树的最优程序的不同组合，就可以生成 S 的最优程序。这是由连续求值的性质来确保的。这个限制减少了需要考虑的情况。

为了计算结点 n 的代价 $C[i]$，我们像 8.9 节中那样把指令看作是树重写规则。考虑和结点 n 处的输入树相匹配的各个模板 E。只要检查 n 的相应后代的代价向量，就可以确定对 E 的叶子结点所代表的运算分量进行求值时所需要的代价。对于 E 的寄存器运算分量，考虑对 T 的相应子树求值并放到寄存器中的各种可能的顺序。在每个顺序中，第一个对应于某个寄存器运算分量的子树可以使用 i 个寄存器，而第二个则使用 $i-1$ 个寄存器，以此类推。考虑结点 n 时，需要加上和模板 E 相关的指令的代价。$C[i]$的值就是所有这些可能的顺序所对应的代价值中的最小者。

整棵树 T 的代价向量可以用自底向上的方式计算。计算所需时间和 T 中结点的个数呈线性正比关系。在每个结点上为各个 i 值保存用于获得最优代价 $C[i]$所使用的指令可以带来方便。T 的根结点的代价向量中的最小值给出了对 T 求值所需的最小代价。

例 8.28 考虑有两个寄存器 R0、R1 及下列的指令的机器。每个指令的代价是一个成本单位：

```
LD  Ri, Mj      // Ri = Mj
op  Ri, Ri, Rj  // Ri = Ri op Rj
op  Ri, Ri, Mj  // Ri = Ri op Mj
LD  Ri, Rj      // Ri = Rj
ST  Mi, Rj      // Mi = Rj
```

在这些指令中，Ri 可以是 R0 或者 R1，而 Mj 则是一个内存位置。运算符 op 对应于某个算术运算符。

让我们应用动态规划算法为图 8-26 中的语法树生成最优的代码。在第一步中，我们计算每个结点的代价向量。这些向量在图中各个结点的旁边显示。为了说明代价计算方法，考虑在叶子结点 a 处的代价向量。$C[0]$（即计算 a 并保存到内存的代价）是 0，因为它已经在内存中了。$C[1]$（即计算 a 并保存到一个寄存器的代价）是 1，因为我们可以使用指令 LD R0，a 把它加载到一个寄存器中。$C[2]$（即在有两个可用寄存器的情况下把 a 加载到一个寄存器中的代价）和只有一个可用寄存器的情况下的代价是一样的。因此，在叶子结点 a 上的代价向量是(0，1，1)。

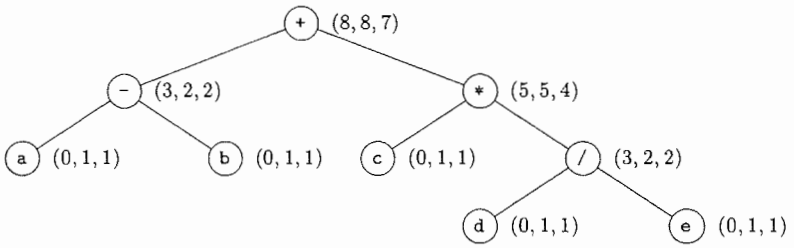


图 8-26 表达式(a - b) + c * (d/e)的语法树，每个结点都标有代价向量

考虑一下根结点处的代价向量。我们首先确定在有一个及两个可用寄存器的情况下计算根结点所需的最小代价。因为根结点的标号是 +，所以机器指令 ADD R0，R0，M 和根结点匹配。使用这个指令，在只有一个可用寄存器的情况下对根结点求值的最小代价的计算方法如下：对其右子树求值并存放到内存的最小代价，加上计算其左子树并保存到寄存器的最小代价，再加上该指令的代价 1。不存在其他的最小代价的计算方式。在根结点的左右子结点上的代价向量说明在只有一个可用寄存器的情况下对根结点求值的最小代价是 5 + 2 + 1 = 8。

现在考虑有两个可用寄存器时对根结点求值的最小代价。根据用于计算根结点的不同指令，以及对根结点的左右子树求值的不同顺序，需要考虑三种情况。

1）使用两个可用寄存器计算左子树的值并放到寄存器 R0 中，使用一个可用寄存器计算右子树的值并放到寄存器 R1 中，并使用指令 ADD R0，R0，R1 来计算根结点。这个指令序列的代价是 5 + 2 + 1 = 8。

2）使用两个可用寄存器计算右子树的值并存放到 R1 中，使用一个可用寄存器计算左子树的值并存放到 R0 中，并使用指令 ADD R0，R0，R1 计算根结点。这个指令序列的代价为 4 + 2 + 1 = 7。

3）计算右子树的值并保存到内存位置 M 中，使用两个可用寄存器计算左子树的值并保存到寄存器 R0 中，并使用指令 ADD R0，R0，M 计算根结点的值。这个指令序列的代价是 5 + 2 + 1 = 8。

可见，第二种选择给出了最小的代价 7。

计算根结点的值并保存到内存中的代价等于使用所有可用寄存器计算根结点的值的最小代价再加上 1。也就是说，我们首先计算根结点并将其存放到一个寄存器中，然后保存结果。因此，在根结点处的代价向量是(8，8，7)。

根据代价向量，我们可以很容易地通过对树的遍历构造出代码序列。假设有两个可用寄存器，图 8-26 的树的最优代码序列是：

```
LD  R0, c          // R0 = c
LD  R1, d          // R1 = d
DIV R1, R1, e      // R1 = R1 / e
MUL R0, R0, R1     // R0 = R0 * R1
LD  R1, a          // R1 = a
SUB R1, R1, b      // R1 = R1 - b
ADD R1, R1, R0     // R1 = R1 + R0
```

□

动态规划技术已经在很多编译器中使用，这些编译器包括可移植 C 编译器版本 2，即 PCC2。因为动态规划技术可以用到很多类型的机器上，这个技术促进了编译器的可重定向特性的发展。

8.11.3 8.11 节的练习

练习 8.11.1：在图 8-20 中的树重写方案中增加代价信息，并用动态规划和树匹配技术来为练习 8.9.1 中的语句生成代码。

!! **练习 8.11.2**：你将如何扩展动态规划技术，以便在 DAG 的基础上生成最优代码？

8.12 第 8 章总结

- *代码生成*是编译器的最后一个步骤。代码生成器把前端生成的中间表示形式映射为目标程序。如果存在一个代码优化阶段，那么代码生成器的输入就是代码优化器生成的中间表示形式。
- *指令选择*是为每个中间表示语句选择目标语言指令的过程。
- *寄存器分配*是决定哪些 IR 值将会保存在寄存器中的过程。图着色算法是一个在编译器中完成寄存器分配的有效技术。
- *寄存器指派*是决定用哪个寄存器来存放一个给定的 IR 值的过程。
- *可重定向编译器*是能够为多个指令集生成代码的编译器。
- *虚拟机*是一些字节代码中间语言的解释程序，这些字节代码是为诸如 Java 和 C#这样的语言生成。
- CISC *机器*通常是一个二地址机器。它的寄存器相对较少，有几种寄存器类型，并具有复杂寻址模式的可变长指令。
- RISC *机器*通常是一个三地址机器。它拥有很多寄存器，且运算都在寄存器中进行。
- *基本块*是一个三地址语句的最大连续序列。控制流只能从它的第一个语句进入，并从最后一个语句离开，中间没有停顿，且除了基本块的最后一个语句之外没有分支语句。
- *流图*是程序的一种图形化表示方式。其中图的结点是基本块，而图的边显示了控制流如何在基本块之间流动。
- *流图中的循环*是一个强连通的区域。这个区域只有一个被称为循环首结点的入口。
- 基本块的 DAG 表示是一个有向无环图。DAG 中的结点表示基本块中的语句，而一个结点的各个子结点所对应的语句是最晚对该结点对应语句的某个运算分量进行定值的语句。
- *窥孔优化*是一种提高代码质量的局部变换。它通常通过一个滑动窗口作用于一个程序。
- *指令选择*可以通过一个树重写过程完成。在这个过程中，对应于机器指令的树模式被用来逐步覆盖一棵语法树。我们可以把树重写规则和相应的指令代价关联起来，并应用动态规划技术来为多种类型的机器和表达式生成最优的覆盖方式。
- Ershov 数指出了如果不把任何临时值保存回内存中，对一个表达式求值需要多少个寄存器。

- 溢出代码是一个把某个寄存器中的值保存到内存中的指令序列。这些指令的目的是在寄存器中腾出空间，以保存另一个值。

8.13 第 8 章参考文献

本章中讨论的很多技术在最早的编译器中就出现了。Ershov 的加标号算法出现在 1958 年[7]。Sethi 和 Ullman[16]在一个算法中使用了这种标号方法。他们还证明了这种算法可以为算术表达式生成最优代码。Aho 和 Johnson[1]使用动态规划技术来为 CISC 机器上的表达式树生成最优代码。Hennessy 和 Patterson[12]对 CISC 和 RISC 机器体系结构的发展，以及在设计一个好的指令集时需要做出的权衡进行了很好的讨论。

虽然 RISC 的历史可以追溯到更早的计算机中，比如最先在 1964 年交付的 CDC6600，但 RISC 体系结构在 1990 年之后才流行起来。在 1990 年之前设计的很多计算机都是 CISC 机器，然而大多数在 1990 年之后安装的通用计算机仍然是 CISC 机器，因为它们都基于 Intel 80x86 或其后代（比如 Pentium 芯片）的体系结构。在 1963 年交付的 Burroughs B5000 是一个早期的栈计算机。

本章中给出的很多关于代码生成的启发式规则已经被用到不同的编译器中。我们描述了在循环执行时用固定数量寄存器存放变量的策略。这个策略被 Lowry 和 Medlock 用在 Fortran H 的实现中[13]。

高效的寄存器分配技术在编译器出现的最早时代就开始研究了。把图着色算法作为一种寄存器分配技术是由 Cocke、Ershov[8]和 Schwartz[15]提出的。针对寄存器分配，人们提出了很多种图着色算法的变体。我们处理图着色的方法来自于 Chaitin[3][4]。Chow 和 Hennessy 在[5]中描述了他们的可用于寄存器分配的基于优先级的着色算法。在[6]中可以见到针对最新的用于寄存器分配的图分划和重写技术的讨论。

词法分析器和语法分析器的自动生成工具刺激了模式制导的指令选择技术的发展。Glanville 和 Graham[11]使用 LR 语法分析器生成技术来处理指令的自动选择。表格驱动的代码生成器发展成为多个基于树模式匹配的代码生成工具[14]。在代码生成工具 twig 中，Aho、Ganapathi 和 Tjiang[2]把高效的树模式匹配技术和动态规划技术结合起来。Fraser、Hanson 和 Proebsting[10]在他们的简单有效的代码生成器的生成器中进一步精化了这些思想。

1. Aho, A. V. and S. C. Johnson, "Optimal code generation for expression trees," *J. ACM* **23**:3, pp. 488–501.

2. Aho, A. V., M. Ganapathi, and S. W. K. Tjiang, "Code generation using tree matching and dynamic programming," *ACM Trans. Programming Languages and Systems* **11**:4 (1989), pp. 491–516.

3. Chaitin, G. J., M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages* **6**:1 (1981), pp. 47–57.

4. Chaitin, G. J., "Register allocation and spilling via graph coloring," *ACM SIGPLAN Notices* **17**:6 (1982), pp. 201–207.

5. Chow, F. and J. L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Trans. Programming Languages and Systems* **12**:4 (1990), pp. 501–536.

6. Cooper, K. D. and L. Torczon, *Engineering a Compiler*, Morgan Kaufmann, San Francisco CA, 2004.

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

8. Ershov, A. P., *The Alpha Automatic Programming System*, Academic Press, New York, 1971.

9. Fischer, C. N. and R. J. LeBlanc, *Crafting a Compiler with C*, Benjamin-Cummings, Redwood City, CA, 1991.

10. Fraser, C. W., D. R. Hanson, and T. A. Proebsting, "Engineering a simple, efficient code generator generator," *ACM Letters on Programming Languages and Systems* **1**:3 (1992), pp. 213–226.

11. Glanville, R. S. and S. L. Graham, "A new method for compiler code generation," *Conf. Rec. Fifth ACM Symposium on Principles of Programming Languages* (1978), pp. 231–240.

12. Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufman, San Francisco, 2003.

13. Lowry, E. S. and C. W. Medlock, "Object code optimization," *Comm. ACM* **12**:1 (1969), pp. 13–22.

14. Pelegri-Llopart, E. and S. L. Graham, "Optimal code generation for expressions trees: an application of BURS theory," *Conf. Rec. Fifteenth Annual ACM Symposium on Principles of Programming Languages* (1988), pp. 294–308.

15. Schwartz, J. T., *On Programming: An Interim Report on the SETL Project*, Technical Report, Courant Institute of Mathematical Sciences, New York, 1973.

16. Sethi, R. and J. D. Ullman, "The generation of optimal code for arithmetic expressions," *J. ACM* **17**:4 (1970), pp. 715–728.