第5章 代码优化

5.1 局部优化

5.1.1 基本块的划分方法

所谓基本块，是指程序中一顺序执行的语句序列，其中只有一个入口和一个出口，入口就是该序列的第一个语句，出口就是该序列的最后一个语句。对一个基本块来说，执行时只能从其入口进入，从其出口退出。对一个给定的程序，我们可以把它划分为一系列基本块，在各个基本块范围内进行的优化称为局部优化。划分基本块的关键问题是准确定义入口和出口语句。下面我们给出划分四元式程序为基本块的算法。

(1) 从四元式序列确定满足以下条件的入口语句：

① 四元式序列的第一个语句。

② 能由条件转移语句或无条件转移语句转移到的语句。

③ 紧跟在条件转移语句后面的语句。

(2) 确定满足以下条件的出口语句：

① 下一个入口语句的前导语句。

② 转移语句(包括转移语句自身)。

③ 停语句(包括停语句自身)。

例如，考察下面求最大公因子的三地址代码程序：

(1) read X
(2) read Y
(3) R=X % Y
(4) if R=0 goto (8)
(5) X=Y
(6) Y=R
(7) goto (3)
(8) write Y
(9) halt

根据上述划分基本块的算法可确定四元式 (1)、(3)、(5)、(8) 是入口语句，而四个基本块分别是：(1) (2)，(3) (4)，(5) (6) (7)，(8) (9)。

5.1.2 基本块的DAG方法

DAG(Directed Acyclic Graph)是一种有向图，常常用来对基本块进行优化。一个基本块的DAG是一种其结点带有下述标记或附加信息的DAG。

(1) 图的叶结点(无后继的结点)以一标识符(变量名)或常数作为标记，表示该结点代表该变量或常数的值。如果叶结点用来表示一变量A的地址，则用addr(A) 作为该结点的标记。通常把叶结点上作为标记的标识符加上下标0，以表示它是该变量的初值。

(2) 图的内部结点(有后继的结点)以一运算符作为标记，表示该结点代表应用该运算符对其直接后继结点所代表的值进行运算的结果。

(3) 图中各个结点上可能附加一个或多个标识符，表示这些变量具有该结点所代表的值。

一个基本块由一个四元式序列组成，且每一个四元式都可以用相应的DAG结点表示。图5-1给出了不同四元式和与其对应的DAG结点形式。图中，各结点圆圈中的n_i是构造DAG过程中各结点的编号，而各结点下面的符号(运算符、标识符或常数)是各结点的标记，各结点右边的标识符是结点上的附加标识符。除了对应转移语句的结点右边可附加一语句位置来指示转移目标外，其余各类结点的右边只允许附加标识符。除对应于数组元素赋值的结点(标记为[ ]=)有三个后继外，其余结点最多只有两个后继。

利用DAG进行基本块优化的基本思想是：首先按基本块内的四元式序列顺序将所有的四元式构造成一个DAG，然后按构造结点的次序将DAG还原成四元式序列。由于在构造DAG的同时已做了局部优化，所以最后所得到的是优化过的四元式序列。

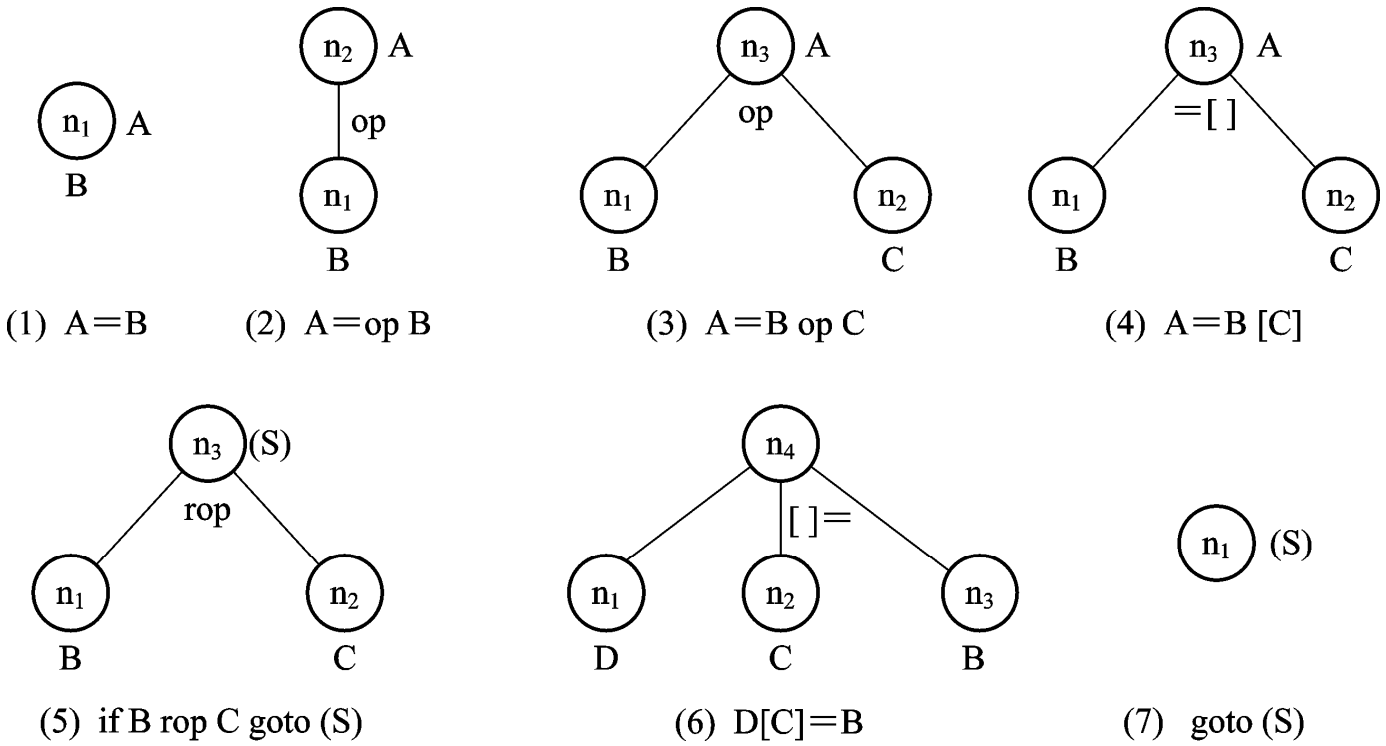


图5-1 四元式与DAG结点

为了DAG构造算法的需要，我们将图5-1中的四元式按照其对应结点的后继结点个数分为四类：

(1) 0型四元式：后继结点个数为0，如图5-1(1) 所示。

(2) 1型四元式：有一个后继结点，如图5-1(2) 所示。

(3) 2型四元式：有两个后继结点，如图5-1(3)、(4)、(5) 所示。

(4) 3型四元式：有三个后继结点，如图5-1(6) 所示。

我们规定：用大写字母(如A、B等)表示四元式中的变量名(或常数)；用函数Node(A) 表示A在DAG中的相应结点，其值可为n或者无定义，并用n表示DAG中的一个结点值。这样，每个基本块仅含0、1、2型四元式的DAG构造算法如下(对基本块的每一个四元式依次执行该算法)：

(1) 若Node(B) 无定义，则构造一标记为B的叶结点并定义Node(B) 为这个结点，然后根据下列情况做不同处理：

① 若当前四元式是0型，则记Node(B)的值为n，转(4)。

② 若当前四元式是1型，则转(2)①。

③ 若当前四元式是2型，则：

i. 如果Node(C) 无定义，则构造一标记为C的叶结点，并定义Node(C) 为这个结点；

ii. 转(2)②。

(2) ① 若Node(B) 是以常数标记的叶结点，则转(2)③，否则转(3)①。

② 若Node(B) 和Node(C) 都是以常数标记的叶结点，则转(2)④，否则转(3)②。

③ 执行op B(即合并已知量)，令得到的新常数为P。若Node(B) 是处理当前四元式时新建立的结点，则删除它；若Node(P) 无定义，则构造一用P做标记的叶结点n并置Node(P)=n；转(4)。

④ 执行B op C(即合并已知量)，令得到的新常数为P。若Node(B)或Node(C)是处理当前四元式时新建立的结点，则删除它；若Node(P)无定义，则构造一用P做标记的叶结点n并置Node(P) =n；转(4)。

(3) ① 检查DAG中是否有标记为op且以Node(B) 为唯一后继的结点(即查找公共子表达式)。若有，则把已有的结点作为它的结点并设该结点为n；若没有，则构造一个新结点；转(4)。

② 检查DAG中是否有标记为op且其左后继为Node(B)、右后继为Node(C) 的结点(即查找公共子表达式)。若有，则把已有的结点作为它的结点并设该结点为n；若没有，则构造一个新结点；转(4)。

(4) 若Node(A) 无定义，则把A附加在结点n上并令Node(A) =n；否则，先从Node(A) 的附加标识符集中将A删去(注意，若Node(A) 是叶结点，则不能将A删去)，然后再把A附加到新结点n上，并令Node(A) =n。

注意：算法中步骤(2)的 ①、② 用于判断结点是否为常数，而步骤(2)的 ③、④ 则是对常数的处理。对任何一个四元式，如果其中参与运算的对象都是编译时的已知量，那么(2)并不生成计算该结点值的内部结点，而是执行该运算并用计算出的常数生成一个叶结点，所以(2)的作用是实现合并已知量。

步骤(3)的作用是检查公共子表达式。对具有公共子表达式的所有四元式，它只产生一个计算该表达式值的内部结点，而把那些被赋值的变量标识符附加到该结点上。这样，当把该结点重新写为四元式时，就删除了多余运算。

步骤(4)的功能是将(1)～(3)的操作结果赋给变量A，也即将标识符A标识在操作结果的结点n上，而执行把A从Node(A) 上的附加标识符集中删除的操作，这意味着删除了无用赋值(对A赋值后但在该A值引用之前又重新对A进行了赋值，则前一个赋值为无用赋值)。

综上所述，DAG可以在基本块内实现合并已知量、删除无用赋值和删除多余运算的优化。

例5.1 试构造以下基本块的DAG：

(1) T_0=3.14
(2) T_1=2*T_0
(3) T_2=R+r
(4) A=T_1*T_2
(5) B=A
(6) T_3=2*T_0
(7) T_4=R+r
(8) T_5=T_3*T_4
(9) T_6=R−r
(10) B= T_5*T_6

[解答] 按照算法顺序处理每一四元式后构造出的DAG如图5-2所示，其中子图 (1)～(10) 分别对应四元式(1)～(10)的DAG构造。

构造过程说明如下：

(1) 对应图5-2(2)，四元式$T_1=2*T_0$首先执行算法中的步骤(1)，因Node(B) 无定义，所以构造一个标记为2的叶结点并定义Node(2) 为这个结点。因当前四元式是2型且Node(C)已有定义(此时为Node(T0))，转算法步骤(2)

②。因Node(B) =Node(2) 和Node(C) =Node(T_0) 都是标记为常数的叶结点，则执行B op C并令新结点为P(=6.28)。由于Node(P) 无定义，故构造Node(P) =Node(6.28)。此外，因Node(B) =Node(2) 是处理当前四元式时新构造出来的结点，故删除n。接下来执行算法步骤(4)，因Node(A)无定义而将T_1附加在结点n上，并令Node(T_1) =6.28；最后DAG生成了2个结点n_1和n，因结点n2被删除而将n_3改名为n_2。图5-2(2) 的形成过程实际上也是合并已知量的优化过程。

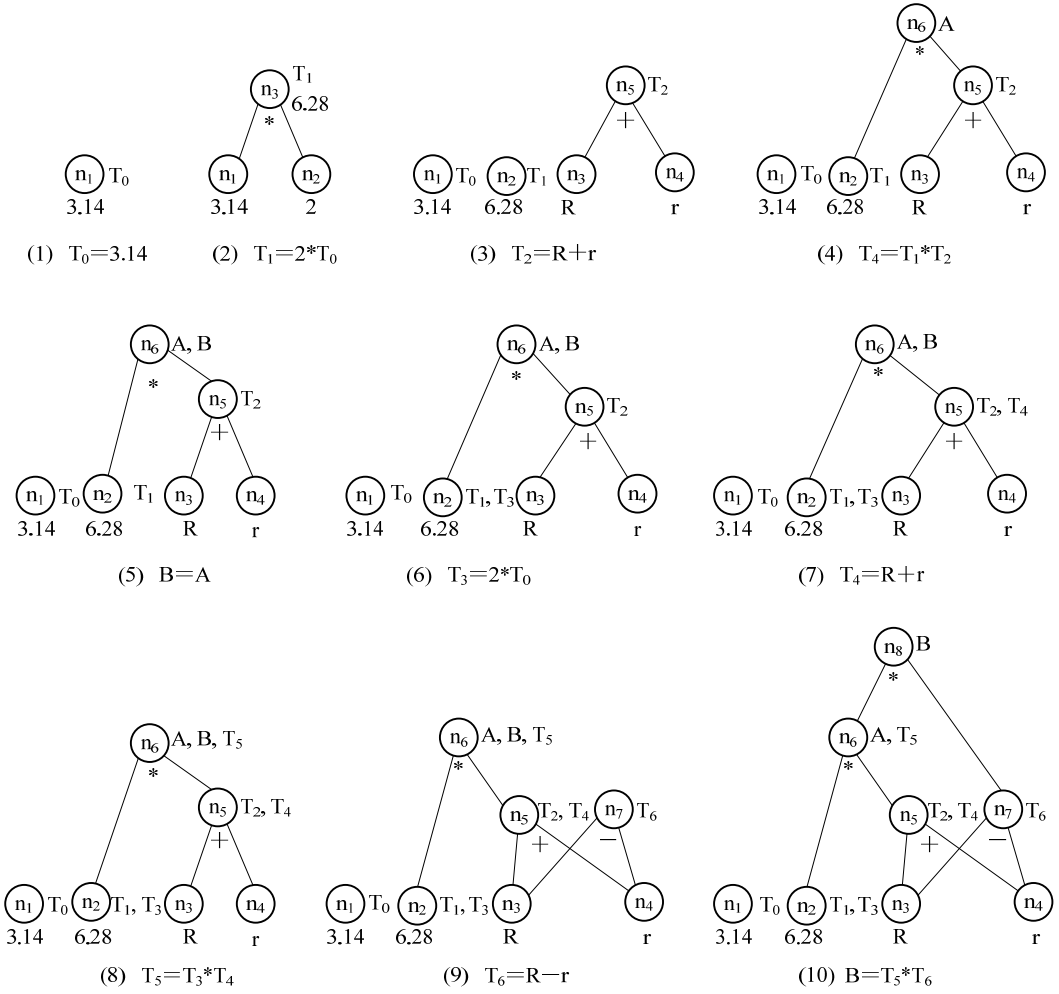

n1 T0
3.14
(1) T0=3.14
n3 T1
6.28
*
n1
n2
3.14
2
(2) T1=2*T0
n5 T2
+
n1 T0
n2 T1
n3
n4
3.14
6.28
R
r
(3) T2=R+r
n6 A
*
n5 T2
+
n1 T0
n2 T1
n3
n4
3.14
6.28
R
r
(4) T4=T1*T2
n6 A, B
*
n5 T2
+
n1 T0
n2 T1
n3
n4
3.14
6.28
R
r
(5) B=A
n6 A, B
*
n5 T2
+
n1 T0
n2 T1, T3
n3
n4
3.14
6.28
R
r
(6) T3=2*T0
n6 A, B
*
n5 T2, T4
+
n1 T0
n2 T1, T3
n3
n4
3.14
6.28
R
r
(7) T4=R+r
n6 A, B, T5
*
n5 T2, T4
+
n1 T0
n2 T1, T3
n3
n4
3.14
6.28
R
r
(8) T5=T3*T4
n6 A, B, T5
*
n5 T2, T4
n7 T6
+
−
n1 T0
n2 T1, T3
n3
n4
3.14
6.28
R
r
(9) T6=R−r
n8 B
*
n6 A, T5
*
n5 T2, T4
n7 T6
+
−
n1 T0
n2 T1, T3
n3
n4
3.14
6.28
R
r
(10) B=T5*T6

图5-2 DAG

(2) 图5-2(4) 中T_1、T_2已有定义，则仅生成一个新结点n_6并将A附加在n_6上。图5-2(5) 中结点B已有定义，故直接附加在n_6上。

(3) 图5-2(6) 的处理过程与图5-2(2) 略同，但在生成P时因其已在DAG中(即Node(6.28))，故不生成新结点而直接将T_3附加在结点6.28上。

(4) 图5-2(7) 的生成过程实质上是删除多余运算(删除公共子表达式)的优化。因为DAG中已有叶结点R与叶结点r，并且执行op操作后得到的新结点T2也已经在DAG中，故执行算法步骤(4)时因T_4无定义而将T4附加在结点n_5上。

(5) 图5-2(9) 中，变量R和r已在DAG中有相应的结点，执行“–”操作后，产生的新结点P无定义，故仅生成一个新结点n7并将T6标记于其上。

(6) 图5-2(10) 中，对当前四元式B=T_5*T_6，DAG中已有结点T_5和T_6；执行算法步骤(4)时因结点B已有定义且不是叶结点，故先将原B从DAG中删除，然后生成一个新结点n_8，将B附加其上并令Node(B) =n_8。这一处理过程实质上是删除了无用赋值B=A。

5.1.3 用DAG进行基本块的优化处理

用DAG进行基本块优化处理的基本思想是：按照构造DAG结点的顺序，对每一个结点写出其相应的四元式表示。

我们根据例5.1中DAG结点的构造顺序，按照图5-2(10)写出四元式序列G'如下：

(1) T_0=3.14
(2) T_1=6.28
(3) T_3=6.28
(4) T_2=R+r
(5) T_4=T_2
(6) A=6.28*T_2
(7) T_5=A
(8) T_6= R−r
(9) B=A*T_6

将G'和原基本块G相比，我们看到：

(1) G中四元式 (2) 和 (6) 都是已知量和已知量的运算，G'已合并。

(2) G中四元式 (5) 是一种无用赋值，G'已将它删除。

(3) G中四元式 (3) 和 (7) 的R+r是公共子表达式，G'只对它们计算了一次，即删除了多余的R+r运算。

因此，G'是对G实现上述三种优化的结果。

通过观察图5-2(10) 中的所有叶结点和内部结点以及其上的附加标识符，还可以得出以下结论：

(1) 在基本块外被定值并在基本块内被引用的所有标识符就是DAG中相应叶结点上标记的标识符。

(2) 在基本块内被定值且该值能在基本块后面被引用的标识符就是DAG各结点上的附加标识符。

这些结论可以引导优化工作的进一步深入，尤其是无用赋值的优化，也即：

(1) 如果DAG中某结点上的标识符在该基本块之后不会被引用，就可以不生成对该标识符赋值的四元式。

(2) 如果某结点ni上没有任何附加标识符，或者ni上的附加标识符在该基本块之后不会被引用，而且ni也没有前驱结点，这表明在基本块内和基本块之后都不会引用ni的值，那么就不生成计算ni结点值的四元式。

(3) 如果有两个相邻的四元式A=C op D和B=A，其中第一条代码计算出来的A值仅在第二个四元式中被引用，则将DAG中相应结点重写成四元式时，原来的两个四元式可以优化为B=C op D。

假设例5.1中T0、T1、T2、T3、T4、T5和T6在基本块后都不会被引用，则图5-2(10) 中的DAG就可重写为如下的四元式序列：

(1) S1=R+r　　//S1、S2为存放中间结果的临时变量

(2) A=6.28*S1

(3) S2=R−r

(4) B=A*S2

以上把DAG重写成四元式序列时，是按照原来构造DAG结点的顺序(即n5、n6、n7、n8)依次进行的。实际上，我们还可以采用其它顺序(如自底向上)重写，只要其中的任何一个内部结点是在其后继结点之后被重写并且转移语句(如果有的话)仍然是基本块的最后一个语句即可。

5.1.4 DAG构造算法的进一步讨论

当基本块中有数组元素引用、指针和过程调用时，构造DAG算法就较为复杂。例如，考虑如下的基本块G：

(1) x=a[i]

(2) a[j] =y

(3) z=a[i]

如果我们用构造DAG的算法来构造上述基本块的DAG，则a[i] 就是一个公共子表达式；且由所构造的DAG重写出优化后的四元式序列G'如下：

(1) x=a[i]

(2) z=x

(3) a[j] =y

如果i≠j，则G与G'是等效的。但是，如果i=j且y≠a[i]，则将y值赋给a[j] 的同时也改变了a[i] 的值(因i=j)；这时z值应为改变后的a[i] 值(即y值)，与x不等。为了避免这种情况的发生，当我们构造对数组a的元素赋值句的结点时，就“注销”所有标记为 [ ] 且左边变量是a(可加上或减去一个常数)的结点。我们认为对这样的结点再添加附加标识符是非法的，从而取消了它作为公共子表达式的资格。

对指针赋值语句*p=w(其中p是一个指针)也会产生同样的问题，如果我们不知道p可能指向哪一个变量，那么就认为它可能改变基本块中任何一个变量的值。当构造这种赋值句的结点时，就需要把DAG各结点上的所有标识符(包括作为叶结点上标记的标识符)都予以注销，这也就意味着DAG中所有的结点也都被注销。

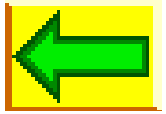
在一个基本块中的一个过程调用将注销所有的结点，因为对被调用过程的情况缺乏了解，所以我们必须假定任何变量都可能因产生副作用而发生变化。

此外，当把DAG重写成四元式时，如果我们不是按照原来构造DAG结点的顺序进行重写，那么DAG中的某些结点必须遵守一定的顺序。例如，在上述基本块G中，z=a[i]必须跟在a[j] =y之后，而a[j] =y则必须跟在x=a[i] 之后。下面，我们根据上述讨论把重写四元式时DAG中结点间必须遵守的顺序归纳如下：

(1) 对数组a中任何元素的引用或赋值，都必须跟在原来位于其前面的(如果有的话，下同)对数组a任何元素的赋值之后；对数组a任何元素的赋值，都必须跟在原来位于其前面的对数组a任何元素的引用之后。

(2) 对任何标识符的引用或赋值，都必须跟在原来位于其前面的任何过程调用或通过指针的间接赋值之后；任何过程调用或通过指针的间接赋值，都必须跟在原来位于其前面的任何标识符的引用或赋值之后。

总之，当对基本块重写时，任何数组a的引用不允许互相调换次序，并且任何语句不得跨越一个过程调用语句或者通过指针间接赋值。



5.2 循环优化

5.2.1 程序流图与循环

为了进行循环优化，必须先找出程序中的循环。由程序语言的循环语句形成的循环是不难找出的，但由条件转移语句和无条件转移语句同样可以形成程序中的循环，并且其结构可能更加复杂。因此，为了找出程序中的循环，就需要对程序中的控制流程进行分析。我们应用程序的控制流程图来给出循环的定义并找出程序中的循环。

一个控制流程图(简称流图)就是具有唯一首结点的有向图。所谓首结点，就是从它开始到控制流程图中任何一个结点都有一条通路的结点。我们可以把控制流程图表示成一个三元组G=(N,E,n0)；其中，N代表图中所有结点集，E代表图中所有有向边集，n0代表首结点。

一个程序可用一个流图来表示。流图的有限结点集N就是程序的基本块集，流图中的结点就是程序的基本块，流图的首结点就是包含程序第一个语句的基本块。流图的有向边集E是这样构成的：假设流图中结点i和结点j分别对应于程序的基本块i和基本块j，则当下述条件有一个成立时，从结点i有一条有向边引到结点j：

(1) 基本块j在程序中的位置紧跟在基本块i之后，并且基本块i的出口语句不是无条件转移语句goto (s) 或停语句。

(2) 基本块i的出口语句是goto (s) 或if…goto (s)，并且(s) 是基本块j的入口语句。

在以后的讨论中，我们所涉及的流图都是程序流图。程序流图和基本块的DAG是不同的概念。程序流图是对整个程序而言的，它表示了各基本块(对应流图中的一个结点)之间的控制关系，图中可以出现环路；DAG是对基本块而言的，是局限于该基本块内的无环路有向图，它表示了这个基本块内各四元式的操作及相互关系。

我们仍以下面求最大公因子的三地址代码程序为例来求其程序流图：

(1) read X
(2) read Y
(3) R=X % Y
(4) if R=0 goto (8)
(5) X=Y
(6) Y=R
(7) goto (3)
(8) write Y
(9) halt

我们知道，该程序的基本块分别为 (1) (2)、(3) (4)、(5) (6) (7) 和 (8) (9)。按构造流图结点间有向边的方法，我们得到该程序的程序流图如图5-3所示。

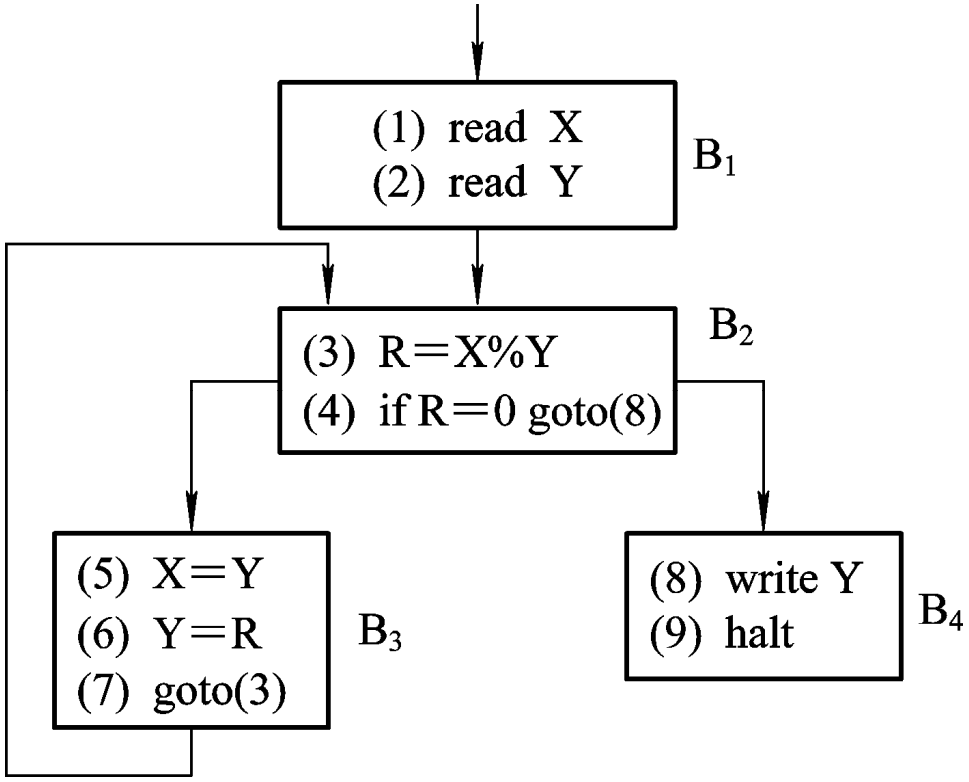


图5-3 求最大公因子的程序流图

有了程序流图，我们就可以对所要讨论的循环结构给出定义。在程序流图中，我们称具有下列性质的结点序列为一个循环：

(1) 它们是强连通的，其中任意两个结点之间必有一条通路，而且该通路上各结点都属于该结点序列；如果序列只包含一个结点，则必有一条有向边从该结点引到其自身。

(2) 它们中间有一个而且只有一个是入口结点。

所谓入口结点，是指序列中具有下述性质的结点：从序列外某结点有一条有向边引到它，或者它就是程序流图的首结点。

注意：此处定义的循环就是程序流图中具有唯一入口结点的强连通子图。从循环外要进入循环，必须先经过循环的入口结点。对于性质(1)，任意两个结点之间必有一条通路，即通路上的尾结点到首结点之间也有一条通路(实际上可认为无首尾之分)，这就构成了一个环形通路。该通路上的各结点都属于该结点序列，即从通路上的任何结点开始所构成的序列都包含该通路上的所有结点，这仍然构成了一个环形通路。因此，性质(1)是任何一种循环结构所必须具备的，否则该结点序列必有一部分是不可能反复执行的。性质(2)出于对循环优化的考虑，当需要把循环中某些代码(如不随循环反复执行而改变的运算)提到循环之外时，可以将代码提到循环结构的唯一入口结点的前面。

例如，对图5-3所示的程序流图，由上述循环的定义可知，结点序列{B2, B3}是程序中的一个循环，其中，B2是循环的唯一入口结点。

对图5-4所示的程序流图，结点序列{6}因其只有一个结点且有一有向边引到自身，并且只有唯一的入口结点6，故是我们所定义的循环。而{2, 3, 4, 5, 6, 7}中的任意两个结点之间都存在通路(即为强连通)，且有唯一的入口结点2，故也是我们所定义的循环。此外，{4, 5, 6, 7}也是强连通且有唯一入口结点4，虽然到入口结点4的有向边不止一条，但仍然是我们所定义的循环。而{2, 4}和 {2, 3, 4}，它们虽然是强连通的，但却存在两个入口结点2、4，故不是我们所定义的循环。{4, 5, 7}和{4, 6, 7}也因其存在两个入口结点4、7而不是我们所定义的循环。

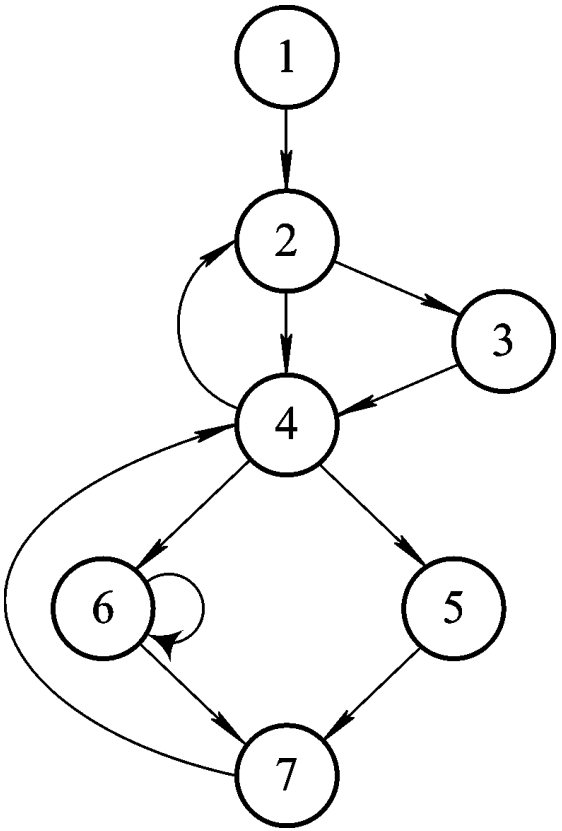


图5-4 程序流图

5.2.2 循环的查找

1. 必经结点集

为了找出程序流图中的循环，需要分析流图中结点的控制关系，为此我们引入必经结点和必经结点集的定义。

在程序流图中，对任意结点m和n，如果从流图的首结点出发，到达n的任一通路都要经过m，则称m是n的必经结点，记为m DOM n；流图中结点n的所有必经结点的集合称为结点n的必经结点集，记为D(n)。

显然，循环的入口结点是循环中所有结点的必经结点；此外，对任何结点n来说都有n DOM n。

如果把DOM看作流图结点集上定义的一个关系，则由定义容易看出它具有下述性质：

(1) 自反性：对流图中任意结点a，都有a DOM a。

(2) 传递性：对流图中任意结点a、b、c，若存在a DOM b和b DOM c, 则必有a DOM c。

(3) 反对称性：若存在a DOM b和b DOM a，则必有a=b。

因此，关系DOM是一个偏序关系，任何结点n的必经结点集是一个有序集。

例5.2 求图5-4中各结点的D(n)。

[解答] 考察图5-4并由必经结点的定义容易看出：首结点1是所有结点的必经结点；结点2是除去结点1之外所有结点的必经结点；结点4是结点4、5、6、7的必经结点；而结点3、5、6、7都只是其自身的必经结点。因此，直接由定义和DOM的性质可求得

D(1) ={1}
D(2) ={1, 2}
D(3) ={1, 2, 3}
D(4) ={1, 2, 4}
D(5) ={1, 2, 4, 5}
D(6) ={1, 2, 4, 6}
D(7) ={1, 2, 4, 7}

求流图G= (N, E, n0) 的所有结点n的必经结点集D(n) 的算法如下(其中P(n) 代表结点n的前驱结点集，它可以从边集E中直接求出)：

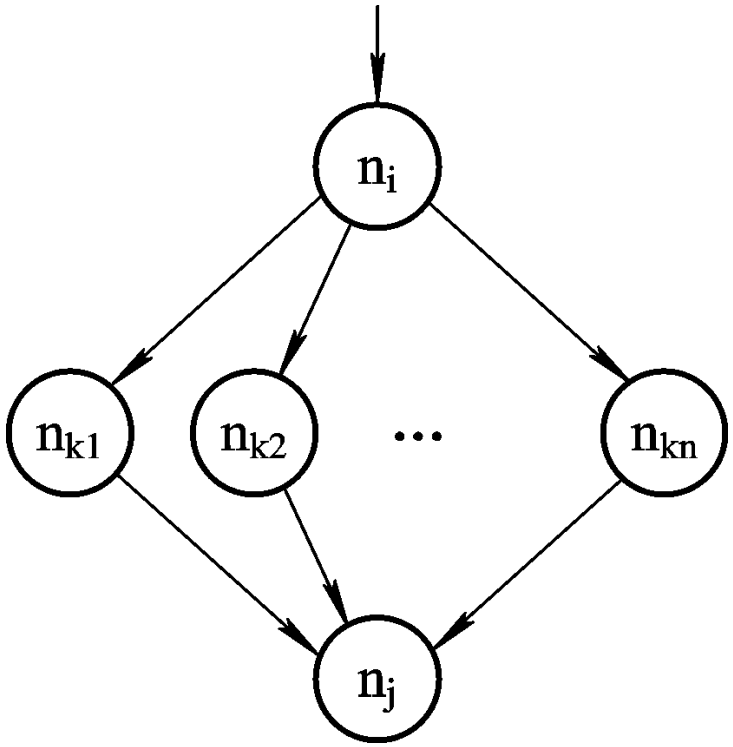


图5-5 ni为nj的必经结点示意

```
D(n0) ={n0};
for( n∈N−{n0} )   D(n) =N;                    //置初值
change=true;
while( change )
{
    change=false;
    for( n∈N−{n0} )
    {
        new={n}∪ ∩ D(p);
                 p∈P(n)
        if( new!=D(n) )
        {
            change=true;
            D(n) =new;
        }
    }
}
```

注意：由于算法中是利用所有前驱信息进行∩运算来获得某结点对应的必经结点集的，因此迭代初值$D(n_i)$必须取最大值，即全集N。此外，由 $\bigcap_{p\in P(n)} D(p)$ 知表示结点n的所有前驱(即父结点)的必经结点集的交集即为n的必经结点集。由图5-5可看出，n_i为n_j的必经结点(n_i为结点n_j所有前驱n_{k1}～n_{kn}必经结点集的交集)，而n_{k1}～n_{kn}都不是n_j的必经结点。另一点要说明的是，因程序流图中有循环情况，所以后面计算的结点其必经结点集$D(n_j)$ 的改变可能要影响到前面所计算的$D(n_i)$ 值。

因此，在一次迭代计算结束时，只要发现某一个$D(n_k)$ 被改变，就必须进行下一次迭代来计算各结点的D(n)(即算法中的while循环继续执行)，直至全部结点的D(n) 都不改变为止(即算法中的change值为false才结束算法的执行)。

例5.3 应用求流图必经结点集的算法求图5-4所示程序流图各结点n的D(n)。

[解答] 算法求解过程如下：

首先置初值：

D(1) ={1}

D(2) =D(3) =D(4) =D(5) =D(6) =D(7)

={1,2,3,4,5,6,7}

置change为false，然后从结点2到结点7依次执行第二个for循环。

对结点2，因

new={2}∪ D(1)∩D(4) ={2}∪ {1}∩{1, 2, 3, 4, 5, 6, 7}

={2}∪ {1}={1, 2}

但迭代前D(2) ={1, 2, 3, 4, 5, 6, 7}，故D(2) ≠new，因此置change=true并令D(2) ={1, 2}。

对结点3，因

new={3}∪ D(2) ={3}∪ {1, 2}={1, 2, 3}

但迭代前D(3) ={1, 2, 3, 4, 5, 6, 7}，故D(3) ≠new，因此令D(3) ={1, 2, 3}。

其余各结点按照上述步骤可求出：

D(4) ={4}∪ D(2)∩D(3)∩D(7) ={4}∪ {1, 2}∩{1, 2, 3}∩{1, 2, 3, 4, 5, 6, 7}={1, 2, 4}

D(5) ={5}∪ D(4) ={1, 2, 4, 5}

D(6) ={6}∪ D(4) ={1, 2, 4, 6}

D(7) ={7}∪ D(5)∩D(6) ={7}∪ {1, 2, 4, 5}∩{1, 2, 4, 6}={1, 2, 4, 7}

一次迭代完毕后，因change为true，故还要进行下一次迭代。

先令change为false，然后继续从结点2到结点7依次执行第二个for循环。

对结点2，因

new={2}∪ D(1)∩D(4) ={2}∪ {1}∩{1, 2, 4}

={2}∪ {1}={1, 2}

但迭代前D(2) ={1,2}，所以D(2) =new，故D(2) 不变。

对结点3，因

new={3}∪ D(2) ={3}∪ {1, 2}={1, 2, 3}

但迭代前D(3) ={1,2,3}，所以D(3) =new，故D(3) 不变。

对其余结点n(n=4～7)求出的new均有D(n) =new，所以第二次迭代后change为false，迭代结束，第一次迭代求出的各个D(n) 就是最后的结果。

2．回边

查找循环的方法是：首先应用必经结点集来求出流图中的回边，然后再利用回边找出流图中的循环。

回边的定义如下：假设a→b是流图中一条有向边，如果b DOM a，则称a→b是流图中的一条回边。

对于一已知流图G，只要求出各结点n的必经结点集，就可以立即求出流图中的所有回边。在求出流图G中的所有回边后，就可以求出流图中的循环。如果已知有向边n→d是一条回边，则由它组成的循环就是由结点d、结点n以及有通路到达n但该通路不经过d的所有结点组成的。

例5.4 求出图5-4所示程序流图的所有回边。

[解答]

(1) 已知D(6) ={1, 2, 4, 6}，因存在6→6且6 DOM 6，故6→6是回边。

(2) 已知D(7) ={1, 2, 4, 7}，因存在7→4且4 DOM 7，故7→4是回边。

(3) 已知D(4) ={1, 2, 4}，因存在4→2且2 DOM 4，故4→2是回边。

容易看出，其它有向边都不是回边。

寻找由回边组成循环的算法如下：

```
void insert( m )
{
    if( m∉loop )
    {
        loop=loop∪{m};
        把 m 压入栈 stack;
    }
}
void main( )
{
    stack=空;                          //stack 是一个工作栈
    loop={d};                          //loop 是所求的循环
    insert( m );
    while( stack 非空 )
    {
        弹出 stack 栈顶元素 m;
        for( p∈P( m ) )                //P( m )为结点 m 的前驱结点集
          insert( p );
    }
}
```

此算法中求回边n→d组成循环的所有结点的方法是：由于循环以d为其唯一入口，n是循环的一个出口，因而只要n不同时是循环入口d，那么n的所有前驱就应属于循环。在求出n的所有前驱之后，只要它们不是循环入口d，就应再继续求出它们的前驱，而这些新求出的所有前驱也应属于循环。然后再对新求出的所有前驱重复上述过程，直到所求出的前驱都是d为止。

3．可归约流图

一个流图被称为可归约的，当且仅当流图中除去回边之外，其余的边构成一个无环路流图。例如，图5-4就是一个可归约流图，而图5-6则是一个不可归约流图，因为图5-6中虽然有2→3，但没有3 DOM 2，即2→3不是一个回边，对3→2也是如此。

如果程序流图是可归约的，那么程序中任何可能反复执行的代码都会被求回边的算法纳入到一个循环当中

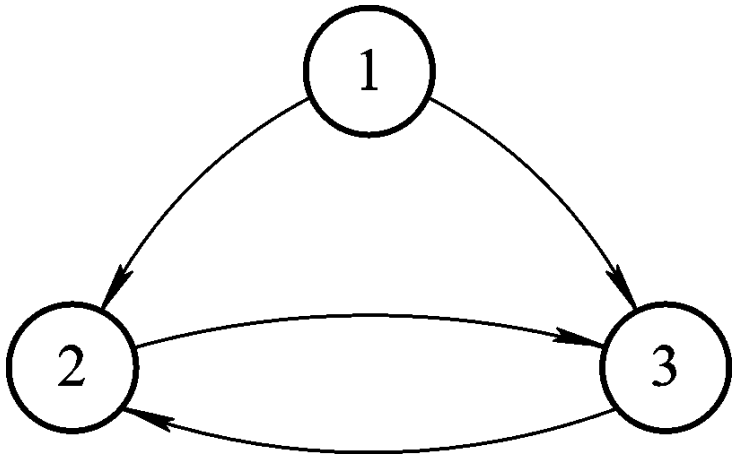


图5-6 不可归约流图

可归约流图是一类非常重要的流图，从代码优化的角度来说，它具有下述重要的性质：

(1) 图中任何直观意义下的环路都属于我们所定义的循环。

(2) 只要找出图中的所有回边，对回边应用查找循环的方法，就可以找出流图中的所有循环。

(3) 图中任意两个循环要么嵌套，要么不相交(除了可能有公共的入口结点)，对这类流图进行循环优化较为容易。

应用结构程序设计原则编写的程序，其流图总是可归约的；而应用高级语言编写的程序，其流图往往也是可归约的。

例5.5 四元式序列如下：

```
(1)        J=0;
(2) L1:    I=0;
(3)        if I< 8 goto L3;
(4) L2:    A=B+C;
(5)        B=D*C;
(6) L3:    if B=0 goto L4;
(7)        write B;
(8)        goto L5;
(9) L4:    I= I+1;
(10)       if I<8 goto L2;
(11) L5:   J= J+1;
(12)       if J≤3 goto L1;
(13)       halt
```

画出该四元式序列的程序流图G并求出G中的回边与循环。

[解答] 该四元式序列的基本块与程序流图如图5-7所示(也可用结点形式画出程序流图，如图5-8所示)。

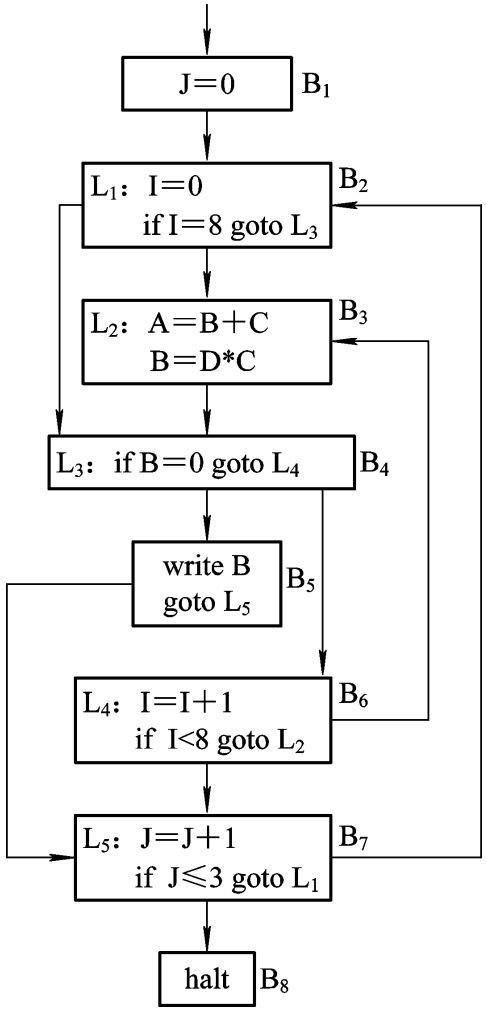


图5-7 例5.5的程序流图

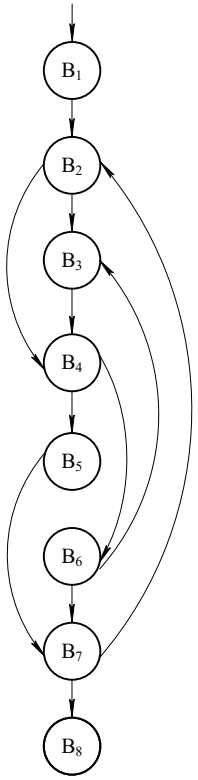


图5-8 例5.5结点形式的程序流图

由求流图G的所有结点n的必经结点集D(n) 的算法可知，初始时D(B_1) ～D(B8) 为全集{B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8}，而各结点的必经结点集的求法是：必经结点集D(B_i) 为{B_i}并上B_i所有前驱(即父结点)的必经结点集的交集。对图5-7(图5-8)中各结点的必经结点集的求法如下：

$D(B_1) = \{B_1\}$

$D(B_2) = \{B_2\} \cup D(B_1) \cap D(B_7) = \{B_2\} \cup \{B_1\} \cap \{B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8\} = \{B_1, B_2\}$

$D(B_3) = \{B_3\} \cup D(B_2) \cap D(B_6) = \{B_3\} \cup \{B_1, B_2\} \cap \{B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8\} = \{B_1, B_2, B_3\}$

$D(B_4) = \{B_4\} \cup D(B_2) \cap D(B_3) = \{B_4\} \cup \{B_1, B_2\} \cap \{B_1, B_2, B_3\} = \{B_1, B_2, B_4\}$

$D(B_5) = \{B_5\} \cup D(B_4) = \{B_1, B_2, B_4, B_5\}$

$D(B_6) = \{B_6\} \cup D(B_4) = \{B_1, B_2, B_4, B_6\}$

$D(B_7) = \{B_7\} \cup D(B_5) \cap D(B_6) = \{B_7\} \cup \{B_1, B_2, B_4, B_5\} \cap \{B_1, B_2, B_4, B_6\} = \{B_1, B_2, B_4, B_7\}$

$D(B_8) = \{B_8\} \cup D(B_7) = \{B_8\} \cup \{B_1, B_2, B_4, B_7\} = \{B_1, B_2, B_4, B_7, B_8\}$

考察流图中的有向边 $B_7 \rightarrow B_2$ 且已知 $D(B_7) = \{B_1, B_2, B_4, B_7\}$，所以有 B_2 DOM B_7，即 $B_7 \rightarrow B_2$ 是流图中的回边。容易看出，其它有向边都不是回边。

因 $B_7 \rightarrow B_2$ 是一条回边，则在流图中能够不经过结点 B_2 且有通路到达结点 B_7 的结点只有 B_3、B_4、B_5 和 B_6，故由回边 $B_7 \rightarrow B_2$ 组成的循环是$\{B_2, B_3, B_4, B_5, B_6, B_7\}$。

5.2.3 循环优化

1. 代码外提

循环中的代码要随着循环反复执行，但其中某些运算的结果并不因循环而改变，对于这种不随循环变化的运算，可以将其外提到循环外。这样，程序的运行结果仍保持不变，但程序的运行效率却提高了。我们称这种优化为代码外提。

实行代码外提时，在循环入口结点前面建立一个新结点(基本块)，称为循环的前置结点。循环前置结点以循环入口结点为其唯一后继，原来流图中从循环外引到循环入口结点的有向边改成引到循环前置结点，如图5-9所示。

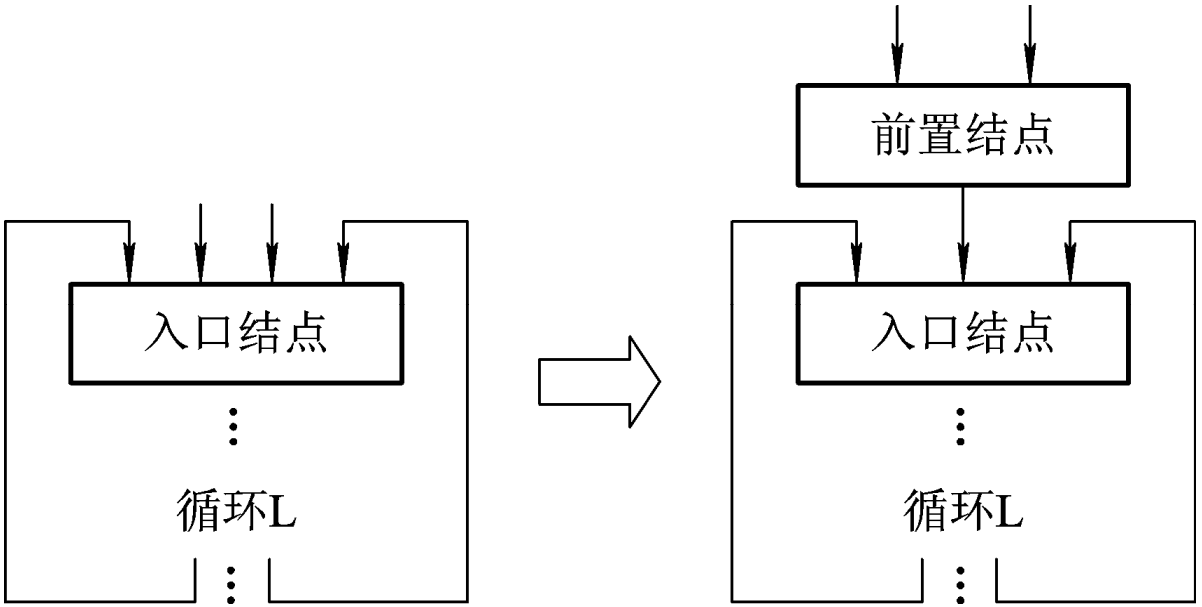


图5-9 给循环建立前置结点

因为在我们所定义的循环结构中，其入口结点是唯一的，所以前置结点也是唯一的。循环中外提的代码将统统外提到前置结点中。但是，循环中的不变运算并不是在任何情况下都可以外提的。对循环L中的不变运算S：A=B op C或A= op B或A=B，要求满足下述条件(A在离开L后仍是活跃的)：

(1) S所在的结点是L的所有出口结点的必经结点。

(2) A在L中其它地方未再定值。

(3) L中的所有A的引用点只有S中A的定值才能到达。

对上述三个条件，我们给出图5-10所示的三种流图予以说明。

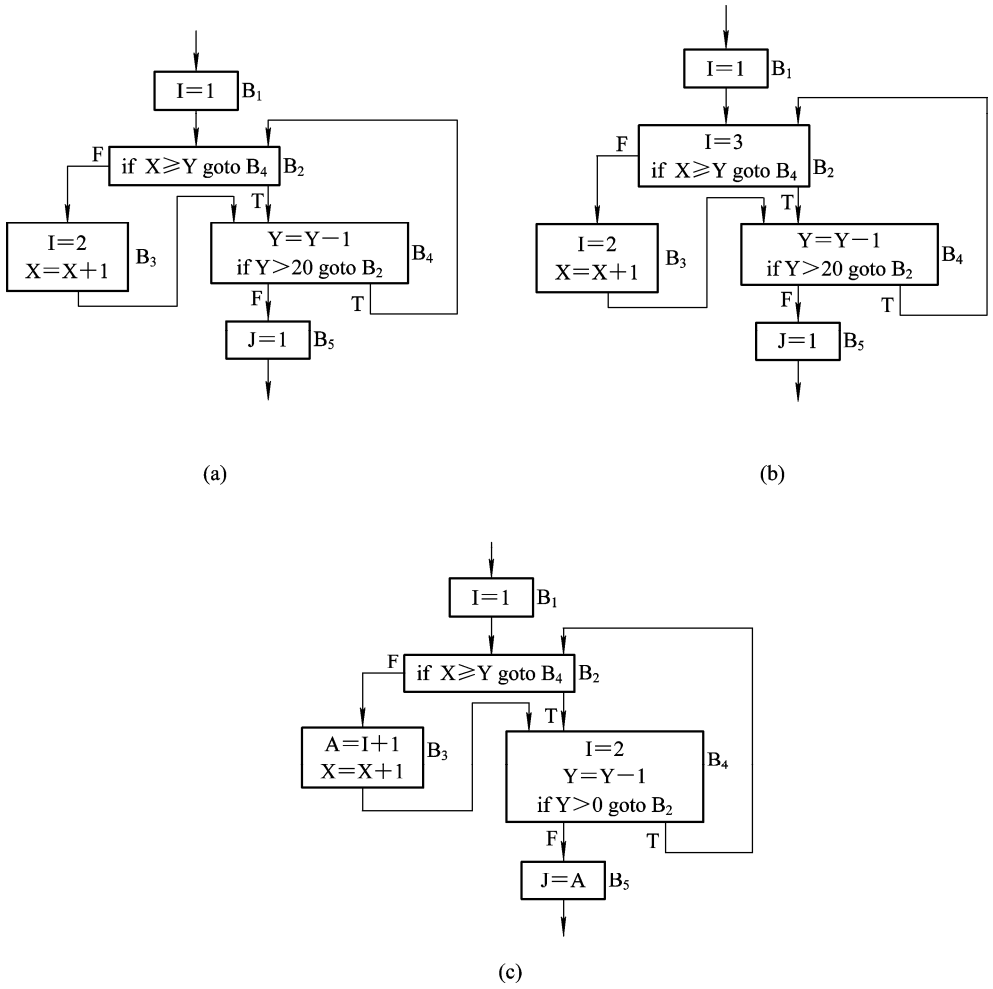


图5-10 代码外提的程序流图示例

(1) 对图5-10(a)，先将B_3中的循环不变运算I=2外提到循环前置结点中，如图5-11所示。

由图5-10(a) 可知，B_3并不是出口结点B_4的必经结点。如果令X=25，Y=22，则按图5-10(a) 的程序流图，B_3是不会执行的；于是，当执行到B_5时，I的值是1。但是，如果按图5-11执行，则执行到B_5时，I的值总是2，因此图5-11改变了原来程序运行的结果。出现以上问题是因为B_3不是循环出口结点B_4的必经结点，因此当把一不变运算外提到循环前置结点时，要求该不变运算所在的结点是循环所有出口结点的必经结点。

(2) 考查图5-10(b)，现在I=3所在的结点B_2是循环出口结点的必经结点，但循环中除B_2外，B_3也对I定值。如果把B_2中的I=3外提到循环前置结点中，且循环前X=21和Y=22，此时循环的执行顺序是$B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$，则到达$B_5$时I值为2；但如果不把$B_2$中的I=3外提，则经过以上执行顺序到达B5时，I值为3。由此可知，当把循环中的不变运算A=B op C外提时，要求循环中其它地方不再有A的定值点。

(3) 考查图5-10(c)，不变运算I=2所属结点B_4本身就是出口结点，而且此循环只有一个出口结点，同时循环中除B4外其它地方没有I的定值点，因此它满足外提的条件(1)、(2)。我们注意到，对循环中B_3的I的引用点，不仅B_4中I的定值能够到达，而且B_1中I的定值也能到达。现在考虑进入循环前X=0和Y=2时的情况，此时循环的执行顺序为$B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$，当到达$B_5$时A值为2；但如果把$B_4$中的I=2外提，则到达$B_5$时A值为3。因此当把循环不变运算A=B op C外提时，要求循环中A的所有引用点都是而且仅仅是该定值所能到达的。

根据以上讨论，给出查找所需处理的循环L中的不变运算和代码外提的算法如下：

(1) 依次查看L中各基本块的每个四元式，如果它的每个运算对象为常数或者定值点在L外，则将此四元式标记为“不变运算”。

(2) 依次查看尚未被标记为“不变运算”的四元式，如果它的每个运算对象为常数或定值点在L之外，或只有一个到达-定值点且该点上的四元式已标记为“不变运算”，则把被查看的四元式标记为“不变运算”。

(3) 重复第(2)步直至没有新的四元式被标记为“不变运算”为止。

例如，循环中的A=3已标记为“不变运算”，则对循环中A=3定值点可唯一到达的X=A+2也标记为“不变运算”。

找出了循环的不变运算就可以进行代码外提了。代码外提算法如下：

(1) 求出循环L的所有不变运算。

(2) 对步骤(1)所求得的每一不变运算S：A=B op C或A=op B或A=B，检查它是否满足以下条件：

① i. S所在的结点是L的所有出口结点的必经结点；

ii. A在L中其它地方未再定值；

iii. L中所有A的引用点只有S中A的定值才能到达。

② A在离开L后不再是活跃的(即离开L后不会引用该A值)，并且条件 ① 的ii和iii两条成立。所谓A在离开L后不再是活跃的，是指A在L的任何出口结点的后继结点(当然是指那些不属于L的后继)的入口处不是活跃的。

(3) 按步骤(1)所找出的不变运算的顺序，依次把步骤(2)中满足条件的不变运算S外提到L的前置结点中。但是，如果S的运算对象(B或C)是在L中定值的，那么只有当这些定值四元式都已外提到前置结点中时，才可把S也外提到前置结点中(B、C的定值四元式提到前置结点后，S的运算对象B、C就属于定值点在L之外了，因此也就是真正的“不变运算”了)。

注意：如果把满足条件(2)② 的不变运算A=B op C外提到前置结点中，则执行完循环后得到的A值可能与不进行外提的情形所得的A值不同，但因为离开循环后不会引用该A值，所以这不会影响程序的运行结果。

例5.6 试对图5-12给定的程序流图进行代码外提优化。

[解答] 确定不变运算的原则是依次查看循环中各基本块的每个四元式，如果它的每个运算对象为常数或者定值点在循环外，则将此四元式标记为“不变运算”。查看图5-12所示的程序流图，可以找出的不变运算是B_3中的I=2和B_4中的J=M+N。

进行代码外提时，只能将J=M+N提到循环前置结点。因为B_3中的I=2虽然是不变运算，但B_3不是循环所有出口结点的必经结点，且循环中所有I的引用点并非只有B_3的I定值能够到达，故B_3中的I=2不能外提。最后，得到代码外提后的程序流图如图5-13所示。

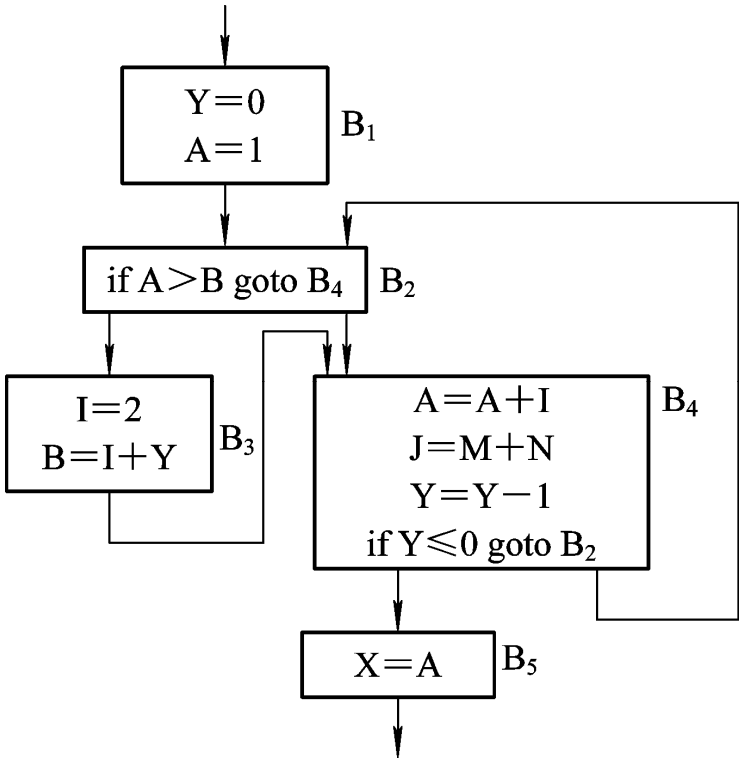


图5-12 例5.6的程序流图

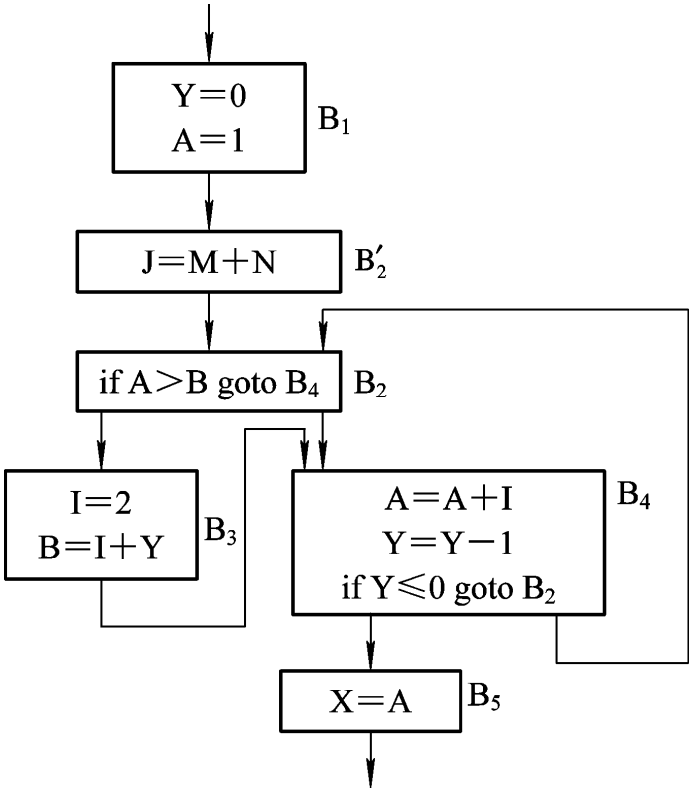


图5-13 代码外提后的程序流图

2．强度削弱

强度削弱是指把程序中执行时间较长的运算替换为执行时间较短的运算。强度削弱不仅可对乘法运算实行(将循环中的乘法运算用递归加法运算来替换)，对加法运算也可实行。

如果循环中有I的递归赋值I=I±C(C为循环不变量)，并且循环中T的赋值运算可化归为T=K*I±C1(K和C1为循环不变量)，那么T的赋值运算可以进行强度削弱。

。

进行强度削弱后，循环中可能出现一些新的无用赋值，如果它们在循环出口之后不是活跃变量，则可以从循环中删除。此外，对下标变量地址计算来说，强度削弱实际就是实现下标变量地址的递归计算

例5.7 试对图5-14给定的程序流图进行强度削弱优化。

[解答] 由图5-14所示的流图可以看出，B_2中的A=K*I和B=J*I因计算K、J的四元式都在循环之外，故可将K、J看作常量，而每次循环I=I+1即I增加1时，对应A=K*I和B=J*I分别增加K和J，因此可以将A=K*I和B=J*I外提到前置结点中B_2'，同时在B_2的I=I+1之后分别给A和B增加一个常量K和J。进行强度削弱后的流图如图5-15所示。

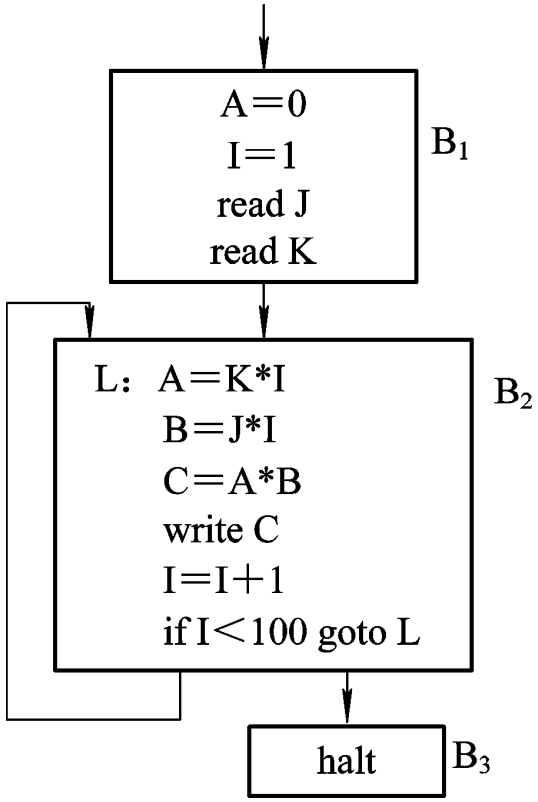


图5-14 例5.7的程序流图

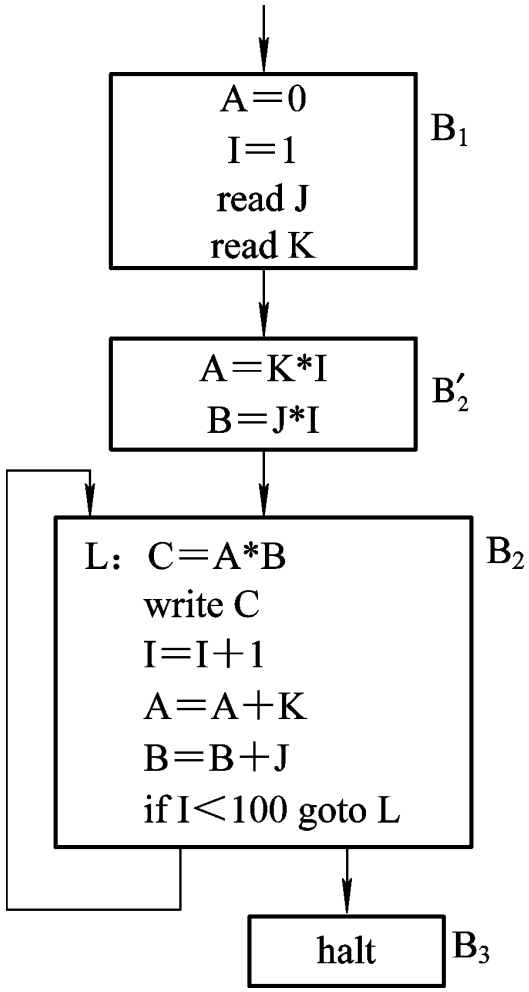


图5-15 例5.7强度削弱后的流图

例5.8 试对图5-16给定的程序流图进行强度削弱优化。

[解答] 强度削弱不仅可对乘法运算进行，也可对加法运算进行。由于本题中的四元式程序不存在乘法运算，所以只能进行加法运算的强度削弱。从图5-16中可以看到，B_2中的C=B+I，B的定值点在循环之外，故相当于常数；而另一加数I值由B_3中的I=I+1决定，即每循环一次I值增1，也即每循环一次，B_2中C=B+I的C值增量与B_3中的I相同，为常数1。因此，我们可以对C进行强度削弱，即将B_2中的四元式C=B+I外提到前置结点中 B_2' ，同时在B_3中I=I+1之后给C增加一个常量1。进行强度削弱后的结果如图5-17所示。

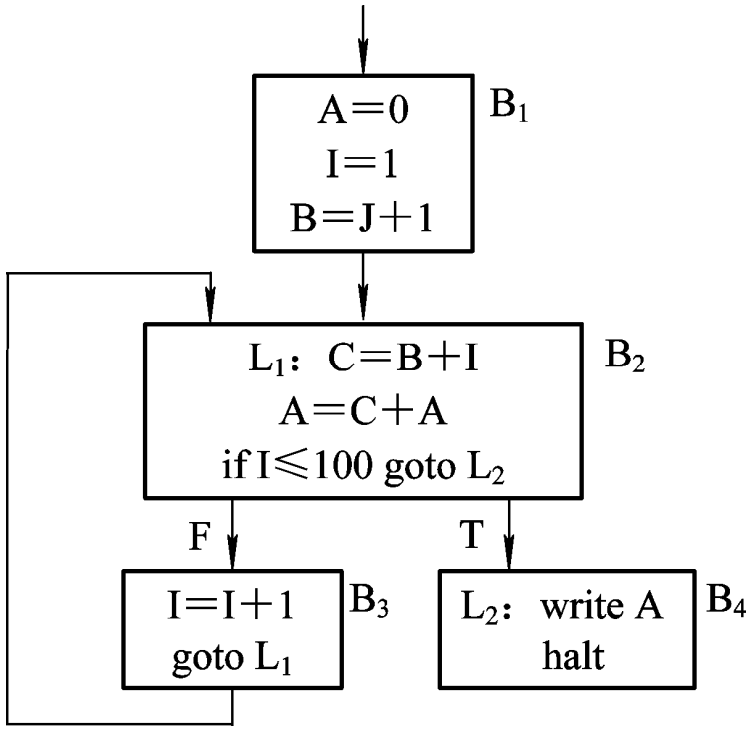


图5-16 例5.8的程序流图

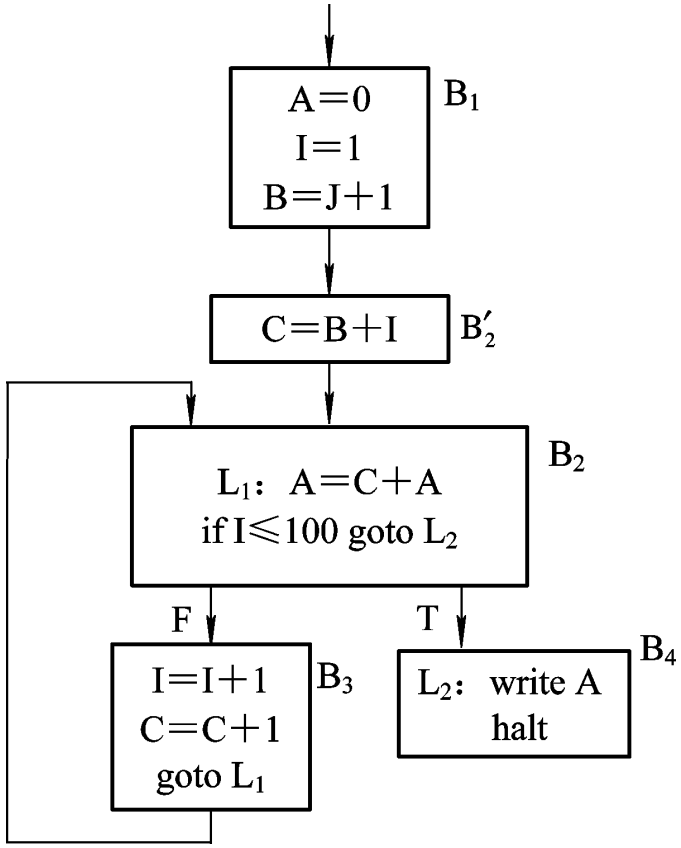


图5-17 例5.8强度削弱后的流图

例5.9 试对图5-18给定的程序流图进行强度削弱优化。

[解答] 由图5-18的B3看到，T_2是递归赋值的变量，每循环一次增加一个常量10。因$T_3=T_2+T_1$，计算T_3值时要引用T_2的值，它的另一运算对象是循环不变量T_1，所以每循环一次，T_3值的增量与T_2相同，即常数10。因此，我们可以对T_3进行强度削弱，即将$T_3=T_2+T_1$外提到前置结点中 B_2'，同时在$T_2=T_2+10$的后面给T_3增加一个常量10。进行以上强度削弱后的结果如图5-19所示。

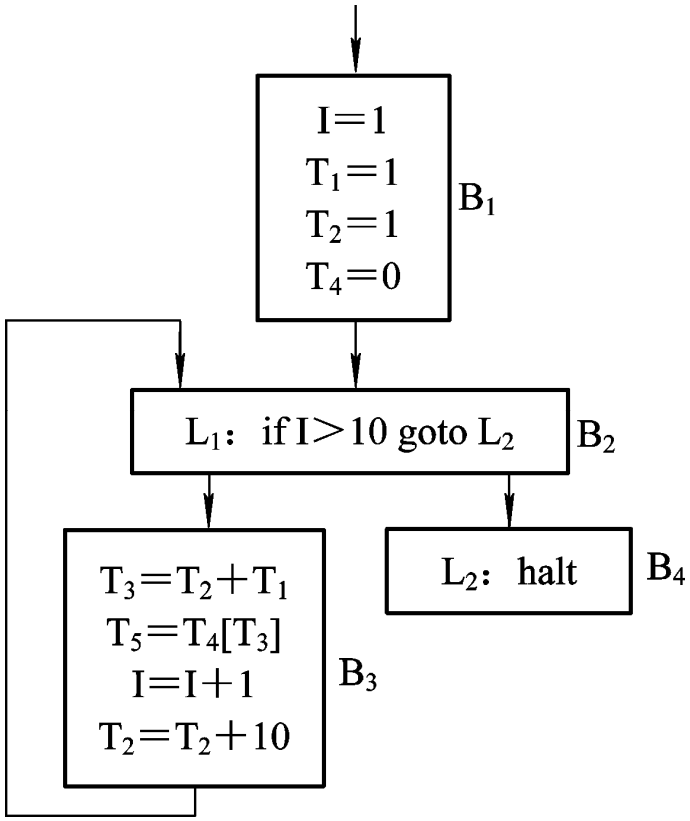


图5-18 例5.9的程序流图

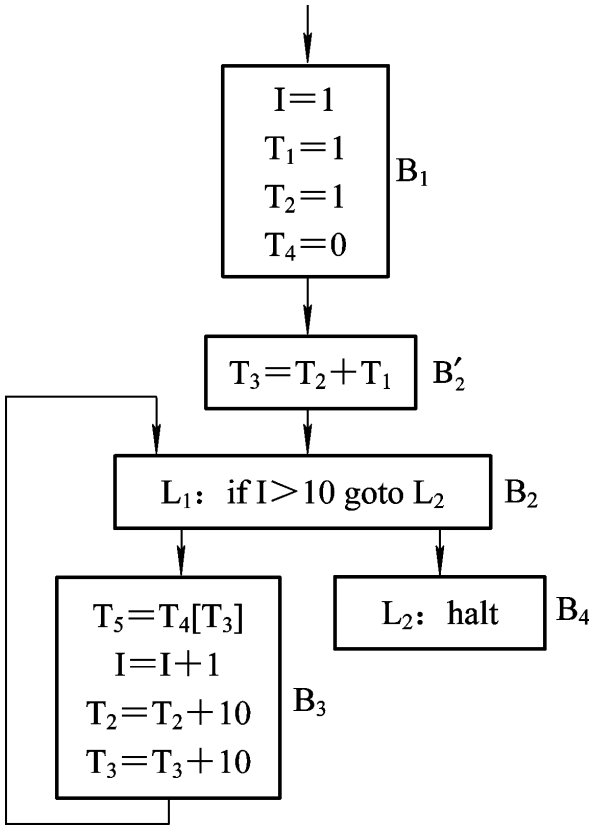


图5-19 例5.9强度削弱后的程序流图

3．删除归纳变量

如果循环中对变量I只有唯一的形如I=I±C的赋值，且其中C为循环不变量，则称I为循环中的基本归纳变量。

如果I是循环中一基本归纳变量，J在循环中的定值总是可化归为I的同一线性函数，也即$J=C_1*I\pm C_2$，其中C_1和C_2都是循环不变量，则称J是归纳变量，并称它与I同族。一个基本归纳变量也是一归纳变量。

一个基本归纳变量除用于其自身的递归定值外，往往只在循环中用来计算其它归纳变量以及控制循环的进行。此时，可以用同族的某一归纳变量来替换循环控制条件中的这个基本归纳变量，从而达到将这个基本归纳变量从程序流图中删去的目的。这种优化称为删除归纳变量或变换循环控制条件。

由于删除归纳变量是在强度削弱以后进行的，因此，我们一并给出强度削弱和删除归纳变量的算法。

(1) 利用循环不变运算信息，找出循环中所有的基本归纳变量。

(2) 找出所有其它归纳变量A，并找出A与已知基本归纳变量X的同族线性函数关系FA(X)；即：

① 在L中找出形如A=B*C、A=C*B、A=B/C、A=B±C、A=C±B的四元式，其中B是归纳变量，C是循环不变量。

② 假设找出的四元式为S：A=C*B，这时有：

i. 如果B就是基本归纳变量，则X就是B，A与基本归纳变量B是同族的归纳变量，且A与B的函数关系就是FA(B) =C*B。

ii. 如果B不是基本归纳变量，假设B与基本归纳变量D同族且它们的函数关系为FB(D)，那么如果L外B的定值点不能到达S且L中B的定值点与S之间未曾对D定值，则X就是D，A与基本归纳变量D是同族的归纳变量，且A与D的函数关系是FA(D) = C*B=C*FB(D)。

(3) (强度削弱)对(2)中找出的每一归纳变量A，假设A与基本归纳变量B同族，而且A与B的函数关系为$FA(B)=C_1*B+C_2$，其中C_1和C_2均为循环不变量，C_2可能为0，执行以下步骤：

① 建立一个新的临时变量$S_{FA(B)}$。如果两个归纳变量A和A'都与B同族且$F_{A(B)}=F_{A'(B)}$，则只建立一个临时变量$S_{FA(B)}$。

② 在循环前置结点原有的四元式后面增加下面的四元式：

$$S_{F_A(B)}=C_1*B$$

$$S_{F_A(B)}=S_{F_A(B)}+C_2 \qquad \text{(注：实现 } A=C_1*B+C_2\text{)}$$

如果$C_1=0$，则无四元式 $S_{F_A(B)}=S_{F_A(B)}+C_2$。

③ 把循环中原来对A赋值的四元式改为$A=S_{FA(B)}$。

④ 在循环中基本的归纳变量B的唯一赋值B=B±E(E是循环不变量)后面增加下面的四元式：

$S_{F_A(B)} = S_{F_A(B)} \pm C_1 * E$ (注：B 增减 E 则 A 应相应地增减 $C_1 * E$，即为 $S_{F_A(B)}$ 增减 $C_1 * E$)

如果$C_1 \neq 1$且E是变量名，则上式为

$$T = C_1 * E$$
$$S_{F_A(B)} = S_{F_A(B)} \pm T$$

其中T为临时变量(由于一个四元式只能完成一个运算，故此处出现两个四元式)。

(4) 依次考察第(3)步中每一归纳变量A，如果在$A=S_{FA(B)}$与循环中任何引用A的四元式之间没有对$S_{FA(B)}$的赋值且A在循环出口之后不活跃，则删除$A=S_{FA(B)}$并把所有引用A的地方改为引用$S_{FA(B)}$。

(5) (删除基本归纳变量)如果基本归纳变量B在循环出口之后不是活跃的，并且在循环中除在其自身的递归赋值中被引用外，只在形为if B rop Y goto Z(或if Y rop B goto Z)中被引用，则：

① 选取一与B同族的归纳变量M，并设FM(B) =C_1*B+C_2(尽可能使所选M的FM(B)简单，并且可能的话，使M是循环中其它四元式要引用的或者是循环出口之后的活跃变量)。

② 建立一临时变量R，并用下列四元式：

R=C1* Y　　//如果C1=1则C1不出现

R=R+ C2　　//如果C2=0则无此四元式

if FM(B) rop R goto Z　(或if R rop FM(B) goto Z)

来替换if B rop Y goto Z(或if Y rop B goto Z)，即将原判断条件B rop Y改为 (C_1*B+C_2) rop (C1*Y+C_2)，也就是FM(B) rop R。

③ 删除循环中对B递归赋值的四元式。

例5.10 试对图5-15给定的程序流图进行删除归纳变量优化。

[解答] 由图5-15可知，循环中I是基本归纳变量，A、B是与I同族的归纳变量且具有如下的线性关系：

$$A=K*I$$

$$B=J*I$$

因此，循环控制条件I<100完全可用A<100*K或B<100*J来替代。这样，基本块B2中的控制条件和控制语句可改写为

T1=100*K

if A<T1 goto L

或者改写为

T2=100*J

if A<T2 goto L

此时的程序流图如图5-20所示。

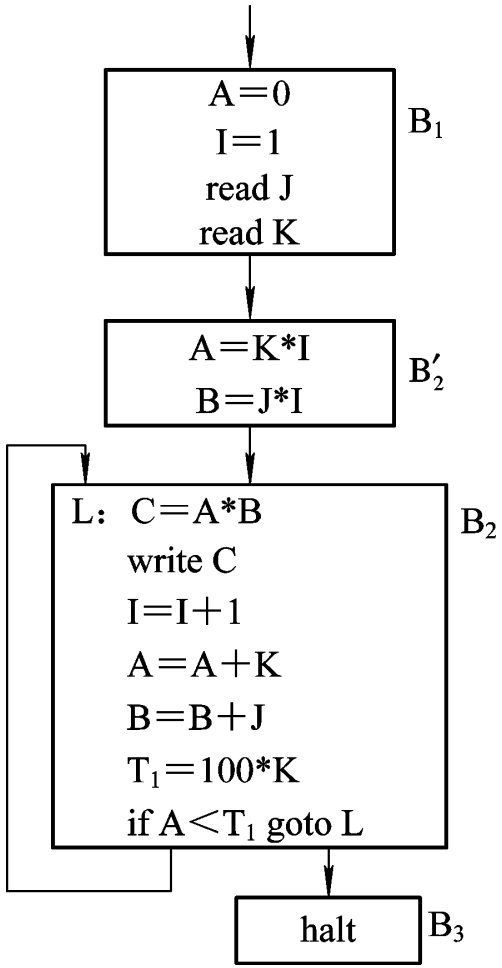


图5-20 变换循环控制条件的程序流图

循环控制条件经过以上改变之后，就可以删除基本块B_2中的语句I=I+1；而语句T_1=100*K是循环中的不变运算，故可由基本块B_2外提到基本块 B_2' 中。最后，经删除归纳变量及代码外提后得到的程序流图如图5-21所示。

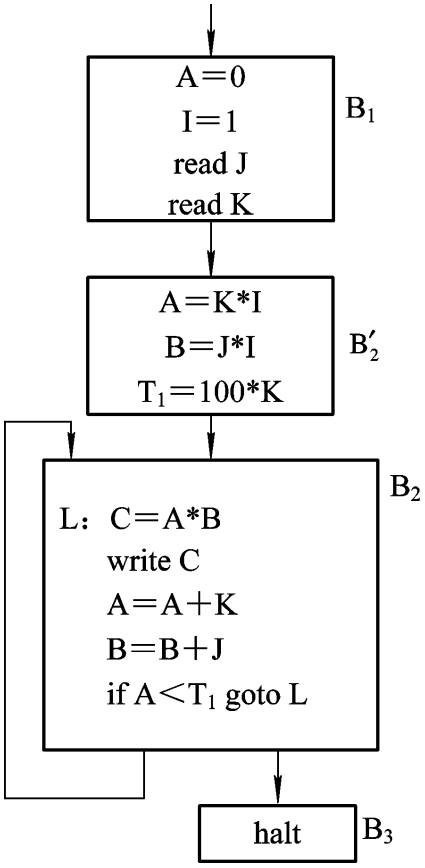


图5-21 删除归纳变量及代码外提后的程序流图

5.3 全局优化概述

全局优化是在整个程序范围内，对程序进行全面分析而进行的优化。在此，我们仅就到达-定值的有关内容和复写传播这种全局优化方法作一介绍。我们约定，本节所提到程序中某一“点”，均指某个四元式的位置(地址或编号)。

5.3.1 到达-定值与引用-定值链

为了进行全局优化，需要分析程序中所有变量的定值(即对变量赋值)和引用之间的关系。首先，我们介绍下面两个重要概念。

(1) 到达-定值：所谓变量A在某点d的定值到达另一点p，是指程序流图中从d有一通路到达p且该通路上没有A的其它定值。

(2) 引用-定值链(简称为ud链)：假设在程序中某点p引用了变量A的值，则把能够到达p的A的所有定值点的全体，称为A在引用点p的引用-定值链。

注意：可能在程序的不同分支道路上都有A的定值点，而这些定值点都可以直接到达点p；也即，这些定值点即为到达p的变量A的所有定值点全体。

为了求出到达点p的各个变量的所有定值点，我们先对程序中所有基本块B作如下定义：

IN[B]——到达基本块B入口之前的各个变量的所有定值点集。

OUT[B]——到达基本块B出口之后的各个变量的所有定值点集。

GEN[B]——基本块B中定值的并到达B出口之后的所有定值点集。

KILL[B]——基本块B外满足下述条件的定值点集：这些定值点所定值的变量在B中已被重新定值。

也即，GEN[B] 为基本块B所“生成”的定值点集，KILL[B] 为被基本块B“注销”的定值点集；GEN[B] 和KILL[B] 均可从给定的程序流图直接求出。

我们先对程序中的所有基本块B求出其IN[B]；一旦求出所有基本块B的IN[B]，就可按下述规则求出到达B中某点p的任一变量A的所有定值点。

(1) 如果B中p的前面有A的定值，则到达p的定值点是唯一的，它就是与p最靠近的那个A的定值点。

(2) 如果B中p的前面没有A的定值，则到达p的A的所有定值点就是IN[B] 中A的那些定值点。

怎样求得IN[B] 和OUT[B] 呢？

对于OUT[B] 容易看出：

(1) 如果定值点d在GEN[B] 中，那么它一定也在OUT[B] 中。

(2) 如果某定值点d在IN[B] 中且被d定值的变量在B中没有被重新定值，则d也在OUT[B] 中。

(3) 除(1)、(2)外没有其它定值点d能够到达B的出口之后，即d $\notin$ OUT[B]。

对于IN[B] 可以看出：某定值点d到达基本块B的入口之前，当且仅当它到达B的某一前驱基本块的出口之后。

综上所述，我们可得到所有基本块B的IN[B] 和OUT[B]的计算公式：

$$OUT[B] = IN[B] - KILL[B] \cup GEN[B]$$

$$IN[B] = \bigcup_{p \in P[B]} OUT[p]$$

在此，P[B] 代表B的所有前驱基本块(即B的父结点)的集合；OUT[B] 意为所有进入B前并在B中没有被修改过的某变量定值点集与B中所“生成”的该变量定值点集的并集，即先计算IN[B] -KILL[B]，然后再与GEN[B] 相“∪ ”。由于所有GEN[B] 和KILL[B] 可以从给定的程序流图中直接求出，故上式是变量IN[B] 和OUT[B] 的线性联立方程并被称之为到达-定值数据流方程。

IN[B]、OUT[B]、GEN[B] 和KILL[B] 均可用位向量来表示。于是上述公式中运算符“∪ ”可用“或”、运算符“-”可用“与非”代替；也即IN[B] -KILL[B] 可表示为IN ∧ ¬ [B]KILL[B]。

设程序流图含有n个结点，则到达-定值数据流方程可用下述迭代算法求解。

```
(1)     for( i=1; i<=n; i++ )
(2)     {   IN[Bi] = ϕ;
(3)     OUT[Bi] =GEN[Bi]; }                    //置初值
(4)     change=true;
(5)     while( change )
(6)     {   change=false;
(7)         for( i=1; i<=n; i++ )
(8)         {   NEWIN= ⋃ OUT[p];
                       p∈P[B]
(9)             if( NEWIN!=IN[Bi] )
(10)            {   change=true;
(11)                IN[Bi] =NEWIN;
(12)                OUT[Bi] =( IN[Bi] -KILL[Bi] )∪GEN[Bi];
(13)            }
(14)        }
(15)    }
```

在上述算法第(3)行中，如果不给OUT[B_i] 赋初值，则无法进行后面的∪ OUT[p] 计算(结果总为Φ)，这将使得IN[Bi]计算没有变化(始终为Φ)，所以必须先给OUT[B_i] 赋初值，而这个初值只能是基本块B_i所产生的GEN[B_i]。在第(7)行中，我们按程序流图中各结点的深度为主次序依次计算各基本块的IN和OUT。change是用来判断结束的布尔变量；NEWIN是集合变量。对每一基本块B_i如果前后两次迭代计算出的NEWIN值不等，则置change为true，表示尚需进行下一次迭代。这是因为程序中可能存在着循环，即后面结点(基本块)IN和OUT的改变可能又影响到前面已计算过的结点之IN和OUT值。所以，只要出现某结点的IN和OUT发生变化的情况，迭代就得继续下去。

例5.11 考察图5-22所示的程序流图，各四元式左边的d分别代表该四元式的位置，假设只考虑变量i和j，求其到达-定值数据流方程的解。

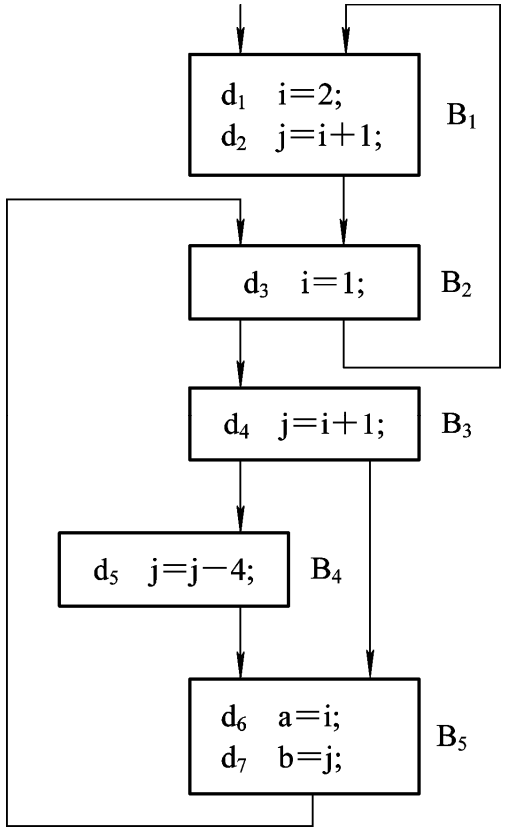


图5-22 程序流图

[解答] (1) 先求出所有基本块B的GEN[B] 和KILL[B]。GEN[B] 和KILL[B] 用位向量来表示，程序流图中每一点d在向量中占一位；如果d属于某个集，则该向量的相应位为1，否则为0。由定义直接计算出的GEN[B] 和KILL[B] 值见表5.1。

表 5.1 程序流图的 GEN[B]和 KILL[B]值

基本块 B	GEN[B]	位向量	KILL[B]	位向量
B_1	$\{d_1,d_2\}$	1100000	$\{d_3,d_4,d_5\}$	0011100
B_2	$\{d_3\}$	0010000	$\{d_1\}$	1000000
B_3	$\{d_4\}$	0001000	$\{d_2,d_5\}$	0100100
B_4	$\{d_5\}$	0000100	$\{d_2,d_4\}$	0101000
B_5	Φ	0000000	Φ	0000000

(2) 图5-23是图5-22程序流图的深度为主扩展树。各基本块的深度为B_1、B_2、B_3、B_4、B_5。根据上述迭代算法求解步骤如下：

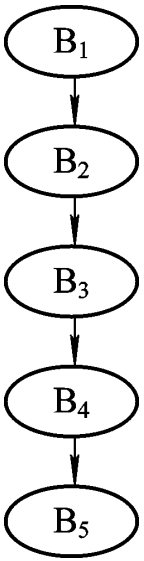


图5-23 深度为主扩展树

首先置迭代初值：

$IN[B_1] = IN[B_2] = IN[B_3] = IN[B_4] = IN[B_5] = 0000000$

$OUT[B_1] = GEN[B_1] = 1100000$

$OUT[B_2] = GEN[B_2] = 0010000$

$OUT[B_3] = GEN[B_3] = 0001000$

$OUT[B_4] = GEN[B_4] = 0000100$

$OUT[B_5] = GEN[B_5] = 0000000$

按深度为主次序对 B_1、B_2、B_3、B_4、B_5 执行算法计算：

对 B_1：因 $NEWIN=OUT[B_2] = 0010000 \neq IN[B_1]$，所以

change=true

$IN[B_1] = 0010000$

$OUT[B_1] = (IN[B_1]-KILL[B_1]) \cup GEN[B_1] = IN[B_1] \wedge \neg KILL[B_1] \cup GEN[B_1]$

$= 0010000 \wedge 1100011 \cup 1100000 = 0000000 \cup 1100000$

$= 1100000$

对 B_2：因 $NEWIN=OUT[B_1] \cup OUT[B_5] = 1100000 \cup 0000000 = 1100000 \neq IN[B_2]$，故

$IN[B_2] = 1100000$

$OUT[B_2] = (IN[B_2] - KILL[B_2]) \cup GEN[B_2] = 0110000$

同理对 B_3：

$IN[B_3] = OUT[B_2] = 0110000$

$OUT[B_3] = (IN[B_3] - KILL[B_3]) \cup GEN[B_3] = 0011000$

B_4：$IN[B_4] = OUT[B_3] = 0011000$

$OUT[B_4] = (IN[B_4] - KILL[B_4]) \cup GEN[B_4] = 0010100$

B_5：$IN[B_5] = OUT[B_3] \cup OUT[B_4] = 0011000 \cup 0010100 = 0011100$

$OUT[B_5] = (IN[B_5] - KILL[B_5]) \cup GEN[B_5] = 0011100$

各次迭代结果见表5.2，因第四次迭代计算出的结果与第三次迭代结果相同，故它就是最后所求的结果。

表 5.2 程序流图的 IN 和 OUT

基本块	第一次		第二次		第三次		第四次	
	IN[B]	OUT[B]	IN[B]	OUT[B]	IN[B]	OUT[B]	IN[B]	OUT[B]
B_1	0010000	1100000	0110000	1100000	0111100	1100000	0111100	1100000
B_2	1100000	0110000	1111100	0111100	1111100	0111100	1111100	0111100
B_3	0110000	0011000	0111100	0011000	0111100	0011000	0111100	0011000
B_4	0011000	0010100	0011000	0010100	0011000	0010100	0011000	0010100
B_5	0011100	0011100	0011100	0011100	0011100	0011100	0011100	0011100

我们知道，程序流图中IN[B_2] 处为循环入口，迭代继续的原因是后面结点计算出的IN和OUT又随循环影响到前面结点的IN和OUT值，故在此只要两次迭代的IN[B_2] 不发生变化，就无需继续迭代。在表5.2中，由于循环入口处的IN[B2] 在第二次和第三次值(下划线处)是相同的，故无需再进行第四次迭代了。

我们可以应用到达-定值信息来计算各个变量在任何引用点的引用-定值链(ud链)，即先找出其所有的引用点，然后再应用规则求各点的ud链。这个规则就是：

(1) 如果在基本块B中，变量A的引用点p之前有A的定值点d，并且A在点d的定值到达p，那么A在点p的ud链就是{d}。

(2) 如果在基本块B中，变量A的引用点p之前没有A的定值点，那么IN[B] 中A的所有定值点均到达p，它们就是A在点p的ud链。

5.3.2 定值-引用链(du链)

我们已经知道了如何计算一个变量A在引用点的ud链，即能到达该引用点的A的所有定值点。反之，对一个变量A在某点p的定值，也可计算该定值能够到达的A的所有引用点，它称为该定值点的定值-引用链(简称du链)。

对程序中某变量A和某点p，如果存在一条从p开始的道路，其中引用了A在点p的值，则称A在点p是活跃的。对于基本块B，如果OUT[B] 仅给出哪些变量的值在B的后继中还会使用的信息，而且还一同指出它们在B的后继中哪些点会被使用，那么就可直接应用这种信息来计算B中任一变量A在定值点p的du链。在此，只要对B中p后面部分进行扫描：如果B中p后面没有A的其它定值点，则B中p后面A的所有引用点加上OUT[B] 中A的所有引用点就是A在定值点p的du链；如果B中p后面有A的其它定值点，则从p到p距离最近的那个A的定值点之间的A的所有引用点，就是A在定值点p的du链。所以，问题归结为如何计算出带有上述引用点信息的OUT[B]。

我们定义：

IN[B]——基本块B入口之前的活跃变量集(进入基本块B时，哪些变量A的定值能够到达B和B的后继中A的哪些引用点)。

OUT[B]——基本块B出口之后的活跃变量集(离开基本块B时，哪些变量A的定值能够到达B的后继中A的哪些引用点)。

USE[B]——所有含 (p, A) 的集，其中p是B中某点，p引用变量A的值且B中在p前面没有A的定值点。

DEF[B]——所有含 (p, A) 的集，其中p是不属于B的某点，p引用变量A的值但A在B中被重新定值。

注意：对USE[B]，B中p点引用变量A值且在B中的p点前没有A的定值点这两个条件必须同时满足；同样，对DEF[B]，p引用变量A是在B之外且A在B中被重新定值这两个条件也必须同时满足。

显然，USE和DEF可以从给定的程序流图中直接求出，问题是如何计算IN和OUT。对已经介绍过的到达-定值方程，那里的基本块B的IN集是由IN的所有前驱的OUT集计算出来的，这是一个由前往后的计算过程。对于我们现在需要计算的基本块活跃变量来说，根据定义它应该是一个由后往前的计算过程，即从某基本块所有后继的IN来求得该基本块的OUT。这是因为：对在某点p定值的变量A，只有在以后引用了它，才表示变量A从p开始是活跃的，所以只有从后面寻找引用了哪些变量并向前寻找这些变量相应的定值点，即此时才能确定与这些找到定值点开始所对应的变量才是活跃的。

因此，计算基本块的活跃变量这个过程是由后向前进行的。按照这一思路，我们可得到计算所有基本块B的IN[B]和OUT[B]的du链数据流方程：

$$IN[B] = OUT[B] - DEF[B] \cup USE[B]$$

$$OUT[B] = \bigcup_{s \in S[B]} IN[s]$$

其中，S[B] 代表B的所有后继基本块集。du链数据流方程也可用迭代法求解，假定已求出程序流图中各结点的深度为主次序，则按深度为主次序的逆序(由后向前)的迭代法求解du链数据流方程算法如下：

```
for( i=1; i<=n; i++ )
    IN[Bi] = Φ;                          //初始化
change=true;
while( change )
{   change=false;
    for( i=n; i>=1; i-- )
    {   OUT[Bi] = ⋃ IN[s]
                  s∈S[B]
        NEWIN= ( OUT[Bi] −DEF[Bi] ) ∪ USE[Bi];
        if( NEWIN!=IN[Bi] )
        {   change=true;
            IN[Bi] =NEWIN;
        }
    }
}
```

例5.12 假定只考虑变量j，试求例5.11中图5-22程序流图的du链数据方程的解，并求基本块B_3中j的定值点d_4以及基本块B_4中j的定值点d_5的du链。

[解答] 图5-22程序流图对应的DEF和USE见表5.3。

表5.3 程序流图对应的DEF和USE

基本块	DEF	USE
B_1	$\{(d_4, j), (d_5, j), (d_7, j)\}$	{ }
B_2	{ }	{ }
B_3	$\{(d_5, j), (d_7, j)\}$	$\{(d_4, j)\}$
B_4	$\{(d_4, j), (d_7, j)\}$	$\{(d_5, j)\}$
B_5	{ }	$\{(d_7, j)\}$

对B_2中的DEF，虽然有许多地方引用了j，但在B_2中j没有重新定值，所以这些引用不属于DEF，B_5也是同理；对B_3中的USE，由于j属于先引用后定值，故 (d_4, j) 属于USE。

根据迭代算法，按照程序流图中结点的深度为主次序的逆序，即B_5、B_4、B_3、B_2、B_1进行计算。

置初值：

$IN[B_1]=IN[B_2]=IN[B_3]=IN[B_4]=IN[B_5]=\{\}$

第一次迭代：

$OUT[B_5]=IN[B_2]=\{\}$

$IN[B_5]=(OUT[B_5]-DEF[B_5])\cup USE[B_5]=(\{\}-\{\})\cup\{(d_7,j)\}=\{(d_7,j)\}$

$OUT[B_4]=IN[B_5]=\{(d_7,j)\}$

$IN[B_4]=(OUT[B_4]-DEF[B_4])\cup USE[B_4]=(\{(d_7,j)\}-\{(d_4,j),(d_7,j)\})\cup\{(d_5,j)\}=\{(d_5,j)\}$

$OUT[B_3]=IN[B_4]\cup IN[B_5]=\{(d_5,j)\}\cup\{(d_7,j)\}=\{(d_5,j),(d_7,j)\}$

$IN[B_3]=(OUT[B_3]-DEF[B_3])\cup USE[B_3]$

$=(\{(d_5,j),(d_7,j)\}-\{(d_5,j),(d_7,j)\})\cup\{(d_4,j)\}=\{(d_4,j)\}$

$OUT[B_2]=IN[B_1]\cup IN[B_3]=\{\}\cup\{(d_4,j)\}=\{(d_4,j)\}$

$IN[B_2]=(OUT[B_2]-DEF[B_2])\cup USE[B_2]=(\{(d_4,j)\}-\{\})\cup\{\}=\{(d_4,j)\}$

$OUT[B_1]=IN[B_2]=\{(d_4,j)\}$

$IN[B_1]=(OUT[B_1]-DEF[B_1])\cup USE[B_1]=(\{(d_4,j)\}-\{(d_4,j),(d_5,j),(d_7,j)\})\cup\{\}=\{\}$

各次迭代结果如表 5.4 所示。

各次迭代结果如表5.4所示。

表 5.4 程序流图的 IN 和 OUT

基本块	第一次迭代		第二次迭代		第三次迭代	
	OUT	IN	OUT	IN	OUT	IN
B_5	{ }	$\{(d_7,j)\}$	$\{(d_4,j)\}$	$\{(d_4,j),(d_7,j)\}$	$\{(d_4,j)\}$	$\{(d_4,j),(d_7,j)\}$
B_4	$\{(d_7,j)\}$	$\{(d_5,j)\}$	$\{(d_4,j),(d_7,j)\}$	$\{(d_5,j)\}$	$\{(d_4,j),(d_7,j)\}$	$\{(d_5,j)\}$
B_3	$\{(d_5,j),(d_7,j)\}$	$\{(d_4,j)\}$	$\{(d_4,j),(d_5,j),(d_7,j)\}$	$\{(d_4,j)\}$	$\{(d_4,j),(d_5,j),(d_7,j)\}$	$\{(d_4,j)\}$
B_2	$\{(d_4,j)\}$	$\underline{\{(d_4,j)\}}$	$\{(d_4,j)\}$	$\underline{\{(d_4,j)\}}$	$\{(d_4,j)\}$	$\{(d_4,j)\}$
B_1	$\{(d_4,j)\}$	{ }	$\{(d_4,j)\}$	{ }	$\{(d_4,j)\}$	{ }

因为第三次迭代结果与第二次迭代结果相同，所以它就是所求的解。同样，由于第一次迭代和第二次迭代的循环入口处IN[B_2]相同，故无需再进行第三次迭代。

求程序流图B_3中j的定值点d_4以及B_4中j的定值点d_5的du链：如果B_3中d_4后有j的其它定值，则j的定值点d_4的du链只包含d_4后最靠近d_4的j的定值以前的所有引用j的点；如果B_3中d_4后没有j的其它定值但有j的引用，则j的定值点d_4的du链除OUT[B3] 外还应包含这些引用点。

因OUT[B_3] ={(d_4, j), (d_5, j), (d_7, j)}而B_3中d_4后面没有j的其它定值也没有j的引用，所以B_3中j的定值点d_4的du链就是OUT[B_3](只考虑变量j)；也即，d_4的j的定值到达程序中j的引用点d_4、d_5和d_7(d_4中j的定值到达d_4，这是由于d_5、d_7中没有其它对j的定值，而d_4又可不经过d_5对j重新定值而到达d_4)。

同样，因OUT[B_4] ={(d_4, j), (d_7, j)}，而B_4中d_5后面没有j的其它定值也没有j的引用，所以B_4中j的定值点d_5的du链就是OUT[B_4]；也即，d_5的j的定值到达程序中j的引用点d_4和d_7(d_5中j的定值只能到达d_4、d_7而不能到达d_5，这是因为在到达d_5前必须经过d_4对j的重新定值)。

5.3.3 复写传播

我们称形如A=D的赋值为复写，复写可直接出现在语法分析后的中间代码中。在5.1节局部优化中重写DAG为四元式时，我们曾经说过，如果A未在该基本块的后继中被引用，则可删除A=D；然而，如果A在基本块的后继中被引用能否也删除它呢？这是在此需要研究的问题。

容易看出，假设有某复写s: A=D；如果对程序中所有引用该A值的四元式p，我们能确定：

(1) 到达p的A的定值点只是s。例如在程序中只有s: A=D；…；p：X=C1*A+C2；也即，A在引用点p的ud链仅仅包含s。

(2) 从s到p的每一条通路包括多次穿过p的通路(但不穿过s多次)，没有对D重新定值。

那么，就可把s: A=D删除，并把p中引用A改为引用D，我们称它为复写传播。

为了确定符合上述条件的s和p，需要在程序流图中进行数据流分析。为此，我们定义：

C_IN[B]——满足下述条件的所有复写s: A=D的集：从首结点到基本块B入口之前的每一通路上都包含有复写s: A=D，并且在每一通路上最后出现的那个复写s: A=D到B入口之前未曾对A或D重写定值。

C_OUT[B]——满足下述条件的所有复写 s: A=D的集：从首结点到基本块B出口之后的每一通路上都包含有复写 s: A=D，并且在每一通路上最后出现的那个复写s: A = D到B出口之间未曾对A或D重新定值。

C_GEN[B]——基本块B中满足下述条件的所有复写s: A=D的集：在B中s的后面未曾对A或D重新定值。

C_KILL[B]——程序中满足下述条件的所有复写s: A=D的集：在s在基本块B外的条件下A或D在B外，且A或D在B中被重新定值。

这里，C_GEN[B] 和C_KILL[B] 均可从给定的程序流图中直接求出。为了求出IN[B] 和OUT[B]，我们列出数据流方程：

$$OUT[B] = IN[B] - C\_KILL[B] \cup C\_GEN[B]$$

$$IN[B] = \bigcap_{p \in P[B]} OUT[p] \qquad //B \text{ 不是首结点}$$

$$IN[B_1] = \Phi \qquad //B_1 \text{ 是首结点}$$

其中，P[B] 代表B的所有前驱基本块集。第二组方程中运算符是∩而不是∪ ，是因为一个复写在某基本块入口之前是可用的，仅当它在该基本块所有前驱出口之后是可用的。

上述数据流方程也可用迭代法求解，并在每次迭代过程中按深度为主次序依次计算各结点的OUT和IN。假设已求出流图的深度为主次序，则求解数据流方程的迭代算法如下：

```
IN[B1] = Φ;                          //首结点 B1 的 IN 和 OUT 的值始终不变
OUT[B1] =C_GEN[B1];
for( i=2; i<=n; i++ )
{    IN[Bi] =ξ;                      //置初值
     OUT[Bi] =ξ -C_KILL[Bi];
}
change=true;
while( change )
{    change=false;
     for( i=2; i<=n; i++ )
     {    NEWIN= ⋂ OUT[p];
               p∈P[B]
          if(IN[Bi] !=NEWIN)
          {    IN[Bi] =NEWIN;
               OUT[Bi] = ( IN[Bi] -C_KILL[Bi] ) ∪C_GEN[Bi];
               change=true;
          }
     }
}
```

这里，ξ代表程序中所有复写A=D的集。只要从上述数据流方程中求出各基本块的IN[B]就可以进行复写传播。复写传播算法如下：

(1) 应用du链信息求出复写s: A=D中A的定值所能到达的A的所有引用点。

(2) 对(1)中求出的A的各个引用点，假设其所属基本块分别为B_1、B_2、…、Br，如果对所有满足$1 \le i \le r$的i，都有$s \in C_IN[B_i]$且上述B_i中各个A的引用点之前都未曾对A或D重新定值，则转(3)，否则转(1)考虑下一复写句。

(3) 删除s，并把(1)中求出的那些引用A的地方改为引用D。

注意：对复写传播算法(2)，因为s属于IN[Bi]，即对A可到达的基本块Bi有：从首结点到Bi入口之前的每一通路上都包含有复写A=D且每一通路上最后出现的那个复写s：A=D到Bi入口之前未曾对A或D重新定值，再加上Bi中各个A的引用点之前都未曾对A或D重新定值；也即，从首结点出发到Bi中各个A的引用点之前必定有A=D，且这个A=D可到达Bi中各个A的引用点，而这些点所引用的A值即为D，所以可改为引用D，因此可将s: A=D删除。

例5.13 对图5-24的程序流图，

(1) 假设只考虑复写A=D和A=C，即把p限制为{A=B, A=C}，求数据流方程的解；

(2) 求解B_1中复写d_1: A=B和B_3中复写d_3: A=C的复写传播问题。

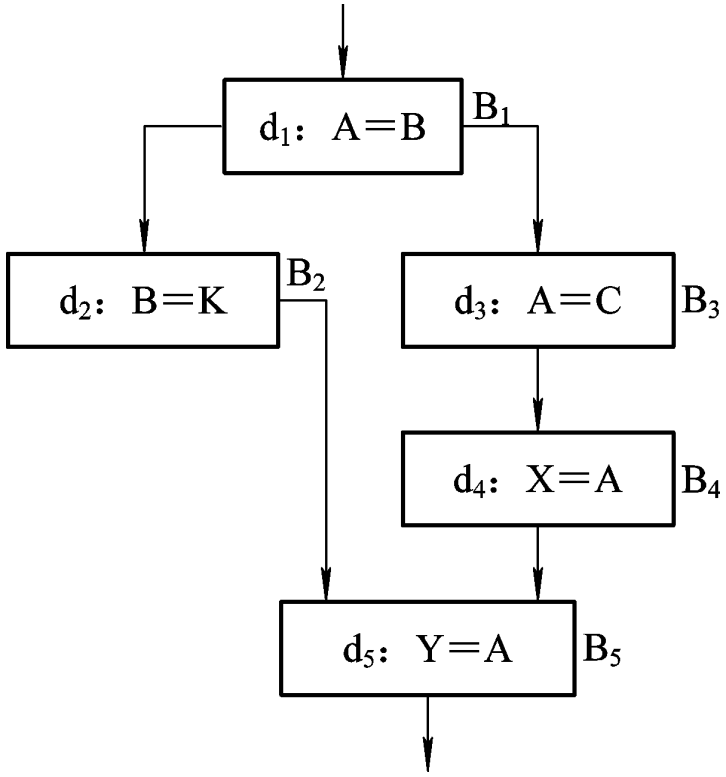


图5-24 程序流图

[解答] (1) 由程序流图先求出C_GEN和C_KILL，如表5.5所示。

表 5.5 程序流图的 C_GEN 和 C_KILL

基本块	C_GEN	C_KILL
B_1	{A=B}	{A=C}
B_2		{A=B}
B_3	{A=C}	{A=B}
B_4		
B_5		

注意：对B_2中的C_KILL，由于B被重新定值，找B_2之外所有形如B=…或…=B的语句，只有d_1满足，即为{A=B}；A被重新定值，由于所有复写仅考虑A=B和A=C，所以在此限制下C_KILL只有{A=C}。

再按程序流图的深度为主次序B1、B2、B3、B4、B5和迭代算法求解：

$IN[B_1] = \Phi$

$OUT[B_1] = \{A=B\}$

置初值：

$IN[B_2] = IN[B_3] = IN[B_4] = IN[B_5] = \{A=B, A=C\}$

$OUT[B_2] = \{A=B, A=C\} - \{A=B\} = \{A=C\}$

$OUT[B_3] = \{A=B, A=C\} - \{A=B\} = \{A=C\}$

$OUT[B_4] = \{A=B, A=C\} - \{\} = \{A=B, A=C\}$

$OUT[B_5] = \{A=B, A=C\} - \{\} = \{A=B, A=C\}$

第一次迭代：

$IN[B_2]=\{A=B\}$

$OUT[B_2]=(IN[B_2]-C\_KILL[B_2])\cup C\_GEN[B_2]=(\{A=B\}-\{A=B\})\cup\{\}=\{\}$

$IN[B_3]=\{A=B\}$

$OUT[B_3]=(IN[B_3]-C\_KILL[B_3])\cup C\_GEN[B_3]=(\{A=B\}-\{A=B\})\cup\{A=C\}=\{A=C\}$

$IN[B_4]=\{A=C\}$

$OUT[B_4]=(IN[B_4]-C\_KILL[B_4])\cup C\_GEN[B_4]=(\{A=C\}-\{\})\cup\{\}=\{A=C\}$

$IN[B_5]=OUT[B_5]\cap OUT[B_4]=\{\}\cap\{A=C\}=\{\}$

$OUT[B_5]=(IN[B_5]-C\_KILL[B_5])\cup C\_GEN[B_5]=(\{\}-\{\})\cup\{\}=\{\}$

第二次迭代与第一次迭代的结果相同，所以它就是所求方程的解。

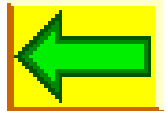
(2) 可以求出：A的定值点d_1的du链是A的引用点d5；A的定值点d_3的du链是A的引用点d_4和d_5。

现在考虑d_1：由du链信息知A的定值点d_1只到达A的引用点d_5，d_5属于基本块B_5而C_IN[B_5]=Φ，故d_1不可删除，d_5中的引用A不可改为引用B。

再考虑d_3：由du链信息知A的定值点d_3到达A的引用点d_4和d_5，因d_4属于基本块B_4，C_IN[B_4] ={d_3: A=C}，d_5属于基本块B_5，C_IN[B_5] =Φ，所以$d_3 \in$C_IN[B4] 而$d_3 \notin$ C_IN[B_5]，故d_3不可删除，d_4和d_5中的引用A不可改为引用C。

也即，以上复写都不可进行复写传播。

最后，需要指出的是，活跃变量与du链信息不仅可用于代码优化中的删除程序中无用赋值(包括无用的递归赋值)、代码外提，而且也可以在代码生成中用于寄存器分配。此外，活跃变量与du链信息还可用在软件测试中，查找程序错误、追踪程序变量的出错地点以及错误影响范围等。



5.4 代码优化示例

我们通过一个高级语言程序的例子来了解代码优化的全过程。下面是一个用C语言编写的快速排序子程序：

```
void quicksort( m, n )
{
   int i, j;
   int v, x;
   if( n<=m ) return;
//fragment begins here
   i=m-1;
   j=n;
   v=a[n];
   while( 1 )
   {
      do i=i+1; while( a[i] <v );
      do j=j-1; while( a[j] >v );
      if( i>=j ) break;
      x=a[i];
      a[i] =a[j];
      a[j] =x;
   }
   x=a[i];
   a[i] =a[n];
   a[n] =x;
//fragment ends here
   quicksort( m, j );
   quicksort( i+1, n );
}
```

通过第四章的中间代码生成方法可以产生这个程序的中间代码。图5-25给出了程序中两个注解行之间的语句翻译成中间代码序列后所对应的程序流图。对图5-25程序流图的代码优化叙述如下。

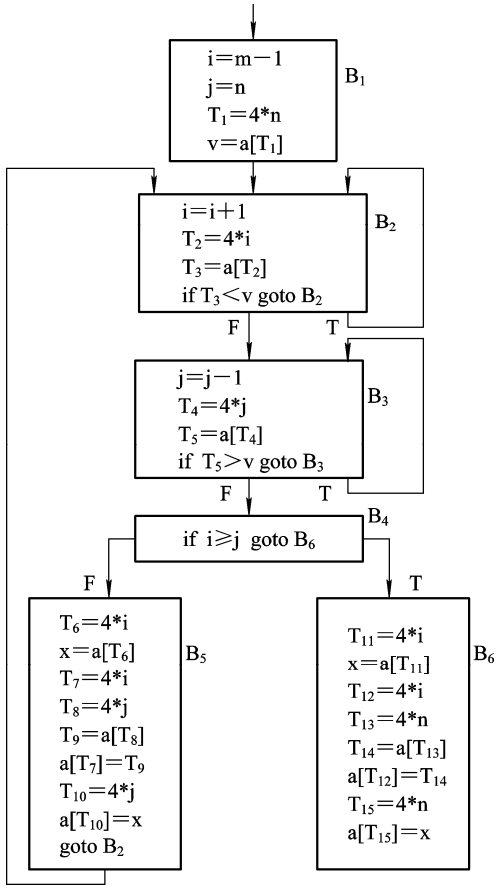

i=m−1
j=n
T1=4*n
v=a[T1]
B1
i=i+1
T2=4*i
T3=a[T2]
if T3<v goto B2
B2
F
T
j=j−1
T4=4*j
T5=a[T4]
if T5>v goto B3
B3
F
T
if i≥j goto B6
B4
F
T
T6=4*i
x=a[T6]
T7=4*i
T8=4*j
T9=a[T8]
a[T7]=T9
T10=4*j
a[T10]=x
goto B2
B5
T11=4*i
x=a[T11]
T12=4*i
T13=4*n
T14=a[T13]
a[T12]=T14
T15=4*n
a[T15]=x
B6

图5-25 程序流图

1．删除公共子表达式

在图5-25的B_5中分别把公共子表达式4*i和4*j的值赋给T_7和T_{10}，因此这种重复计算可以消除，即B_5中的代码变换成

B_5:　$T_6=4*i$

$x=a[T_6]$

$T_7=T_6$

$T_8=4*j$

$T_9=a[T_8]$

$a[T_7]=T_9$

$T_{10}=T_8$

$a[T_{10}]=x$

goto B_2

对B_5删除了公共子表达式后，仍然要计算4*i和4*j，我们还可以在更大范围内来考虑删除公共子表达式的问题，即利用B_3中的四元式T_4=4*j可以把B_5中的代码T_8=4*j替换为$T_8=T_4$。

同样，利用B_2中的赋值句T_2=4*i可以把B_5中的代码T_6=4*i替换为$T_6=T_2$。

对于B_6也可以同样考虑，最后，删除公共子表达式后的程序流图如图5-26所示。

2．复写传播

图5-26中的B5还可以进一步改进，四元式$T_6=T_2$把T_2赋给了T_6，而四元式$x=a[T_6]$ 中引用了T_6的值，但这中间并没有改变T_6的值，因此可以把$x=a[T_6]$ 变换为$x=a[T_2]$。这种变换称为复写传播。用复写传播的方法可以把B_5变为

$T_6=T_2$
$x=a[T_2]$
$T_7=T_2$
$T_8=T_4$
$T_9=a[T_4]$
$a[T_2]=T_9$
$T_{10}=T_4$
$a[T_4]=x$
goto B_2

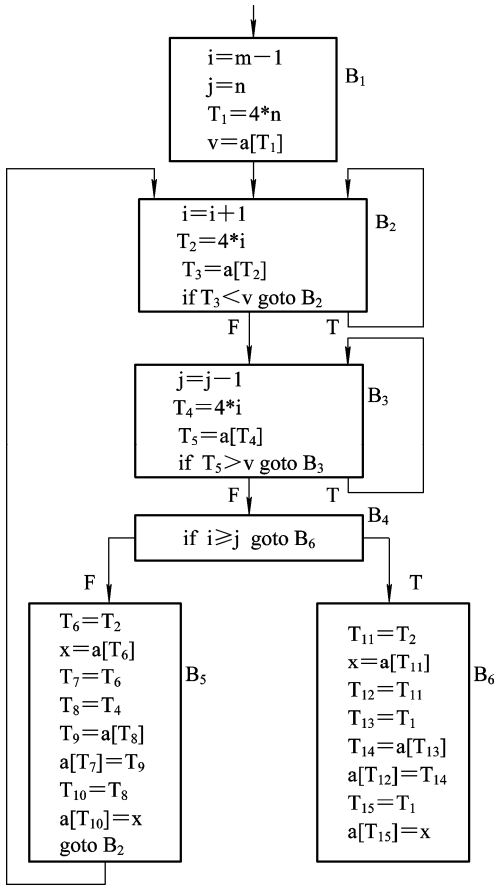


图5-26 删除公共子表达式后的程序流图

作进一步的考察可以发现，在B_2中计算了$T_3=a[T_2]$，因此在B_5中可以删除公共子表达式，即把$x=a[T_2]$ 替换为$x=T_3$，并继续通过复写传播，把B_5中的$a[T_4]=x$替换为$a[T_4]=T_3$。

同样，把B_5中的$T9=a[T_4]$ 替换为$T_9=T_5$，$a[T_2]=T_9$替换为$a[T_2]=T_5$。

这样，B_5就变为

$T_6=T_2$
$x=T_3$
$T_7=T_2$
$T_8=T_4$
$T_9=T_5$
$a[T_2]=T_5$
$T_{10}=T_4$
$a[T_4]=T_3$
goto B_2

复写传播的目的就是使对某些变量的赋值成为无用赋值。

$T_6=T_2$

$x=T_3$

$T_7=T_2$

$T_8=T_4$

$T_9=T_5$

$a[T_2]=T_5$

$T_{10}=T_4$

$a[T_4]=T_3$

goto B_2

复写传播的目的就是使对某些变量的赋值成为无用赋值。

3．删除无用赋值

对于进行了复写传播后的B_5，其中的变量x及临时变量T_6、T_7、T_8、T_9、T_{10}在整个程序中不再使用，故可以删除对这些变量赋值的四元式。删除无用赋值后B_5变为

$$a[T_2] = T_5$$

$$a[T_4] = T_3$$

$$\text{goto } B_2$$

对B6进行相同的复写传播和删除无用赋值后变为

$$a[T_2] = v$$

$$a[T1_] = T_3$$

复写传播和删除无用赋值后的程序流图如图5-27所示。

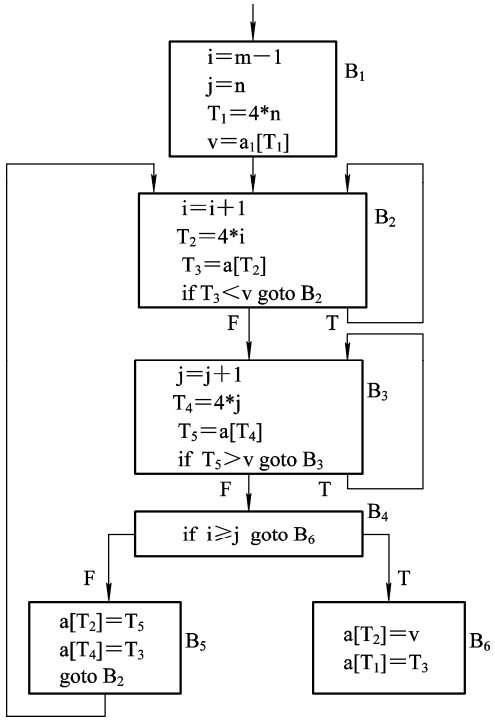


图5-27 复写传播和删除无用赋值后的程序流图

4．代码外提

考察图5-27，没有发现可外提到循环之外的不变运算。

5．强度削弱

观察图5-27中的内循环B_3，每循环一次，j的值减1；而T_4的值始终与j保持着$T_4=4*j$的线性关系，即每循环一次，T_4值随之减少4。因此，我们可以把循环中计算T_4值的乘法运算变为在循环前进行一次乘法运算而在循环中进行减法运算。同样，对循环B_2中的$T_2=4*i$也可以进行强度削弱。经过强度削弱后的程序流图如图5-28所示。

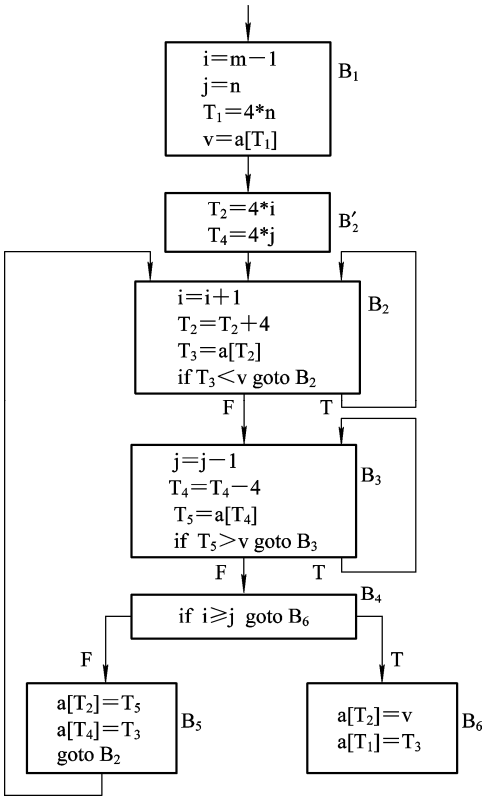


图5-28 强度削弱后的程序流图

6. 删除归纳变量

由图5-28可知，B_2中每循环一次，i值加1，T_2与i之间保持着$T_2=4*i$的线性关系；而B_3中每循环一次，j值减1，T_4与j之间保持着$T_4=4*j$的线性关系。在对$T_2=4*i$和$T_4=4*j$进行了强度削弱后，i和j仅出现在条件句if i≥j goto B_6中，其余地方不再被引用。因此，我们可以变换归纳变量而把此条件句变换为if $T_2 \geq T_4$ goto B_6。经过这种变换，我们又可以将无用赋值i=i+1和j=j–1删去。删除归纳变量后的程序流图如图5-29所示。

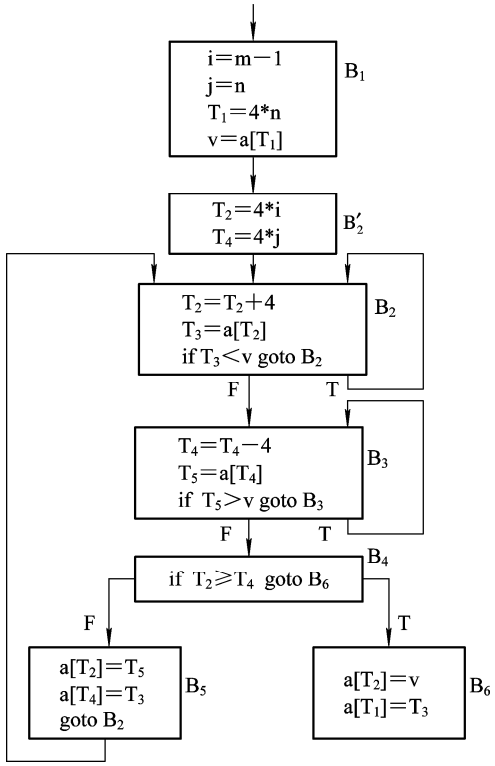
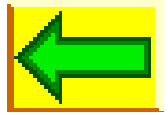


图5-29 删除归纳变量后的程序流图

通过上述各种优化，最终得到图5-29所示的优化结果。比较图5-25和图5-29可知：B_2和B_3中的四元式从4条减为2条，而且一条是由乘法变为加法；B_5中的四元式由9条变为3条，B_6中的四元式由8条变为2条。以上这些优化对循环执行来说，效果是非常明显的。虽然B_1的四元式由4条变为现在的B_1和B_2'共6条，但因其仅被执行一次，所以影响甚微。



习 题 5

5.1 完成以下选择题：

(1) 优化可生成　　的目标代码。

A．运行时间较短

B．占用存储空间较小

C．运行时间短但占用内存空间大

D．运行时间短且占用存储空间小

(2) 下列优化方法中不是针对循环的是　　。

A．强度削弱　　B．删除归纳变量

C．删除多余运算　　D．代码外提

(3) 对一个基本块来说，_____是正确的。

A．有一个入口语句和一个出口语句

B．有一个入口语句和多个出口语句

C．有多个入口语句和一个出口语句

D．有多个入口语句和多个出口语句

(4) _____不能作为一个基本块的入口。

A．程序的第一个语句

B．条件转移语句转移到的语句

C．无条件转移语句后的下一条语句

D．无条件转移语句转移到的语句

(5) 基本块内的优化为　　。

A．代码外提，删除归纳变量

B．删除多余运算，删除无用赋值

C．强度削弱，代码外提

D．循环展开，循环合并

(6) 在程序流图中，称具有　　的结点序列为一个循环。

A．非连通的且只有一个入口结点

B．强连通的但有多个入口结点

C．非连通的但有多个入口结点

D．强连通的且只有一个入口结点

(7) 关于必经结点的二元关系，下列叙述中不正确的是　　。

A．满足自反性　　B．满足传递性

C．满足反对称性　　D．满足对称性

(8) 已知有向边a→b是一条回边，则由它组成的循环是　　的所有结点组成的。

A．由结点a、结点b以及有通路到达b但该通路不经过a

B．由结点a、结点b以及有通路到达a但该通路不经过b

C．仅由结点a以及有通路到达a但该通路不经过b

D．仅由结点b以及有通路到达b但该通路不经过a

(9) 对循环中各基本块的每个四元式，如果它的　　，则将此四元式标记为“不变运算”。

A．每个运算对象为常数

B．每个运算对象定值点在循环内

C．每个运算对象定值点在循环外

D．每个运算对象为常数或者定值点在循环外

(10) 循环中进行代码外提时，对循环L中的不变运算S：A=B op C或A= op B或A=B，要求满足下述条件(A在离开L后仍是活跃的)中错误的是　　。

A．S所在的结点是L的所有出口结点的必经结点

B．A在L中其它地方未再定值

C．不变运算S中的B和C必须是常量

D．L中的所有A的引用点只有S中A的定值才能到达

5.2 何谓局部优化、循环优化和全局优化？优化工作在编译的哪个阶段进行？

5.3 将下面程序划分为基本块并作出其程序流图：

```
        read( A, B )
        F=1
        C=A*A
        D=B*B
        if C<D goto L1
        E=A*A
        F=F+1
        E=E+F
        write( E )
        halt
L1:     E=B*B
        F=F+2
        E=E+F
        write( E )
        if E>100 goto L2
        halt
L2:     F=F−1
        goto L1
```

5.4 基本块的DAG如图5-30所示。若：

(1) b在该基本块出口处不活跃；

(2) b在该基本块出口处活跃；

请分别给出下列代码经过优化之后的代码：

(1) a=b+c

(2) b=a−d

(3) c=b+c

(4) d=a−d

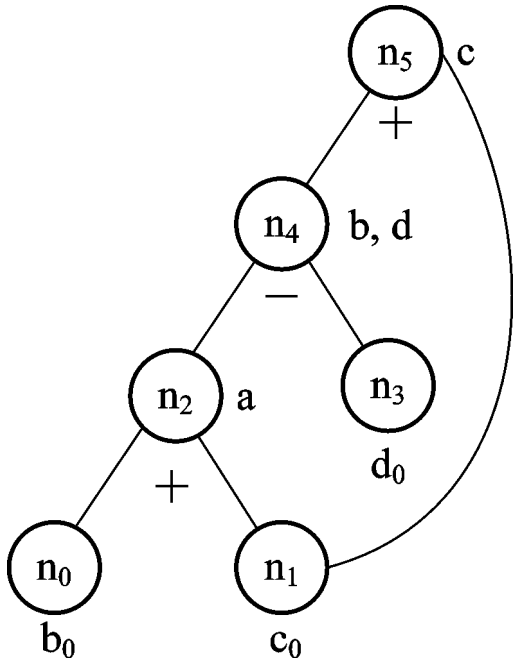


图5-30 习题5.4的DAG图

5.5 对于基本块P：

$S_0=2$

$S_1=3/S_0$

$S_2=T-C$

$S_3=T+C$

$R=S_0/S_3$

$H=R$

$S_4=3/S_1$

$S_5=T+C$

$S_6=S_4/S_5$

$H=S_6*S_2$

(1) 应用DAG对该基本块进行优化；

(2) 假定只有R、H在基本块出口是活跃的，试写出优化后的四元式序列。

5.6 对图5-31所示的流图，求出图中各结点i的必经结点集D(i) 以及流图中的回边与循环。

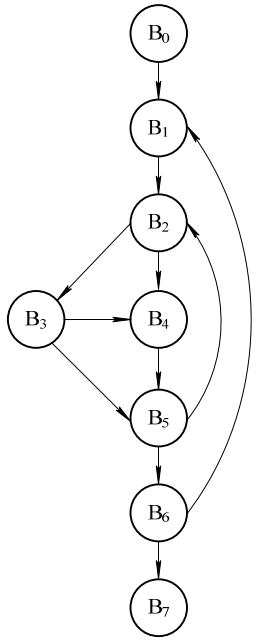


图5-31 习题5.6的程序流图

5.7 证明：如果已知有向边n→d是一回边，则由结点d、结点n以及有通路到达n而该通路不经过d的所有结点组成一个循环。

5.8 对下面四元式代码序列：

```
      A=0
      I=1
L1:   B=J+1
      C=B+I
      A=C+A
      if I=100 goto L2
      I=I+1
      goto L1
L2:   write A
      halt
```

(1) 画出其控制流程图；

(2) 求出循环并进行循环的代码外提和强度削弱优化。

5.9 某程序流图如图5-32所示。

(1) 给出该流图中的循环；

(2) 指出循环不变运算；

(3) 指出哪些循环不变运算可以外提。

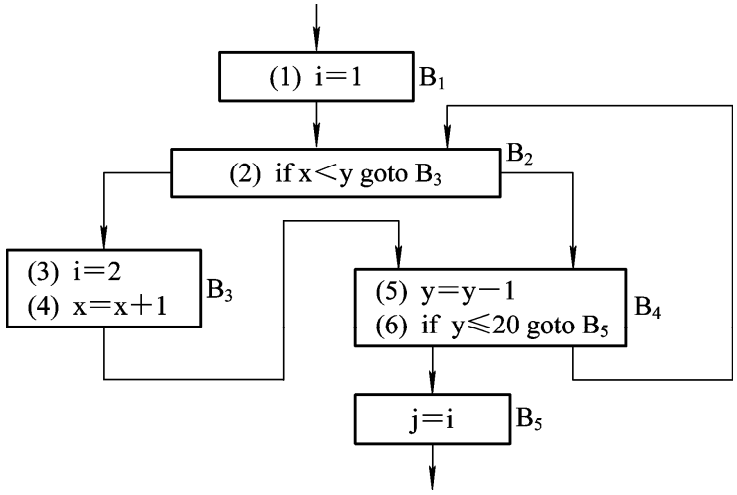


图5-32 习题5.9的程序流图

5.10 一程序流图如图5-33所示，试分别对其进行代码外提、强度削弱和删除归纳变量等优化。

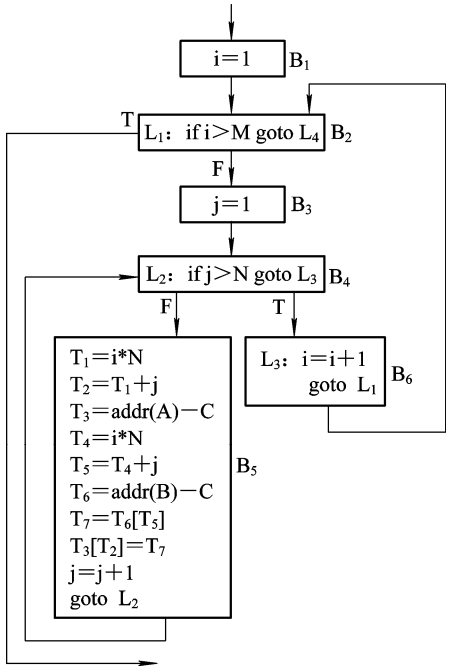


图5-33 习题5.10的程序流图