第十二讲　目标代码生成和代码优化基础

2021-12

编译过程最后阶段的工作是目标代码生成，其输入是某一种中间代码（如三地址码）以及符号表等信息，输出是特定目标机或虚拟机的目标代码。通常，在各级中间代码以及目标代码层次上，往往要通过各种等价变换对代码进行改进，我们把这种变换称为代码优化，优化的目标可以是程序性能（运行速度更快），也可以是其它方面，如代码体积（占用空间更少），程序功耗（使用能量更少），等等。

在现代编译器设计中，代码优化和目标代码生成是最复杂、最灵活和最值得研究的部分，内容相当丰富。限于本课的目标和范围，在这一章里，我们仅涉及代码优化和目标代码生成的一些基础内容，涵盖必要的知识点和重要概念。首先介绍程序控制流分析方面的基本知识，包含基本块、流图、以及循环等概念。然后是程序数据流分析方面的基本内容，包括几种重要的数据流信息及其求解算法。接着是关于代码优化的简介，包括个别优化算法以及对代码优化的概述。最后是目标代码生成的内容，介绍典型的代码生成过程。

如果不特别指明，这一章的中间代码形式均指三地址码（简称 TAC，参见第 9 章）。

1 基本块、流图和循环

1.1 基本块

基本块（Basic Block）是指程序中一个顺序执行的语句序列，其中只有一个入口语句和一个出口语句。执行时只能从其入口语句进入，从其出口语句退出。对于一个给定的程序，我们可以把它划分为一系列的基本块。因为在作优化时需要尽可能大地扩大优化范围，所以一般我们会默认在划分基本块时总是考虑尽可能大的基本块，即所谓极大基本块（若再添加一条语句就不满足基本块的条件了）。

从前面的定义可知，基本块的入口语句可以是下面任意三类语句：（1）程序的第 1 条语句；或者，（2）条件跳转语句或无条件跳转语句的跳转目标语句；或者，（3）条件跳转语句后面的相邻语句。

划分基本块的方法：

（1）先求出各个基本块的入口语句；

（2）对每一入口语句，构造其所属的基本块。它是由该语句到下一入口语句（不包括下一入口语句），或者到某个跳转语句（包括该跳转语句），或者到某个停语句（包括该停语句）之间的语句序列组成的。

凡未被纳入某一基本块的语句，都是程序中控制流程无法到达的语句，因而也是不会被执行到的语句，优化时可以把它们删除。

例如，图 1(a) 是一段三地址码程序，用于计算一个以 16 的阶乘为半径的圆的周长，然后输出结果。

利用前述划分基本块的方法来分析这段代码中有哪些基本块，首先确定入口语句，有(1)，(5)，(6)和(9)。语句(1)是程序的开始，语句(5)表示语句(8)跳转的目标语句，语句(6)是条件跳转指令之后的相邻语句，语句(9)是语句(5)跳转的目标语句。从入口指令开始，将代码分为四个基本块 *BB1*，*BB2*，*BB3*，和 *BB4*，如图 1(b)。基本块 *BB1* 由语句(1)到(4)组成。基本块 *BB2* 由第(5)条语句组成。基本块 *BB3* 是由语句(6)到(8)组成。基本块 *BB4* 即最后三条语句(9)～(11)。

确定基本块之后，采用<*BBi*, j>形式的局部编号（同时保留全局行号）来表示基本块 *BBi* 中的第 j 条语句，将图 1(b)重写为图 1(c)。

1.2 流图

把程序划分为基本块后，可以在基本块内实施一些局部优化。为了实施循环和全局优化等更大范围的优化工作，需要把程序（过程）作为一个整体来收集信息，需要分析基本块之间的控制流程关系、分析基本块内部以及基本块之间的变量赋值变化情况。

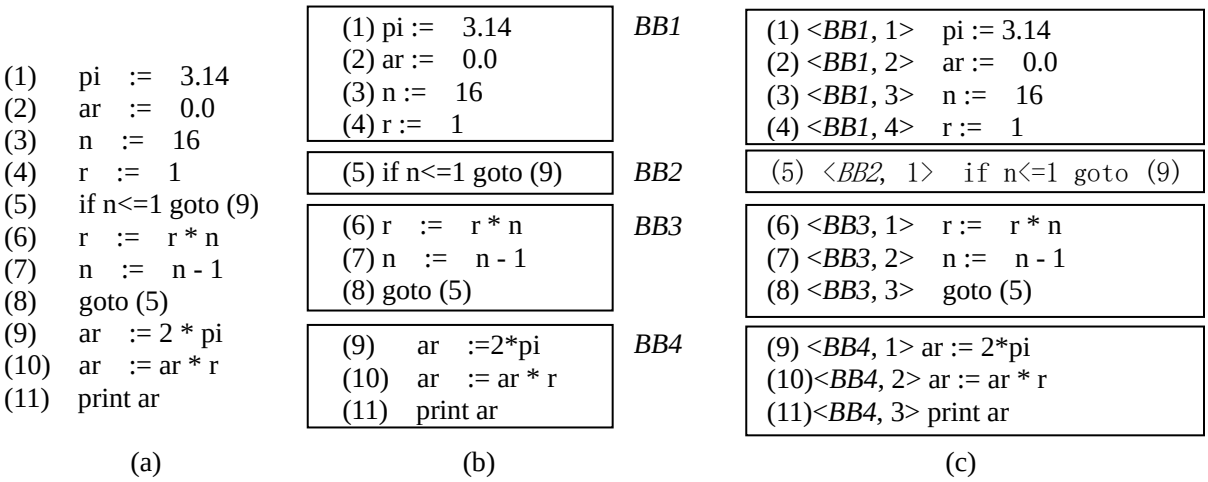

图 1 计算圆的周长的三地址中间代码 和基本块

可以为构成程序的基本块增加控制流程信息，方法是构造一个有向图，称之为**流图**或**控制流图**（*CFG*，*Control-Flow Graph*）。流图以基本块集为结点集的有向图；第一个结点为含有程序第一条语句的基本块，称为首结点；从基本块 *i* 到基本块 *j* 之间存在有向边，记作(*i*→*j*)，当且仅当

（1）基本块 *j* 是程序中基本块 *i* 之后的相邻基本块，并且基本块 *i* 的出口语句不是无条件跳转语句 goto 或停语句或返回语句; 或者，

（2）基本块 *i* 的出口语句是无条件跳转 goto *L* 或者条件跳转 if … goto *L*，并且 *L* 是基本块 *j* 的入口语句标号，即基本块 *i* 出口语句的跳转目标地址指向基本块 *j* 的入口语句。

根据基本块的划分以及流图的构造方法，一个流图的首结点是唯一的，并且从首结点出发可以到达流图中任何一个结点。

对于图 1 的程序和所划分的基本块，可以构造流图如图 2。其中结点基本块的集合为{*BB1*, *BB2*, *BB3*, *BB4*}，首结点基本块是 *BB1*，有向边的集合为{(*BB1*→*BB2*), (*BB2*→*BB3*), (*BB3*→*BB2*), (*BB2*→*BB4*)}。

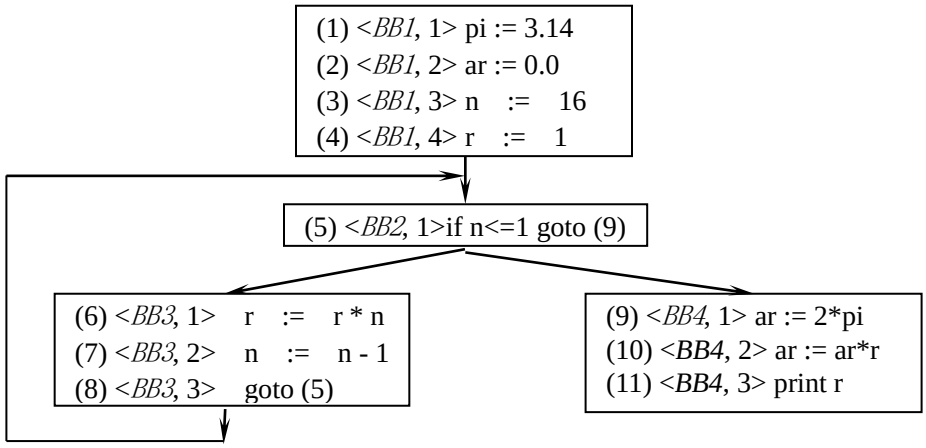

图 2 流图的例子

流图可以用来精确刻画一个程序的控制流程，即程序中所有基本块之间的执行顺序。流图中，某一个基本块运行之后可以到达的所有基本块是该基本块的**后继基本块**，可以直接运行并到达某一个基本块的所有基本块是该基本块的**前驱基本块**。图 2 中，*BB2* 的前驱基本块包括 *BB1* 和 *BB3*，而 *BB2* 的后继基本块为 *BB3* 和 *BB4*。

划分基本块、构造程序流图之后，就可以利用这些它们来捕获程序中的基本特征，以此为基础开展各种各样的优化以及服务于目标代码的生成。

1.3 循环

统计分析表明，对于大多数应用程序，绝大多数运行时间都花费在循环部分，因此循环优化对于整个程序的性能改进具有决定性意义。这里介绍如何利用流图来识别程序中的循环，这是开展循环优化的必要条件。其基本思路是根据流图计算所有结点的支配结点集，然后得到流图中的回边，根据回边就可以确定该流图中的循环。

在程序流图中，对任意两个结点 *m* 和 *n* 而言，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**支配结点**（*dominator*），记为 *m DOM n*。流图中结点 *n* 的所有支配结点的集合，称为结点 *n* 的支配结点集，记为 *D*(*n*)。设 *n*0 为流图中的首结点。根据这个定义，对流图中任意结点 *a*，一定有 *a DOM a* 以及 *n*0 *DOM a*，即任意结点是自身的支配结点，首结点是任意结点的支配结点。

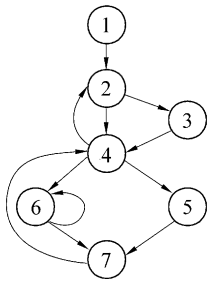

图 3 某程序的流图结构

图 3 中给出某个程序的流图，其结点即程序中的基本块，结点 *n* 的支配结点集 *D*(*n*)如下：

D(1)={1}
D(2)={1, 2}
D(3)={1, 2, 3}
D(4)={1, 2, 4}

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

假设 $n\rightarrow d$ 是流图中的一条有向边，如果 $d\ DOM\ n$ 则称 $n\rightarrow d$ 是流图中的一条**回边**（*back edge*）。作为例子，下面我们找出图 3 中流图的所有回边。

可以看出，该图的有向边集合为{(1→2), (2→3), (2→4), (3→4), (4→2), (4→5), (4→6), (5→7), (6→6), (6→7), (7→4)}。对照支配结点集 *D*(*n*)可知，图中的有向边(6→6)、(7→4)以及(4→2)是回边，因为有 6 *DOM* 6，4 *DOM* 7 以及 2 *DOM* 4。其他有向边都不是回边。

如果已知有向边 $n\rightarrow d$ 是回边，那么就可以求出包含该回边的自然循环（*natural loop*），简称**循环**。该循环就是由结点 *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，并且 *d* 是该循环的惟一入口结点。同时，因 *d* 是 *n* 的支配结点，所以 *d* 必可达该循环中任意结点。

对于图 3 流图中的例子，我们很容易看出。包含回边(6→6)的循环是{6}，其入口结点为 6；包含回边 7→4 的循环是{4，5，6，7}，其入口结点为 4；而包含回边 4→2 的循环是{2，3，4，5，6，7}，其入口结点为 2。

又如图 2 中流图的例子，有向边集合为{(*BB1*→*BB2*), (*BB2*→*BB3*), (*BB3*→*BB2*), (*BB2*→*BB4*)}，每个结点的支配结点集合为：

D (*BB1*)＝{*BB1*}
D (*BB2*)＝{*BB1*，*BB2*}
D (*BB3*)＝{*BB1*，*BB2*，*BB3*}
D (*BB4*)＝{*BB1*，*BB2*，*BB4*}

由此可以判定回边只有为(*BB3*→*BB2*)，相应的循环为{*BB2*，*BB3*}，其入口结点为 *BB2*。

简单总结一下，我们所讨论的循环结构是程序流图中具有下列性质的结点序列：

（1）它们是强连通的。也即，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，则必有一有向边从该结点引到其自身。

（2）它们中间有且只有一个是入口结点。对于入口结点来说，或者从序列外某结点有一有向边引到它，或者它本身就是程序流图的首结点。

因此，我们定义的循环就是流图中具有唯一入口结点的一个强连通子图，从循环外要进入循环，必须首先经过循环的入口结点。

找到了程序中的循环，就可以针对循环开展相关优化。

2 数据流分析基础

为做好代码生成和代码优化工作，通常需要收集整个程序流图的一些特定信息，并把这些信息分配到流图中的程序单元（如基本块、循环、或单条语句等）中。这些信息称为数据流信息，收集数据流信息的过程称为**数据流分析**（*data-flow analysis*）。

实现数据流分析的一种途径是建立和求解**数据流方程**（*data-flow equations*）。下面先介绍数据流方程的概念（2.1 节），然后通过以基本块为单位的两种重要重要数据流为例来介绍数据流方程求解的基本过程。这两种重要重要数据流分别是到达-定值数据流（2.2 节）和活跃变量数据流（2.3 节），前者是一种正向数据流信息，后者则是一种反向数据流信息。

除了到达-定值和活跃变量这两种数据流信息，还将介绍其它几种常用的数据流信息及其分析算法（2.4 节），包括 UD 链和 DU 链，以及基本块内变量的待用信息和活跃信息。

2.1 数据流方程的概念

数据流方程用于描述流入和流出某程序单元或程序中不同点之间的数据流信息之间的联系。例如，以下是一类典型的数据流方程：

$$out\,[S] = gen\,[S] \cup (in\,[S] - kill\,[S]) \qquad (11\text{-}1)$$

其含义为：离开程序单元 *S* 时的数据流信息（*out* [*S*]），或者是 *S* 内部产生的信息（*gen* [*S*]），或者是从 *S* 开始处进入（*in* [*S*]）但在穿过 *S* 的控制流时未被杀死（*killed*）、即不在 *kill* [*S*] 中的信息。

这里的 *S* 可以是任何我们所关注的程序单元，比如基本块、循环、或单条语句等。*S* 内部产生的数据流信息 *gen* [*S*]，以及 *S* 内部能够杀死的数据流信息 *kill* [*S*]，均依赖于依赖于所需要的信息，即根据数据流方程所要解决的问题来决定。

上述数据流方程中，数据流信息是沿着控制流前进，由 *in* [*S*]来定义 *out* [*S*]，这种数据流称为是**向前流**，相应的数据流方程（如 11-1）称为**正向数据流方程**。

对某些问题，数据流信息有可能不是沿着控制流前进，由 *in* [*S*]来定义 *out* [*S*]，而是反向前进，由 *out* [*S*]来定义 *in* [*S*]。这种数据流称为是**向后流**，相应的数据流方程（如 11-2）称为**反向数据流方程**。典型的反向数据流方程形如

$$in\,[S] = gen\,[S] \cup (out\,[S] - kill\,[S]) \qquad (11\text{-}2)$$

除了 11-1 和 11-2，另一类数据流方程是描述合流算符问题。所谓的合流算符，是指当多条边到达程序单元 *S* 时，由 *S* 前驱单元的 *out* 集合计算 *in* [*S*]时采用的运算是用交运算还是并运算，或者当多条边离开 *S* 时，由 *S* 后继单元的 *in* 集合计算 *out* [*S*]时采用的运算是用交运算还是并运算。通过合流运算计算 *in* [*S*]或 *out* [*S*]的数据流方程和 11-1 或 11-2 的联立和求解，就可以计算出所需求的数据流信息。

下面两小节分别介绍以基本块作为程序单元 *S* 的两种数据流方程及其求解。一个数据流方程是用于到达-定值数据流分析，是一种正向数据流方程。另一个数据流方程是用于活跃变量数据流分析，是一种反向数据流方程。到达-定值数据流和活跃变量数据流是两种十分常用的数据流信息。

2.2 到达-定值数据流分析

对变量 A 的**定值**（definition）是指一条（TAC）语句赋值或可能赋值给 A。最普通的定值是对 *A* 的赋值或读值到 *A* 的语句，该语句的位置称作 *A* 的**定值点**。

变量 A 的**定值点 d 到达某点 p**，是指如果有路径从紧跟 d 的点到达 p，并且在这条路径上 d 未被“杀死”，即该变量未被重新被定值。直观地说，是指流图中从 d 有一条路径到达 p 且该通路上没有 A 的其它定值。

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，定义下面几个集合：

in[B]：到达基本块 B 入口处（入口语句之前的位置）的各个变量的所有定值点集合。

out[B]：到达 B 出口处（紧接着出口语句之后的位置）的各个变量的所有定值点集合。

gen[B]：B 中定值的并能够到达 B 出口处的所有定值点集合，即 B 所“产生”的定值点集合。

kill[B]：基本块 B 外满足下述条件的定值点集：这些定值点能够到达 B 的入口处，但所定值的变量在 B 中已被重新定值；即 B 所“杀死”的定值点集合。[1]

---

[1] 注：在龙书中，*kill*[B]中未排除那些不能到达 B 入口处的定值点；*kill*[B]的这两种含义不影响 *in*[B]和 *out*[B]

分析这几个集合之间的关系，会发现它们符合

$$out[B] = gen[B] \cup (in[B] - kill[B]) \qquad (11\text{-}3)$$

这恰好是一个正向数据流方程，所描述的数据流信息我们称之为**到达-定值数据流**。

对于 *out*[B]，其计算方法为所有该基本块入口处的定值点集合 *in*[B]中去除当前基本块“杀死”的定值点，再加入当前基本新“产生”的定值点，因此有：

(1) 如果某定值点 d 在 *gen*[B]中，则 d 一定也在 *out*[B]中；

(2) 如果某定值点 d 在 *in*[B]中而且 d 定值的变量在 B 中没有重新定值，那么 d 在 *out*[B]中；

(3) 除(1)(2)两种情况外，没有其他的 d∈*out*[B]。

进一步，经过分析到达-定值数据流的合流问题，容易得出每个基本块 B 的 *in* [B]和 B 的所有前驱基本块的 *out* 集合之间的关系为

$$in[B] = \cup (out[b]) \qquad b \in P[B] \qquad (11\text{-}4)$$

其中，*P*[B]为 B 的所有前驱基本块的集合。

由 11-4，*in*[B]是 B 的所有前驱基本块的出口处 *out* 集合的并集。对于 *in*[B]，容易看出：某定值点 d 到达 B 的入口处，当且仅当它到达 B 的某一前驱基本块的出口处。

我们称 11-3 和 11-4 两个数据流方程的联立为**到达-定值数据流方程**。

gen[B]和 *kill*[B] 是每个基本块 B 的固有属性，均可直接从基本块本身给出。这样，通过到达-定值数据流方程，就可求解出所有的 *in*[B]和 *out*[B]。

考察图 4 的流图。为简洁，我们省略了各基本块出口处的跳转语句（如果有的话）。各 TAC 语句左边的 d 分别代表该语句的位置。假设只考虑变量 i 和 j，我们先计算出所有基本块 B 的 *gen*[B]和 *kill*[B]，如图 5 所示。

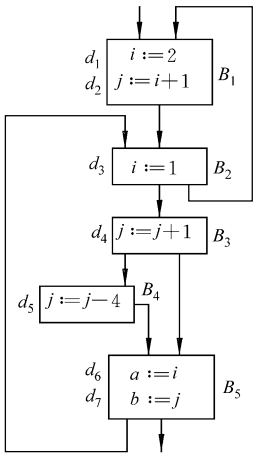

图 4 某个流图

基本块 B	gen(B)	kill(B)
B_1	$\{d_1, d_2\}$	$\{d_3, d_4, d_5\}$
B_2	$\{d_3\}$	$\{d_1\}$
B_3	$\{d_4\}$	$\{d_2, d_5\}$
B_4	$\{d_5\}$	$\{d_4\}$
B_5	Φ	Φ

图 5 计算 gen 和 kill

有了 *gen*[B]和 *kill*[B]，我们就可以通过数据流方程 11-3 和 11-4 来求解 *out*[B]和 *in*[B]了。

设流图中有 *n* 个结点，则数据流方程 11-3 和 11-4 是共有 2*n* 个变量的 *in*[B]和 *out*[B]的线性联立方程组。可以采用如下的迭代算法来给出这个方程组的一个最小不动点解（实际中需要的解）：

的计算结果。

```
(1)   for  i:= 1 to n {                                  //置初值
(2)        in[Bi] :=Φ;
(3)        out[Bi] := gen[Bi];
(4)   }
(5)   change := true;
(6)   while change  {
(7)        change := false;
(8)        for i := 1 to n  {
(9)             newin := ∪ out[b] (b∈P[Bi]) ;
(10)            if newin ≠ in[Bi]  {
(11)                 change := ture;
(12)                 in[Bi] := newin;
(13)                 out[Bi] := gen[Bi] ∪ (in[Bi] – kill[Bi])
(14)            }
(15)       }
(16)  }
```

上述算法中，首先设置每个 $in[B_i]$和 $out[B_i]$的迭代初值分别为Φ和 $gen[B_i]$。然后在第(8)行中，我们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 值不相等，则置 change 为 true，这表示尚需进行下一次迭代。

例如，对图 4 的流图，假设只考虑变量 *i* 和 *j*，应用以上算法，求联立数据流方程 11-3 和 11-4 的的解。

图 5 已列出各基本块的 *gen* 和 *kill*。根据上述算法，求解步骤如下：

开始，置迭代初值

$in[B_1] = in[B_2] = in[B_3] = in[B_4] = in[B_5] = \Phi$

$out[B_1] = \{d_1,d_2\}$

$out[B_2] = \{d_3\}$

$out[B_3] = \{d_4\}$

$out[B_4] = \{d_5\}$

$out[B_5] = \Phi$

执行算法第（5）行，置 change 为 true。第 1 次迭代开始，首先置 change 为 false。在算法第（8）行依次对 B_1，B_2，B_3，B_4 和 B_5 执行算法第（9）行至第（13）行。这样，一直迭代下去，直至 newin 值不发生变化为止。我们发现，第 4 次迭代的结果与第 3 次迭代完全相同。因此，第 3 次迭代后的 *in* 和 *out* 就是最后求出的结果。前 3 次迭代结果如图 6 所示。

基本块	第1次迭代		第2次迭代		第3次迭代	
	in(B)	out(B)	in(B)	out(B)	in(B)	out(B)
B_1	$\{d_3\}$	$\{d_1, d_2\}$	$\{d_2, d_3\}$	$\{d_1, d_2\}$	$\{d_2, d_3, d_4, d_5\}$	$\{d_1, d_2\}$
B_2	$\{d_1, d_2\}$	$\{d_2, d_3\}$	$\{d_1, d_2, d_3, d_4, d_5\}$	$\{d_2, d_3, d_4, d_5\}$	$\{d_1, d_2, d_3, d_4, d_5\}$	$\{d_2, d_3, d_4, d_5\}$
B_3	$\{d_2, d_3\}$	$\{d_3, d_4\}$	$\{d_2, d_3, d_4, d_5\}$	$\{d_3, d_4\}$	$\{d_2, d_3, d_4, d_5\}$	$\{d_3, d_4\}$
B_4	$\{d_3, d_4\}$	$\{d_3, d_5\}$	$\{d_3, d_4\}$	$\{d_3, d_5\}$	$\{d_3, d_4\}$	$\{d_3, d_5\}$
B_5	$\{d_3, d_4, d_5\}$	$\{d_3, d_4, d_5\}$	$\{d_3, d_4, d_5\}$	$\{d_3, d_4, d_5\}$	$\{d_3, d_4, d_5\}$	$\{d_3, d_4, d_5\}$

图 6 计算 in 和 out

有了到达-定值数据流信息，我们就可以方便地求出到达程序某一点 p 的各个变量的所有定值点。可以按下述规则求出到达基本块 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 A 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。

2.3 活跃变量数据流分析

一些数据流信息的获得依赖于从程序流图反方向进行计算，活跃变量信息就是其中的一种。这一小节，我们讨论以基本块为单位的活跃变量数据流分析。

对程序中的某变量 A 和某点 p 而言，如果存在一条从 p 开始的通路，其中引用了 A 在点 p 的值，则称 A 在点 p 是**活跃的**。否则称 A 在点 p 是死亡的。为了求出各基本块 B 入口和出口处的活跃变量信息，我们定义以下集合：

LiveIn[B]：B 入口处为活跃的变量的集合。

LiveOut[B]：B 的出口处的活跃变量的集合。

Def[B]：B 中定值的且定值前未曾在 B 中引用过的变量集合。

LiveUse[B]：B 中被定值之前要引用的变量集合。

这几个集合之间满足下列数据流方程

$$LiveIn[B] = LiveUse[B] \cup (LiveOut[B] - Def[B]) \qquad (11\text{-}5)$$

这个方程是通过 B 出口处的信息来计算 B 入口处的信息，是一个反向数据流方程，所描述的数据流信息我们称之为**活跃变量数据流**。可以看出，如果变量在 B 中定值前有引用，或者在 B 出口处活跃并且没有在 B 中被定值，那么它在 B 入口处就是活跃的。

此外，容易看出每个基本块 B 的 *LiveOut*[B]和 B 的所有后继基本块的 *LiveIn* 集合之间的关系为

$$LiveOut[B] = \cup (LiveIn[b]) \qquad b \in S[B] \qquad (11\text{-}6)$$

其中，*S*[B]为 B 的所有后继基本块的集合。

方程 11-6 指出，变量在 B 出口处活跃，当且仅当它在 B 的某个后继基本块入口处活跃。

我们称 11-5 和 11-6 两个数据流方程的联立为**活跃变量数据流方程**。

LiveUse[B]和 *Def*[B]是每个基本块 B 的固有属性，均可直接从基本块本身给出。这样，通过活跃变量数据流方程，就可求解出所有的 *LiveIn*[B]和 *LiveOut*[B]。

假设流图中有 *n* 个结点，则数据流方程 11-5 和 11-6 是共有 2*n* 个变量的线性联立方程组。可以采用如下的迭代算法来给出这个方程组的一个最小不动点解（实际中需要的解）：

```
(1)   for  i:= 1 to n {                          //置初值
(2)        LiveOut[Bi] :=Φ;
(3)        LiveIn[Bi] := LiveUse[Bi];
(4)   }
(5)   change := true;
(6)   while change  {
(7)        change := false;
(8)        for i := 1 to n  {
(9)              newout := ∪ LiveIn[b] (b∈S[Bi]) ;
(10)             if newout ≠ LiveOut[Bi]  {
(11)                   change := ture;
(12)                   LiveOut[Bi] := newout;
(13)                   LiveIn[Bi] := LiveUse[Bi] ∪ (LiveOut[Bi] – Def[Bi])
(14)             }
(15)       }
(16)  }
```

考察图 4 的流图。我们先计算出所有基本块 B 的 *LiveUse*[B]和 *Def*[B]。然后执行上述迭代算法，可求解出 *LiveIn*[B]和 *LiveOut*[B]。计算结果如图 7 所示

基本块 B	LiveUse(B)	Def(B)	LiveOut(B)	LiveIn(B)
B_1	Φ	$\{i, j\}$	$\{j\}$	Φ
B_2	Φ	$\{i\}$	$\{i, j\}$	$\{j\}$
B_3	$\{j\}$	Φ	$\{i, j\}$	$\{i, j\}$
B_4	$\{j\}$	Φ	$\{i, j\}$	$\{i, j\}$
B_5	$\{i, j\}$	Φ	$\{j\}$	$\{i, j\}$

图 7 计算活跃变量数据流信息

2.4 几种重要的变量使用数据流信息

利用基本块和流图可以方便地跟踪变量的使用信息。代码优化和目标代码生成的基本依据是变量的使用信息，只有确切获得基本块内部以及块间的变量使用情况之后，才能够进行适当的代码变换以及进行寄存器分配等工作。

这一小节介绍几种重要的变量使用数据流信息的获取。首先是有关刻画流图范围内变量的

引用点和定值点相关联信息的 UD 链和 DU 链，然后是关于基本块流图范围内变量的代用信息和活跃信息。

2.4.1 UD链

从 2.2 节可知，利用达-定值数据流信息可以方便地求出到达基本块 B 中某点 p 的任一变量 A 的所有定值点。这个过程可用于计算典型的数据流信息 —UD 链。

假设在程序中某点 u 引用了变量 A 的值，则把能到达 u 的 A 的所有定值点的全体，称为 A 在引用点 u 的**引用-定值链**（*Use-Definition Chaining*），简称 **UD 链**。类似于 2.2 节结尾处所述，可以在到达-定值数据流信息基础上计算 UD 链信息，其规则如下：

（1）如果在基本块 B 中，变量 A 的引用点 u 之前有 A 的定值点 d，并且 A 在 d 的定值可以到达 u，那么 A 在点 u 的 UD 链就是{d}。

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

采用上述规则，我们可以求出图 4 中变量 *i* 和 *j* 的 UD 链：

i 在引用点 d_2 的 UD 链为 $\{d_1\}$，

j 在引用点 d_4 的 UD 链为 $\{d_2, d_4, d_5\}$，

j 在引用点 d_5 的 UD 链为 $\{d_4\}$，

i 在引用点 d_6 的 UD 链为 $\{d_3\}$，

j 在引用点 d_7 的 UD 链为 $\{d_4, d_5\}$

2.4.2 DU链

和 UD 链对应的另一个典型的数据流信息是 DU 链。

假设在程序中某点 p 定义了变量 A 的值，从 p 存在一条到达 A 的某个引用点 s 的路径，且该路径上不存在 A 的其他定值点，则把所有此类引用点 s 的全体称为 A 在定值点 p 的**定值－引用链**（Definition-Use Chaining），简称 **DU 链**。

直观上理解，DU 链的计算可采用向后流的方法。一种可选的方案是首先对活跃变量信息进行扩展。

活跃变量数据流方程的解 *LiveOut*[B]所给出的信息是：离开基本块 B 时，哪些变量的值在 B 的后继中还会被引用。如果 *LiveOut*[B]不仅给出上述信息，而且还同时给出它们在 B 的后继中哪些点会被引用，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 u 的 DU 链。这时，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 *LiveOut*[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 到与 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 *LiveOut*[B]。

为此，我们需要把活跃变量数据流方程 11-5 和 11-6 中的 *LiveUse* 和 *Def* 代表的信息进行如下扩充：

（1）*LiveUse*[B]为(s,A)的集合，其中 s 是 B 中某点，s 引用变量 A 的值，且 B 中在 s 前面没有 A 的定值点。

（2）*Def*[B]为(s,A)的集合，其中 s 是不属于 B 的某点，s 引用变量 A 的值，但 A 在 B 中被重新定值。

扩充后的方程我们称之为 DU 链数据流方程。其求解算法完全类似于活跃变量数据流方程 11-5 和 11-6 的求解算法，只是其中 *Def* 和 *LiveUse* 指扩充后的 *Def* 和 *LiveUse*。

采用上述求解方法，我们可以求出图 4 中变量 *i* 和 *j* 的 DU 链：

i 在定值点 d_1 的 DU 链为 $\{d_2\}$，

j 在定值点 d_2 的 DU 链为 $\{d_4\}$，

i 在定值点 d_3 的 DU 链为 $\{d_6\}$，

j 在定值点 d_4 的 DU 链为 $\{d_4, d_5, d_7\}$，

j 在定值点 d_5 的 DU 链为 $\{d_4, d_7\}$

2.4.3 基本块内变量的待用信息

跟踪变量的值在单个基本块内部变化的目标之一是找到修改或使用该变量值的位置，分别对应为该变量的定值点和引用点。本小节介绍的**待用信息**（next use）是用来跟踪变量在基本块内紧接着一次使用该变量的情况。

假定一个基本块 *BB* 如下：

<*BB*, 1>　t := a - b
<*BB*, 2>　u := u - c
<*BB*, 3>　v := t + u
<*BB*, 4>　c := v + u

下面来分析这个基本块 *BB* 中各变量的待用信息情况。

对于当前基本块的第 i 条语句的某个变量 v，如果在当前基本块 i 语句之后的第 j 条语句中被引用，而第 j 条语句之前（第 i 条语句之后）未被引用，则 i 语句的变量 v 的待用信息记为 j；如果当前基本块内 i 语句之后不再引用该变量，其待用信息记为 0。我们的目标是把这些信息标注于各个变量的右上角，并跟踪基本块内各变量待用信息的变化情况。

用 nextuse(x)来表示处理过程中变量 x 当前的待用信息，初始时，令

nextuse(a)= nextuse(b)= nextuse(c) = nextuse(t)= nextuse(u)= nextuse(v)=0

随后，我们从后向前逐条语句地考查基本块中所有变量，从基本块出口到入口对每个 TAC 语句 i: A:=B op C 依次执行下述步骤：

（1）把变量 A 的 nextuse 信息附加到 TAC 语句 i 上；

（2）置 nextuse(A)为 0（由于在 i 中对 A 的定值只能在 i 以后的 TAC 语句才能引用，因而对 i 以前的 TAC 语句来说 A 不可能是待用的）；

（3）把变量 B 和 C 的 nextuse 信息附加到 TAC 语句 i 上；

（4）置 nextuse(B)=nextuse(C)=i 。

注：以上（1），（2），（3），（4）的次序不能颠倒。

对于上述基本块 *BB*，首先将当前 c 的 nextuse 信息附加到语句<*BB*, 4>上：

<*BB*, 4>　c^0 := v + u

重置 nextuse(c)为 0，并将 v 和 u 的最新 nextuse 信息附加到语句<BB, 4>上：

<BB, 4>　$c^0 := v^0 + u^0$

重置 nextuse(v)=nextuse(u)=4 。当前，各变量的 nextuse 信息变为：

nextuse(a)= nextuse(b)= nextuse(c)= nextuse(t)=0，nextuse(u)= nextuse(v)=4

重复上述过程，将当前 v 的 nextuse 信息附加到语句<BB, 3>上：

<*BB*, 3>　$v^4 := t + u$
<*BB*, 4>　$c^0 := v^0 + u^0$

重置 nextuse(v)为 0，并将 t 和 u 的最新 nextuse 信息附加到语句<BB, 3>上：

<*BB*, 3>　$v^4 := t^0 + u^4$
<*BB*, 4>　$c^0 := v^0 + u^0$

重置 nextuse(t)=nextuse(u)=3 。当前，各变量的 nextuse 信息变为：

nextuse(a)=nextuse(b)=nextuse(c)=nextuse(v)=0，nextuse(t)=nextuse(u)=3

再重复上述过程，将当前 u 的 nextuse 信息附加到语句<BB, 3>上：

<BB, 2>　$u^3 := u - c$
<*BB*, 3>　$v^4 := t^0 + u^4$
<*BB*, 4>　$c^0 := v^0 + u^0$

重置 nextuse(u)为 0，并将 u 和 c 的最新 nextuse 信息附加到语句<BB, 2>上：

<BB, 2>　$u^3 := u^0 - c^0$
<*BB*, 3>　$v^4 := t^0 + u^4$
<*BB*, 4>　$c^0 := v^0 + u^0$

重置 nextuse(u)=nextuse(c)=2 。当前，各变量的 nextuse 信息变为：

nextuse(a)=nextuse(b)=nextuse(v)=0，nextuse(t)=3，nextuse(c)= nextuse(u)=2

最后一次重复上述过程，将当前 t 的 nextuse 信息附加到语句<BB, 1>上：

<*BB*, 1>　$t^3 := a - b$
<BB, 2>　$u^3 := u^0 - c^0$
<*BB*, 3>　$v^4 := t^0 + u^4$
<*BB*, 4>　$c^0 := v^0 + u^0$

重置 nextuse(t)为 0，并将 a 和 b 的最新 nextuse 信息附加到语句<BB, 1>上：

<*BB*, 1>　$t^3 := a^0 - b^0$
<BB, 2>　$u^3 := u^0 - c^0$
<*BB*, 3>　$v^4 := t^0 + u^4$
<*BB*, 4>　$c^0 := v^0 + u^0$

重置 nextuse(a)=nextuse(b)=1 。当前，各变量的 nextuse 信息变为：

nextuse(t)=nextuse(v)=0，nextuse(c)= nextuse(u)=2，nextuse(a)=nextuse(b)=1

算法结束。这样，我们便成功跟踪了基本块内各变量的待用信息情况。

2.4.4 基本块内变量的活跃信息

一个基本块中，如果某个变量 A 在语句 i 中被定值，在 i 之后的语句 j 中要引用 A 值，且从 i 到 j 之间没有其他对 A 的定值点，则在语句 i 到 j 之间 A 是活跃的。

为了跟踪在一个基本块内每个变量的活跃信息，同样可以从基本块出口语句开始由后向前扫描，为每个变量名建立相应的活跃信息链。考虑到处理的方便，假定对基本块中可能在其他基本块中使用的变量在出口处都是活跃的，而对基本块内的临时变量不允许在基本块外引用，因此这些临时变量在基本块出口处都认为是不活跃的。

考虑与 2.4.3 节相同的基本块 BB：

<*BB*, 1>　t := a - b
<*BB*, 2>　u := u - c
<*BB*, 3>　v := t + u
<*BB*, 4>　c := v + u

其中，a，b 和 c 是在出口处是活跃的变量，而 t，u 和 v 是临时变量，在出口处不活跃。

下面来分析和标记这个基本块 *BB* 中出现的所有变量的活跃信息链。活跃变量标记为 L，非活跃变量标记为 F。同样，我们的目标是把这些信息标注于各个变量的右上角，并跟踪基本块内各变量活跃信息的变化情况。

用 live(x)来表示处理过程中变量 x 当前的活跃信息，初始时，令

live(a)=live(b)=live(c)=L，live(t)=live(u)=live(v)=F

随后，我们从后向前逐条语句地考查基本块中所有变量，从基本块出口到入口对每个 TAC 语句 i: A:=B op C 依次执行下述步骤：

（1）把变量 A 的 live 信息附加到 TAC 语句 i 上；

（2）置 live(A)为 F（由于在 i 中对 A 的定值只能在 i 以后的 TAC 语句才能引用，因而对 i 以前的 TAC 语句来说 A 是不活跃的）；

（3）把变量 B 和 C 的 live 信息附加到 TAC 语句 i 上；

（4）置 live(B)=live(C)=L。

注：以上（1），（2），（3），（4）的次序不能颠倒。

对于上述基本块 *BB*，首先将当前 c 的 live 信息附加到语句<*BB*, 4>上：

<*BB*, 4>　c^{L} := v + u

重置 live(c)为 F，并将 v 和 u 的最新 live 信息附加到语句<BB, 4>上：

<BB, 4>　c^{L} := v^{F} + u^{F}

重置 live(v)=live(u)=L 。当前，各变量的 live 信息变为：

live(a)=live(b)=live(v)=live(u)=L，live(c)=live(t)=F

重复上述过程，将当前 v 的 live 信息附加到语句<BB, 3>上：

<*BB*, 3>　v^{L} := t + u
<*BB*, 4>　c^{L} := v^{F} + u^{F}

重置 live(v)为 F，并将 t 和 u 的最新 live 信息附加到语句<BB, 3>上：

<*BB*, 3>　v^{L} := t^{F} + u^{L}
<*BB*, 4>　c^{L} := v^{F} + u^{F}

重置 live(t)=live(u)=L 。当前，各变量的 live 信息变为：

live(a)=live(b)=live(t)=live(u)=L，live(c)=live(v)=F

再重复上述过程，将当前 u 的 live 信息附加到语句<BB, 3>上：

<BB, 2>　u^{L} := u – c
<*BB*, 3>　v^{L} := t^{F} + u^{L}
<*BB*, 4>　c^{L} := v^{F} + u^{F}

重置 live (u)为 F，并将 u 和 c 的最新 live 信息附加到语句<BB, 2>上：

<BB, 2> $u^L := u^F – c^F$
<*BB*, 3> $v^L := t^F + u^L$
<*BB*, 4> $c^L := v^F + u^F$

重置 live(u)=live(c)=L 。当前，各变量的 live 信息变为：

live(a)=live(b)=live(c)=live(t)=live(u)=L，live(v)=F

最后一次重复上述过程，将当前 t 的 live 信息附加到语句<BB, 1>上：

<*BB*, 1> $t^L := a - b$
<BB, 2> $u^L := u^F – c^F$
<*BB*, 3> $v^L := t^F + u^L$
<*BB*, 4> $c^L := v^F + u^F$

重置 live (t)为 F，并将 a 和 b 的最新 live 信息附加到语句<BB, 1>上：

<*BB*, 1> $t^L := a^L - b^L$
<BB, 2> $u^L := u^F – c^F$
<*BB*, 3> $v^L := t^F + u^L$
<*BB*, 4> $c^L := v^F + u^F$

重置 live(a)=live(b)=L 。当前，各变量的 live 信息变为：

live(a)=live(b)=live(c)=live(u)=L，live(t)=live(v)=F

算法结束。这样，我们便成功跟踪了基本块内各变量的活跃信息情况。

3 代码优化技术

代码优化工作可以在编译的各个阶段进行。本质上讲，保证程序的含义保持一致的情况下对代码的任何修改都是允许的。也就是说，代码优化不应改变程序运行的结果，必须要保证优化后的代码与原来的代码完成相同的工作。

“没有最优，只有更优”，这句话极为准确地刻画出编译器优化的特点，可以从理论上证明，不论针对那一个优化目标，都无法找到一个能够生成最优代码的编译器，因此，人们总是在不断地研究和开发性能更好的编译器。下面给出两张图，分别展示两个重要开源编译器 gcc（http://gcc.gnu.org）和 open64（http://www.open64.net/）的程序性能和代码体积比较结果，测试用的 gcc 版本为 4.2.4，open64 版本为 4.2。

硬件环境是联想 X200 笔记本电脑，配置为 Intel® Core®2 Duo CPU P8400 处理器，主频 2.26GHz，操作系统为 Ubuntu Linux 8.10。测试软件用于计算快速傅立叶变换，对比内容为两个方面：一是相同的应用程序源代码，分别采用 gcc 和 open64 在不同的编译优化级别下进行编译，比较可执行程序的代码体积；二是给定一组相同的输入数据，分别运行前面得到的可执行程序，确保计算结果正确的情况下对比运行时间。

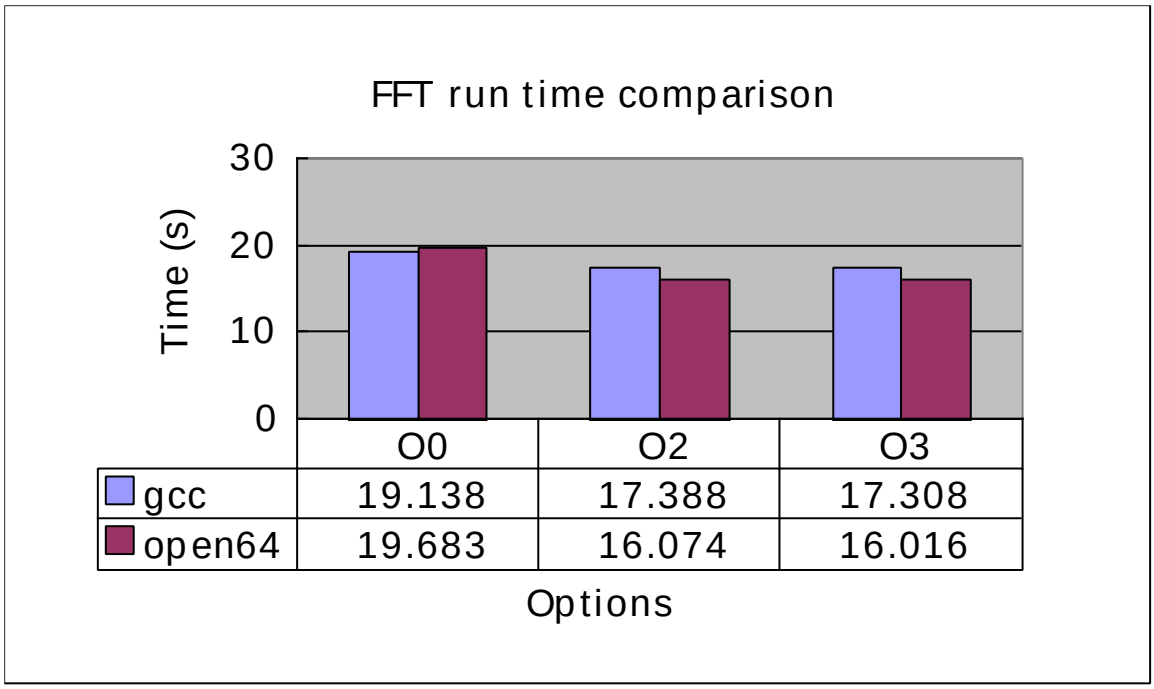

图 10 优化实例：快速傅立叶变换性能优化比较

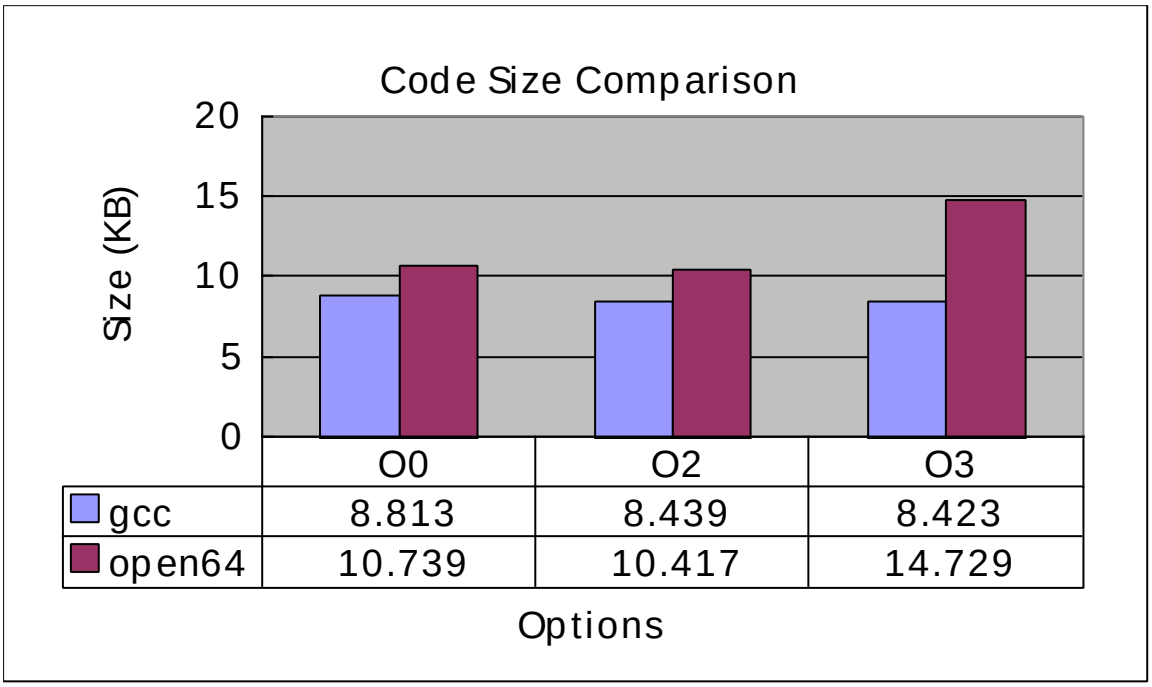

图 11 优化实例：快速傅立叶变换代码体积比较

对于 gcc 和 open64 这样的通用编译器，其优化选项会划分 O0、O1、O2、O3 等不同的级别，其中数字从 0 到 3 表示优化逐步加强，比如 O0 表示不优化，O3 则进行更多、更激进的优化工作。通常来讲，在相同的硬件和操作系统环境下，不同的编译优化选项其结果是不同的，采用 O3 选项编译的结果运行要更快一些，而代码体积也有可能会更大一些。

图 10 给出计算性能比较，纵轴为时间，单位是秒，运行时间越短越好。图中可以看出随着

优化级别的不断升高，程序的性能在不断提高，运行时间不断减少。

图 11 给出可执行代码体积比较，纵轴为可执行程序在硬盘中所占用的存储空间，单位是Kbytes，体积越小越好。图中可以看到，不同的优化选项生成的可执行程序体积差别很大。

图 10 和图 11 至少可以给我们这样一些启示：1）编译优化将带来性能改进，不同的编译器的改进情况不同，本例中 O3 优化选项的性能改进分别达到 9.5% (gcc)和 18.6% (open64)，事实上，同样的编译器和编译选项，对于不同应用程序的性能改进也可能不同；2）不同优化选项生成的可执行程序代码体积差别很大；3）不同的编译器优化策略和优化方法不同，本例中，gcc 编译器优化性能改进的同时代码体积在减小，而 open64 编译器性能改进的同时代码体积在不断增加，O3 和 O1 结果相比，代码体积增加接近 50%；4）对于编译器使用者来讲，要根据实际应用需要和优化目标来选取合适的编译器和编译选项；5）对于编译器开发人员来讲，则要根据编译器的使用场合和优化需求来设计合适的优化功能。

编译器的优化不管哪个阶段，为了尽可能达到全局最优，都需要对尽可能大的程序单元实施优化。然而，现实情况是我们很难将一个软件作为整体来实施优化。依据所涉及的程序范围，优化可以分为窥孔优化、局部优化、超局部优化、循环优化、过程内全局优化和过程间优化等不同的级别。本课只介绍其中部分常见的窥孔优化、局部优化、循环优化和全局优化方法的基本原理。

3.1 窥孔优化

窥孔优化是指在语句/指令序列上滑动一个包含几条语句/指令的窗口（称为窥孔），发现其中不够优化的语句/指令序列，用一段更有效的序列来替代它，使整个代码得到改进。以下我们举例说明几种常见的窥孔优化。

删除冗余的“取”和“存”（*redundant loads and stores*） 当我们看到下列 MIPS 指令序列

```
(1)  lw $t2, 5($t3)          /*  取地址$t3+5 中的字到寄存器$t2 */
(2)  sw $t2, 5($t3)          /*  将寄存器$t2 的字写入地址为$t3+5 的内存单元*/
```

可优化为

```
(1)  lw $t2, 5($t3)          /*  取地址$t3+5 中的字到寄存器$t2 */
```

需要注意的是，安全实施这个变换的前提条件是这两条语句必须在一个基本块内。因为，如果语句（2）前有标号，则不能保证（1）总是在（2）前执行，就不能把（2）优化掉。

常量合并（*constant folding*） 当我们看到下列 TAC 语句

```
(1)  r2:=3*2
```

可优化为

```
(1)  r2:=6
```

常量传播（*constants propagating*）当看到下列 TAC 语句序列

```
(1)  r2:=4
(2)  r3:=r1+r2
```

我们可将其优化为

```
(1)  r2:=4
(2)  r3:=r1+4
```

这里值得注意的是，虽然优化后语句的条数未减少，但若是知道 r2 不再活跃时，可进一步删除（1）。

代数化简（*algebraic simplification*）当看到下列语句序列

(1) x:=x+0
(2) ……
……
(n) y:=y*1

我们可将其中的的（1）和（n）在窥孔优化时可以被直接删掉。

控制流优化（*flow-of-control optimization*）当看到下列跳转语句序列

goto L1
……
L1:
goto L2

我们可将其替换为

goto L2
……
L1:
goto L2

需要注意的是，这个语句序列不属于一个基本块。窥孔优化的窗口有时会超越基本块。

死代码删除（*dead-code elimination*）有时我们可以利用窥孔优化删除逻辑上的死代码。比如，当看到如下代码序列

debug := false
if (debug) print …
……

我们可将其替换为

debug := false
……

强度削弱（*reduction in strength*）有时，我们可以适当改变运算强度来改进代码执行效率。例如，当看到下列 TAC 语句序列

x:=2.0*f

我们可将其替换为

x:=f+f

当看到下列 TAC 语句序列

x:=f/ 2.0

我们可将其替换为

x:=f*0.5

使用目标机惯用指令（*use of idioms*）有时，还可以针对目标机的特点用惯用指令来代替代价较高的指令，例如： 某个操作数与 1 相加，通常用“加 1”指令，而不是用“加”指令； 某个定点数乘以 2 ，可以采用“左移”指令；而除以 2 ， 则可以采用“右移”指令等等；再比如说对于多媒体处理，可以使用并行加或者并行比较等指令。

以上列举窥孔优化的典型类别，每个类别也仅给出了少量例子。例子虽然是以特定层次的代码表示形式给出的，但其思路并不限于那个层次的代码。许多窥孔优化策略同时适用于多种代码层次，如 AST 层，TAC 层，目标代码层，等等。

3.2 局部优化

局部优化指的是在一个基本块范围内进行的优化。常见的局部优化如：常量传播，常量合并，删除公共子表达式，复写传播，删除无用赋值，代数化简，等等。

基本块内的许多优化也可以看作是将基本块作为窗口的窥孔优化，但所采用的优化算法可以比传统的窥孔优化（仅限于扫描当前语句的前后）更复杂或适用性更强。这一小节中，我们通过例子介绍一种借助于构造基本块**有向无圈图**（简称 DAG 图，*Directed Acyclic Graph*）进行局部优化的方法。例子中的基本块由 *TAC* 语句组成。

为简化描述，我们仅考虑以下三种形式的 *TAC* 语句：原子表达式赋值语句 *A* := *B*；一元运算表达式赋值语句 *A* := op *B*；二元运算表达式赋值语句 *A* := *B* op *C*。这里，A 是变量，B 和 C 可以是变量或者常量。基本块的 DAG 图中，每个结点都带有标记（运算符，变量名字或常量），有向边由基本块内的 *TAC* 语句确定。图 12 表示三类 TAC 语句对应的 DAG 子图，分别有一个结点，两个结点和三个结点，有向边的方向通过图中不同的位置体现，高处结点的计算依赖于低处的结点。在每个这样的子图中，我们称高处的结点为父结点，低处的结点为子结点，子结点之间互称为兄弟结点，两个兄弟结点的位置处于同一高度。对应于一元运算和二元运算对应的子图，我们也将运算符标记在父结点上。基本块的 DAG 图是由这三类子图组成的。因为只有从高到低的依赖关系，所以 DAG 图是一种有向无圈图，即图中任一通路都不是环路。

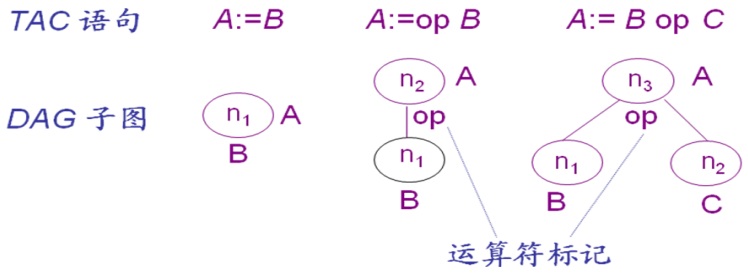

图 12 三类 TAC 语句对应的 DAG 子图

DAG 图中，不依赖于任何结点的结点为叶结点，其他结点为内部结点。在我们所定义的 DAG 图中，叶结点代表名字的初值，以唯一的变量名字或常数来标记，为避免混乱，用 x_0 表示变量名字 *x* 的初值。DAG 图的内部结点，都标记有相应的运算符。所有结点都可有一个附加的变量名字表。对于只含上述三类 TAC 语句的基本块来说，其 DAG 图的内部结点至少会附加一个变量名字。下面描述此类基本块的 *DAG* 图构造算法。

设 *x:=y op z, x:=op y, x:=y* 分别为第 1、2、3 种 *TAC* 语句。设函数 *node(name)* 返回最近创建的关联于 *name* 的结点。首先，置 *DAG* 图为空。对基本块的每一 TAC 语句，依次进行下列步骤：

（1）若 *node(y)* 无定义，则创建一个标记为 *y* 的叶结点，并令 *node(y)* 为这个结点；对第 1 种语句，若 *node(z)* 无定义，再创建标记为 *z* 的叶结点，并令 *node(z)*为这个结点。

（2）对于第 1 种语句，若 *node(y)*和 *node(z)*都是标记为常数的叶结点，执行 *y* op *z*，令得到的新常数为 *p*；若 *node(p)*无定义，则构造一个用 *p* 做标记的叶结点 *n*. 若 *node(y)*或 *node(z)*是处理当前语句时新构造出来的结点，则删除它；置 *node(p)=n*。这一步起到**常量合并**的作用。若 *node(y)* 或 *node(z)*不是标记为常数的叶结点，则检查是否存在某个标记为 *op* 的结点，其左孩子是 *node(y)*，而右孩子是 *node(z)*？若不存在，则创建这样的结点。无论存在或不存在，都令该结

点为 *n*。这一步有可能起到**删除公共子表达式**的作用。

（3）对于第 2 种语句，若 *node(y)*是标记为常数的叶结点，执行 op *y*，令得到的新常数为 *p*。若 *node(p)*无定义，则构造一个用 *p* 做标记的叶结点 *n*。若 *node(y)*是处理当前语句时新构造出来的结点，则删除它；置 *node(p)=n*。这一步起到**常量合并**的作用。若 *node(y)*不是标记为常数的叶结点，则检查是否存在某个标记为 *op* 的结点，其唯一的孩子是 *node(y)*？若不存在，则创建这样的结点。无论存在或不存在，都令该结点为 *n*。这一步有可能起到**删除公共子表达式**的作用。

（4）对于第 3 种语句，令 *node(y)* 为 *n*。

（5）最后，从 *node(x)*的附加标识符表中将 *x* 删除，将其添加到结点 *n* 的附加变量名字表中，并置 *node(x)* 为 *n*。这一步起到**删除无用赋值**的作用。

考虑由下列 TAC 语句序列构成的基本块：

（1） T0:=3.14
（2） T1:=2*T0
（3） T2:=R+r
（4） A:=T1*T2
（5） B:=A
（6） T3:=2*T0
（7） T4:=R+r
（8） T5:=T3*T4
（9） T6:=R-r
（10） B:=T5*T6

该基本块 DAG 图的构造过程如图 13 所示。顺序处理每条 TAC 语句后形成的子图分别如图 13（ａ）~（j）所示。

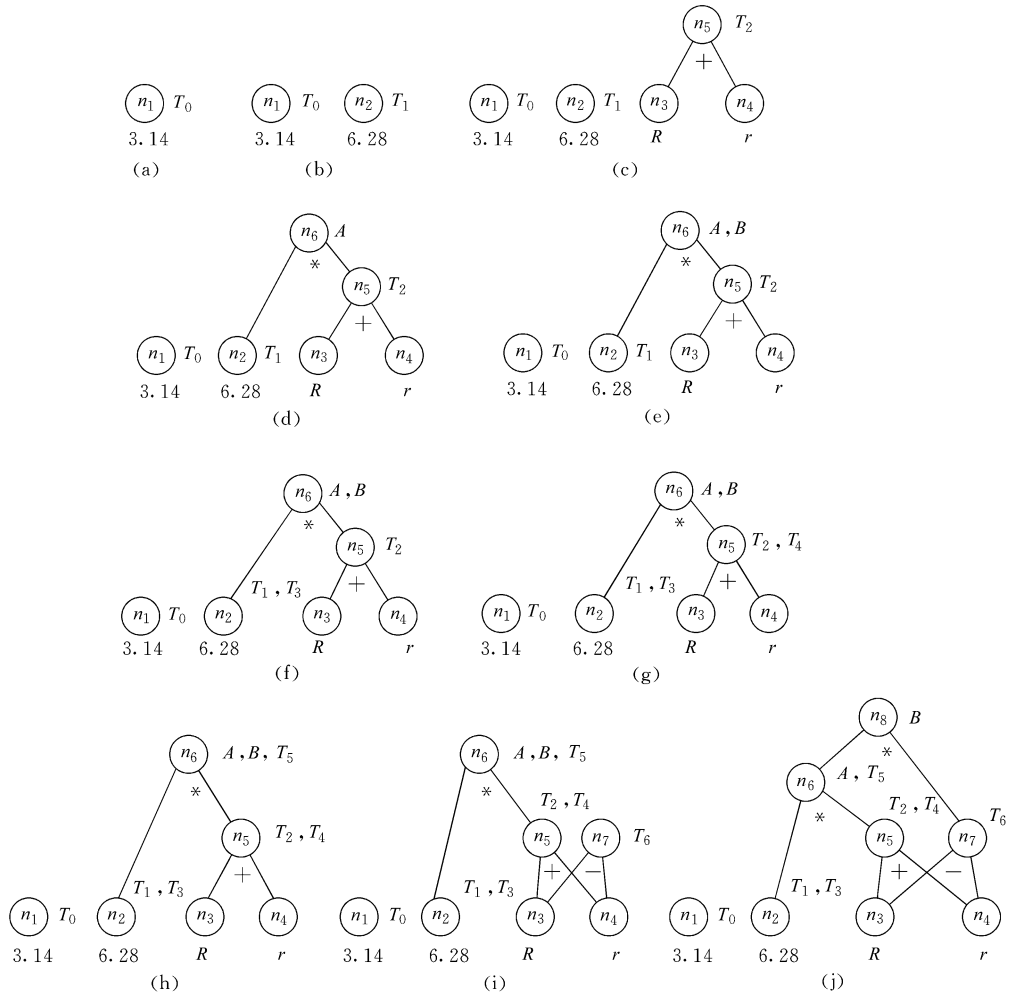

图 13 由 TAC 语句序列构造 DAG 图[2]

如前所述，在一个基本块被构造成相应的 DAG 图的过程中实际上已经进行了一些基本的优化工作。而后，我们可由 DAG 图重新生成原基本块的一个优化的语句序列。

例如，我们将如图 13（ j ）的 DAG 图按其结点构造的顺序，重新写成 TAC 语句[3]，得到如下 TAC 语句序列：

（1） T0:=3.14
（2） T1:=6.28
（3） T3:=6.28
（4） T2:=R+r
（5） T4:=T2
（6） A:=6.28*T2
（7） T5:=A
（8） T6:=R-r
（9） B:=A*T6

[2]图 13 中的 R 和 r 应改为 R_0 和 r_0.
[3]如果某个非叶子结点的附加标识符集合为空，则重新生成语句序列时需要引入一个新的标识符，以确保 DAG 图的内部结点至少会附加一个变量名字。

将这个结果和原基本块的语句序列相比，可以看出：

（1）原来的语句（2）和（6）中的常量已合并。这些常量合并的过程实际上穿插了**常量传播**：语句（1）是 T0 的定值点，其值是一个常数，T0 的值可以到达这个基本块的出口点，而且在本基本块中没有其他 T0 的定值点，因此，本基本块中所有 T0 的值都相等且为常数，这种情况下可以用常数来取代语句（1）之外的所有 T0。

（2）原来的语句（5）中的无用赋值已被删除；

（3）原来的语句和（7）中的公共子表达式 R+r 只被计算一次，即删除了多余的公共子表达式。

（4）形成结果中复写语句（5）和（7）的过程能提供**复写传播**的机会。比如，在形成结果中的语句（7），即复写语句 T5:=A 时，基本块内剩余语句中没有其他语句为 T5 定值，因此，在这些语句中均可以用 A 来代替 T5。结果，原来的语句（10），即 B:=T5*T6 最终被替换为结果中的语句（9）。这种情况下，如果在基本块出口处 T4 和 T5 不在活跃，那么就可以将（5）和（7）两条复写语句删除掉。

所以，结果 TAC 语句序列构成的基本块是原先基本块的一个优化。

顺便，除了可应用于基本块内的优化外，DAG 图还能体现出某些有用的数据流信息。例如，在基本块外被定值并在基本块内被引用的所有标识符，就是作为叶子结点上标记的那些标识符；在基本块内被定值且该值能在基本块后被引用的所有标识符，就是 DAG 图各结点上的那些附加标识符。

3.3 循环优化

循环优化是对循环中的代码进行的优化。循环内的指令是重复执行的，于大多数应用程序来讲，循环部分的执行时间在整个程序执行时间中所占的比重非常大，因此针对循环的优化通常是最值得关注的部分。有大量关于循环优化的研究成果和实用算法，限于篇幅，这里我们仅介绍最基本的两类循环优化：代码外提与归纳变量的删除。

3.3.1 代码外提

减少循环中代码数目的一个重要办法是**代码外提**（loop-invariant code motion）。这种变换把所谓的**循环不变量**（即产生的效果独立于循环执行次数的表达式计算）放到循环的前面。这里，所讨论的循环只存在一个入口。

借助于 UD 链可以查找循环不变量，如对于循环内部的语句 x:=y+z，若 y 和 z 的定值点都在循环外，则 x:=y+z 为循环不变量。

实行代码外提时，在循环的入口结点前面建立一个新结点（基本块），称之为循环的前置结点。循环的前置结点以循环的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 14 所示。由于入口结点是唯一的，所以，前置结点也是唯一的。循环中外提的代码将全部提至前置结点中。

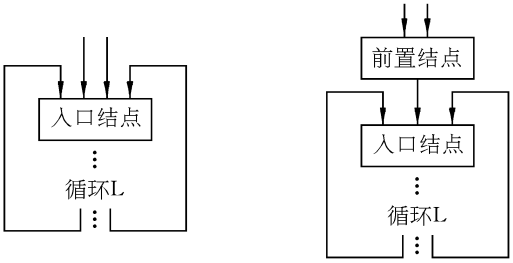

图 14 为代码外提建立前置结点

考查图 15 (a)基本块 B_2，它自身构成一个循环，可以知道其中 b 和 c 的定值点都在定值点都在循环外，表明不管基本块执行多少次，b 和 c 的值都不会改变。因此，t1 := b*c 是循环不变量。我们把这个循环不变量外提，得到图 15 (b)所示的流图。不难看出，图 15 的两个流图有相同的计算结果。

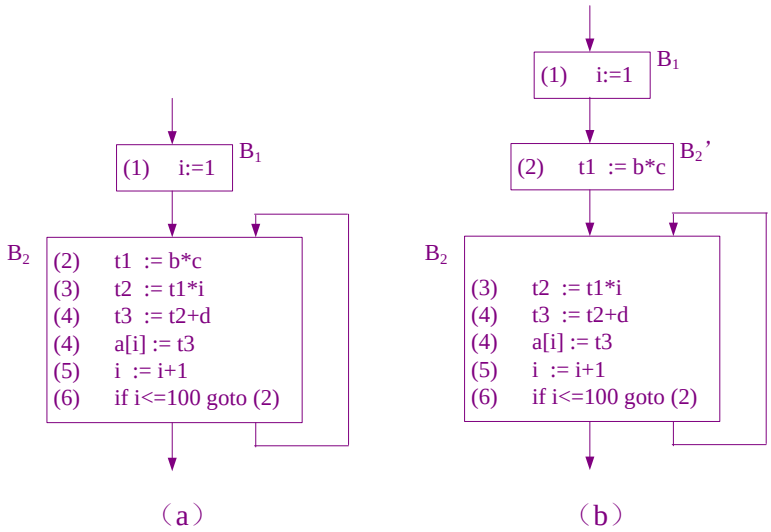

图 15 循环不变量代码外提

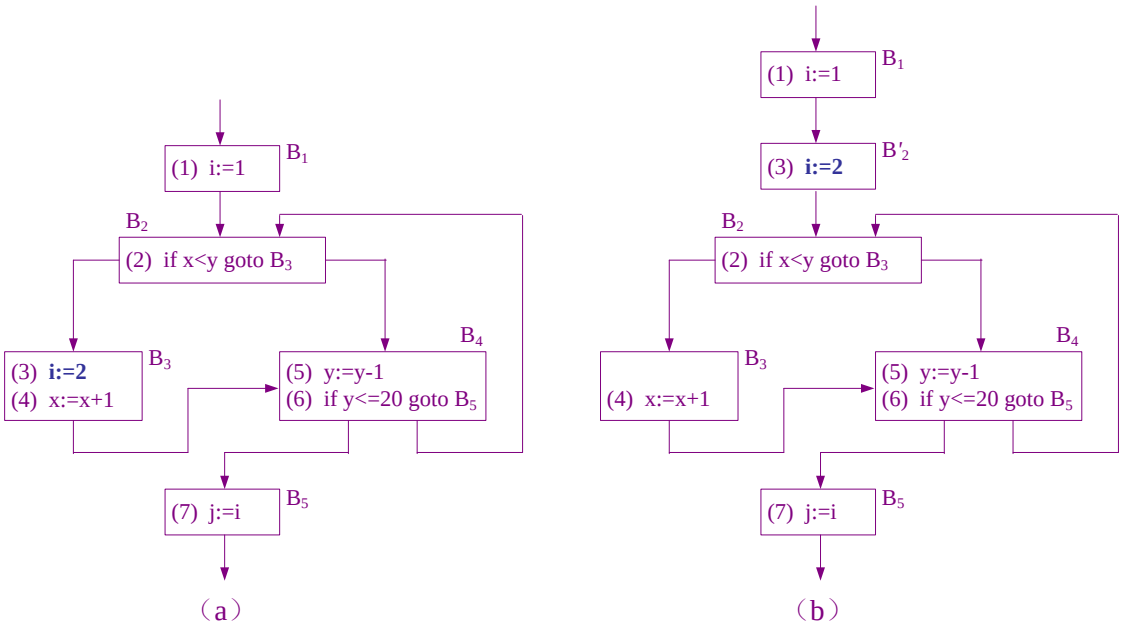

图 16 含循环不变量但不符合外体条件的流图

是否在任何情况下，都可把循环不变运量外提呢？我们来看一个例子。

考察图 16(a) 的流图。容易看出｛B_2，B_3，B_4｝是循环，其中 B_2 是循环的入口结点，B_4 是出口结点。所谓出口结点，是指从该结点有一条有向边引到循环外的某个结点。

B_3 中 i:=2 是循环不变运量。假如我们把 i:=2 提到循环的前置结点 B'_2 中，如图 16(b)所示。

若按此程序流图，执行完 B_5 时，i 的值总为 2，则 j 的值也为 2。事实上，按图 16(a)的流图，若 x=30，y=25，则 B_3 不被执行，执行完 B_5 时， i 和 j 的值都为 1 ，所以图 16(b)的流图改变了原来程序的运行结果。

问题的原因在于 B_3 不是循环出口结点 B_4 的必经结点。所以，当把一个循环不变运量提到循环的前置结点时，要求该循环不变运量所在的结点是循环所有出口结点的必经结点。此外，如果循环中 i 的所有引用点只是 B_2 中 i 的定值点所能达到的， i 在循环中不再有其它定值点，并且出循环后不再引用该 i 的值，那么，即使 B_3 不是 B 的必经结点，也还是可以把 i:=2 提到 B'_2 中，因为这不影响原来程序的运行结果。

综上所述，我们可总结出循环不变量代码外提的一个充分条件。以不变量 x:=y+z 为例，该条件可以叙述为：

（1）所在结点是循环的所有出口结点的支配结点；

（2）循环中其它地方不再有 x 的定值点；

（3）循环中 x 的所有引用点都是且仅是这个定值所能达到的；

（4）若 y 或 z 是在循环中定值的，则只有当这些定值点的语句（一定也是循环不变量）已经在之前被执行过代码外提；

或者，在满足上述 第 2、3 和 4 条的前提下，将第 1 条替换为：

（5）x 在离开循环之后不再是活跃的。

注意：如果把满足条件（2）~（5）而不满足条件（1）的循环不变量 x:=y+z 外提到前置结点中 ，那么，执行完循环后得到的 x 值，可能与不进行外提的情形所得 x 值不同。但因为离开循环后不会引用该 x 值，所以不影响程序运行结果。

根据以上讨论，我们给出一个循环不变运量代码外提的算法：

（1）为所要处理的循环为建立用于代码外提的前置结点。

（2）查看当前循环中各基本块的每条 TAC 语句，如果发现某个循环不变量，并且该循环不变量符合上述代码外提的充分条件，那么就将它插入到前置结点的尾部，即作为前置结点的最末一条语句，并将该语句从当前循环中删除。

（3）重复以上第（2）步的工作，直至当前循环中（不包括前置结点）已不存在任何符合外提充分条件的循环不变量为止。

3.3.2 归纳变量的删除

通过强度削弱和变换循环控制条件，经常会带来循环中归纳变量的优化使用甚至可以将其删除。

首先介绍基本归纳变量和归纳变量的含义。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 是循环中的基本归纳变量，J 在循环中的定值总是可以化归为 I 的同一线性函数，即 $J=C_1*I±C_2$，其中 C_1 和 C_2 都是循环不变量，则称 J 为**归纳变量**，并称它与 I **同族**。显然，基本归纳变量也是归纳变量。

一个基本归纳变量除用于自身的递归定值外，往往只在循环中用来计算其它归纳变量以及用来控制循环的进行。这时我们就可以用与循环控制条件中的基本归纳变量同族的某一归纳变量来替换它。进行这些变换后，常常会伴随着可将基本归纳变量的递归定值作为无用赋值而删除。此类变换往往也会带来运算强度的削弱，如将乘法转换成加法。循环内部的强度削弱通常是非常有价值的优化。

我们来考察图 17（a）的流图，其中基本块 B_2 和 B_3 构成循环。可以看出，x 是循环中的一个基本归纳变量，而 i 是一个与 x 同族的归纳变量。因为基本归纳变量由 x:=x+2 定值，所以可以把同族归纳变量的计算 i:=3*x 化归为 i:=i+6，也算是一种强度削弱。这样，循环控制条件 x<100 可变换为 i<300。变换后的流图如图 17（b）所示。

假如基本归纳变量 x 在 1.18（a）的循环中只用于计算归纳变量 i 和控制循环执行，当离开循环时就不活跃了。那么，在 1.18（b）的循环中，基本归纳变量 x 的递归定值就变为了无用赋值。删除基本块 B_1 和 B_3 中 x 的无用赋值，我们得到如图 17（c）所示的流图。结果，我们实现了对循环中基本归纳变量 x 的彻底删除。

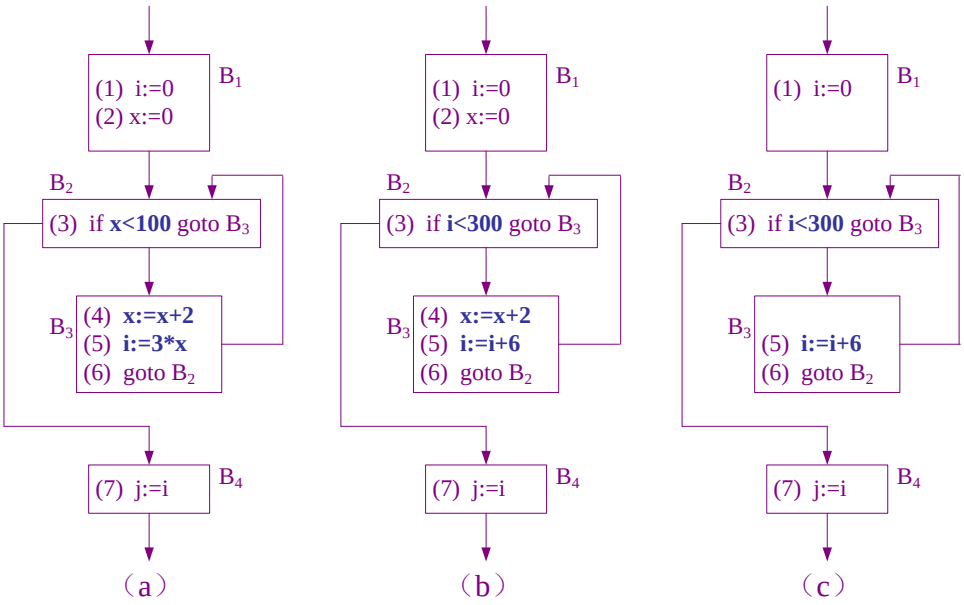

图 17 归纳变量的删除

3.4 全局优化

过程内全局优化（简称为全局优化）是在一个程序过程（C 语言中称为函数，在不引起误解的情况下统称为过程）范围内进行的优化。

前面介绍过的常量传播、常量合并、删除公共子表达式、复写传播、控制流优化和删除无用赋值等都是可用于不同范围的优化方法，也可以用到跨越多个基本块的全局优化当中，其关键点在于确定相关变量的使用情况。

考虑 3.2 小节中经过构造 DAG 图进行局部优化的例子，其优化后的结果如图 18 (a)所示。

（1） T0:=3.14
（2） T1:=6.28
（3） T3:=6.28
（4） T2:=R_0+r_0
（5） T4:=T2
（6） A:=6.28*T2
（7） T5:=A
（8） T6:=R_0-r_0
（9） B:=A*T6

(a) 局部优化结果

（1） T2:=R_0+r_0
（2） A:=6.28*T2
（3） T6:=R_0-r_0
（4） B:=A*T6

(b) 删除无用赋值

图 18 利用全局数据流信息进行优化

通过跟踪基本块之间的变量使用信息，如果我们能够判定在该基本块出口处 T0、T1、T2、T3、T4 和 T5 均不是活跃变量，而 A 和 B 是活跃变量，那么我们可以断定图 18(a)中语句（1）、（2）、（3）、（5）和（7）的定值点的 DU 链均为空集合，也就是说这些定值点的赋值都是无用的。删除这些无用赋值，优化之后的结果如图 18 (b)，其优化效果相当明显。这便是使用流图范围内的数据流信息（DU 链）进行全局删除无用赋值的结果。

回到本讲开头图 1(a)的一段三地址码程序，该程序计算一个以 16 的阶乘为半径的圆的周长，然后输出结果。为方便，我们将其重现于 20(a)。从入口指令开始，将代码分为四个基本块 *BB1*，*BB2*，*BB3*，和 *BB4*，如图 1 (b)。它们构成的流图如 2 所示。

```
(1)   pi   :=   3.14
(2)   ar   :=   0.0
(3)   n   :=   16
(4)   r   :=   1
(5)   jle n 1 (9)
(6)   r   :=   r * n
(7)   n   :=   n – 1
(8)   goto (5)
(9)   ar   := 2 * pi
(10)  ar   := ar * r
(11)  print ar
```

(a)原始代码

```
(1)   pi   :=   3.14
(2)   ar   :=   0.0
(3)   n   :=   16
(4)   r   :=   1
(5)   jle n 1 (9)
(6)   r   :=   r * n
(7)   n   :=   n – 1
(8)   goto (5)
(9)   ar   := 2 * 3.14
(10)  ar   := ar * r
(11)  print ar
```

(b)全局常量传播

```
(1)   pi   :=   3.14
(2)   ar   :=   0.0
(3)   n   :=   16
(4)   r   :=   1
(5)   jle n 1 (9)
(6)   r   :=   r * n
(7)   n   :=   n – 1
(8)   goto (5)
(9)   ar   := 6.28
(10)  ar   := ar * r
(11)  print ar
```

(c) 常量合并

```
(1)   pi   :=   3.14
(2)   ar   :=   0.0
(3)   n   :=   16
(4)   r   :=   1
(5)   jle n 1 (9)
(6)   r   :=   r * n
(7)   n   :=   n – 1
(8)   goto (5)
(9)   ar   := 6.28
(10)  ar   := ar * r
(11)  print ar
```

(d) 局部常量传播

```
(1)   n   :=   16
(2)   r   :=   1
(3)   jle n 1 (7)
(4)   r   :=   r * n
(5)   n   :=   n – 1
(6)   goto (3)
(7)   ar   := 6.28 * r
(8)   print ar
```

(e) 全局删除无用赋值

```
(1)   print 12586308608.0
```

(f) 静态计算

图 19 实例程序的全局优化

通过分析基本块内部以及基本块之间的变量使用信息，可以知道，原始代码中变量 pi 的定值点是语句(1)，其 DU 链上只有唯一的引用点是语句(9)，因此可以开展全局的常量传播优化，结果如图 19(b)。

常量传播之后，出现了新的优化机会，这时语句(9)是两个常数的运算，因此可以开展常量合并的优化，其结果如图 19(c)。

这时，进一步的常量传播，我们可以得到如图 19 (d)的结果。

图 19(d)中语句(2)、(9)和(10)中都是变量 ar 的定值点，但定值点(2)、(9)的 DU 链都是空集，因而它们是无用赋值。类似的，我们也可以确定图 19 (d)中语句(1)中对 pi 定值也无用赋值。图 19(d)中用下划线将这些无用赋值进行了标记。删除无用赋值，优化之后的结果如图 19(e)所示。

常量合并是在编译过程中进行的计算，通过编译期间来获得目标程序的结果，从而缩短所生成目标代码的运行时间。更进一步，我们讨论的图 19 的实例程序，该程序计算并输出以 16 的阶乘为半径的圆的周长，程序执行过程中不需要任何输入数据，所有参与运算的值都是已知的，因此这个程序的最终结果 ar 的值完全可以在编译过程中静态地确定。事实上，很多编译器会进行类似的优化，称为**编译过程中进行计算(Computation during Compilation)**优化。经过这样的优化，整个程序可以改写为图 19(f)的样子。所有的计算工作都已经在编译过程中完成了，程序最终运行的工作仅仅是结果输出。

4 目标代码生成技术

编译过程最后阶段的工作是生成目标体系结构的汇编语言代码或机器语言代码。通常情况下，我们面对的是真实的处理器体系结构的，如 X86、MIPS、ARM、以及 PowerPC 等。然而，有时也指特定的虚拟机结构，如 JAVA 虚拟机（JVM）。

由于和目标机环境密切相关，所以生成目标代码时需要从逻辑上考虑清楚程序中的代码和数据是如何映射到运行时的虚拟存储空间中。常量或全局量将映射到静态数据区；代码将映射到代码存放区；局部数据和临时数据的组织则是体现在所生成代码的指令中，运行时将被存放在寄存器或动态数据区的内存单元。编译器应将这些目标代码和数据以约定的形式准备好，将来由链接和装入程序加载到目标平台。或者，如果编译器生成的是汇编代码，则在运行链接和装入程序之前还需要由汇编器先生成可重定位的（relocatable）机器语言程序。

目标代码生成技术的核心问题主要包括指令选择（instruction selection）、寄存器分配（register allocation）与指令调度（code scheduling）。这些问题若是考虑最优化目标，那么各自基本上都是 NP 完全或 NP 难问题，更不用说将它们统一考虑的多目标优化问题。因此，在实际中，目标代码生成的算法多是启发式的。本课的定位是使读者了解目标代码生成的基本过程，并不会去深入探讨目标代码生成过程中所面临的优化问题。

本节首先介绍目标代码生成的主要环节（指令选择、寄存器分配与指令调度）的基本过程以及它们之间的关联，并讨论基本块范围内的一些具体做法。最后，作为这一节的结束，简单介绍图着色全局寄存器分配算法的基本思想。

4.1 目标代码生成的主要环节

4.1.1 指令选择

所谓**指令选择**，就是为每条中间语言语句选择恰当的目标机指令或指令序列。这里，中间语言语句是泛指中间表示的一个独立的操作，如在三地址码中指一条 TAC 语句，而在树形中间表示中则指其结点所代表的一个独立操作。

指令选择的原则首先是要保证语义的一致性。若目标机指令系统比较完备，则可以很直接的（在不考虑执行效率的情形下）为中间语言语句找到语义一致的指令序列模板。例如，针对某种具有 CISC 特征的计算机体系结构，TAC 语句 a:=b+c 可转换为如下汇编代码序列：

```
MOV     b, R0                   /* b 装入寄存器 R0 */
ADD     R0, c                   /* c 加到 R0 */
MOV     R0, a                   /* 存 R0 的内容 到 a */
```

其次要考虑所生成代码的效率（考虑时间/空间代价）。这并不容易做到，因为执行效率往往与该语句的上下文以及目标机体系结构（如流水线）有关。目标机指令集的性质决定指令选择的难易。一个有着丰富的目标指令集的机器中可以为一个给定的操作提供几种实现方法。比如，我们考虑因不同的寻址方式所附加的指令执行代价。假设每条指令在操作数准备好后执行其操作的代价均为 1，而是否会有附加的代价则要视获取操作数时是否访问内存而定，每访问

一次内存则增加代价 1。由此，以上汇编代码序列的执行代价为 6。同样，代码序列

```
MOV     b, a                    /*  取出 b 的值保存到 a 的存储单元  */
ADD     a, c                    /*  取出 c 和 a 的值相加结果保存到 a 的存储单元  */
```

的执行代价也为 6。然而，如假定 R1 和 R2 中已经分别包含了 b 和 c 的值，那么 a:=b+c 也可转换为下列汇编代码序列：

```
MOV     R1, R0                  /*  寄存器 R1 的内容装入寄存器 R0 */
ADD     R0, R2                  /* R2 的内容加到 R0 */
MOV     R0, a                   /*  存 R0 的内容 到 a */
```

这个代码序列的执行代价为 4。进一步，如果已知 R1 和 R2 中已经分别包含了 b 和 c 的值，并且可以知道 b 的值在 a:=b+c 这个赋值以后不再需要，那么 a:=b+c 可以只转换为下列汇编代码序列：

```
ADD     R1, R2                  /* R2 的内容加到 R1 */
MOV     R1, a                   /*  存 R1 的内容 到 a */
```

该代码序列的执行代价为 3，执行的效率明显提高，表明生成了更优的目标代码。

对于执行代价的考虑，可以不局限于性能（执行周期数），还可以考虑其他指标，如代码的尺寸（条数）。

可以根据不同上下文为每条中间语言语句设计指令序列的模板，这样可以直接编写**代码生成器**（code generator）实现指令选择。此外，人们还提出了许多实现指令选择的自动化方法，从而构造所谓的**代码生成器的生成器**（code generators generator）。这些方法中影响较大有 BURG 和 Twig 工具，它们都是基于动态规划（dynamic programming）的方法，所生成的代码生成器都是以树形中间表示为输入，然后自动完成指令选择并生成目标代码。

BURG 是基于自下而上重写系统（BURS，bottum-up rewriting systems）理论[PLG88]构造的一种有效的代码生成器，它在进行动态规划时使用了预先构造的一种特殊的 BURS 自动机，经历自下而上的标记过程和自上而下的指令选择过程快速地生成目标代码。

Twig 是基于一种树模式匹配（tree pattern matching）方法[AGT89]构造的工具。给定树模式规范和相应的执行代价（cost），Twig 能够生成一个自上而下的树自动机，后者能够为树形中间表示找到一种最小代价的覆盖。与 BURS 方法相比，Twig 可以在编译时动态计算模式的执行代价，而 BURS 是在编译前就已经计算好这些代价，因而灵活性和适应性方面不如 Twig。BURS 方法的优势是速度比较快。

4.1.2 寄存器分配

通常情况下，指令在寄存器中访问操作数的开销要比在内存中访问小很多。同时，一些像 RISC 这样的体系结构往往要求除 loads/stores 之外的指令都使用寄存器操作数。因此，在生成的代码中，尽可能多地、有效地利用寄存器非常重要。寄存器分配可以分成分配和指派两个阶段来考虑：

（1）在**分配**（allocation）期间，为程序的某一点选择驻留在寄存器中的一组变量；

（2）在随后的**指派**（assignment）阶段，挑出变量将要驻留的具体寄存器，即寄存器赋值。

寄存器分配的原则是充分、高效地使用寄存器。一方面，应尽量让变量的值或计算结果保留在寄存器中。另一方面，不再被引用的变量所占用的寄存器应尽早释放，以提高寄存器的利用率。选择最优的寄存器分配方案是困难的。从数学上讲，这是 NP 完全问题。当考虑到目标处理器硬件和操作系统可能要求寄存器的使用遵守一些约定时，这个问题将更加复杂。因此，

实际编译器中通常采用某种启发式算法，在尽可能短的时间内寻找一种较优的结果。

在基本块范围内的寄存器分配称为**局部寄存器分配**（local register allocation），在过程范围内的寄存器分配称为过程级寄存器分配（procedure-level register allocation）或**全局寄存器分配**（global register allocation）。

寄存器是目标计算机系统的紧缺资源。CISC 特征的体系结构中可用于应用程序的**通用寄存器**（general purpose registers）很少，如 X86-32（IA32）有 8 个通用寄存器。RISC 特征的体系结构中通用寄存器数目相对多一些，如 MIPS-32 有 32 个通用寄存器。一般情况下，就是通用寄存器也不能全部用来自由分配。在寄存器分配时，一定要明确目标环境（处理器和操作系统）下有关寄存器使用的约定。通常，可以把通用寄存器分为可分配寄存器、保留寄存器以及工作寄存器等类别。

可分配寄存器（allocatable registers）是可以用于自由分配和释放的寄存器。一旦分配给特定的变量，这些寄存器就受到了保护，在完成特定任务之前不会再分配给其他变量。在某个寄存器的特定任务结束后，编译器必定会将它释放，此后便可自由分配给其他变量了。寄存器分配的一个重要方面就是尽可能使某个寄存器最大限度地被多个变量“分享”，达到有效使用寄存器的目标。

保留寄存器（reserved registers）是在整个程序范围内起固定作用的那些寄存器。这些寄存其如栈顶指针寄存器，栈桢指针寄存器，Display 寄存器，参数和返回值寄存器，以及返回地址寄存器等等。最好不要随意将这些寄存器用于完成约定功能之外的任务，否则会造成不兼容甚至难以想象的后果。

工作寄存器（work registers）是代码生成过程中可随时短暂使用但用完后必须马上释放的寄存器。此类寄存器不需要多，通常 3、4 个就足够了。至于将哪些通用寄存器用作工作寄存器，可以固定下来，也可以由临时设定。有了工作寄存器的存在，就可以不用担心在临时需要时没有寄存器可用，也可以简化寄存器分配算法。比如，MIPS 的 add 指令需要所有操作数在寄存器中，若某个操作数不在寄存器中，那么就可以将它临时装入工作寄存器中，add 指令执行结束后就马上释放这个寄存器以备再次临时使用。另外，工作寄存器的存在还会使我们感觉到有比实际更多的寄存器，比如在寄存器分配算法中可以假定没有寄存器数目的限制，即可以使用为**伪寄存器**（pseudo-registers）。伪寄存器不是真实的物理寄存器，而是由对应的存储单元模拟的，在需要实行物理寄存器作用时，就可以将它们的值取到工作寄存器中，用后者替代之。今后在不至于混淆的情况下，我们将不是真实的寄存器统称为伪寄存器。

4.2 节介绍以基本块为单位的一种简单代码生成算法，其中寄存器分配是一种简单的局部寄存器分配，并且寄存器数目没有设定上限，即可以使用伪寄存器。4.3 节是关于高效使用寄存器的内容，从基本块的 DAG 图生成 TAC 语句的次序与寄存器分配的关系，目标是使所产生的 TAC 语句尽可能节省使用寄存器，同时还介绍一种表达式求值过程中使用最少数目寄存器的经典方法。在 4.4 节，将介绍图着色寄存器分配算法的基本思想，可应用于全局寄存器分配。

在寄存器分配方案确定后，具体的寄存器指派就不难实现了。

4.1.3 指令调度

指令调度是指对指令的执行顺序进行适当的调整，从而使得整个程序得到优化的执行效果。指令调度对于现代计算机系统结构的高效使用是十分重要的环节，比如对于具有流水线的体系结构，指令调度阶段往往是必需的。例如 RISC 体系结构一种通用的流水线限制为： 从内存中取入寄存器中的值在随后的某几个周期中是不能用的。在这几个周期中，调出不依赖于该取入值的指令来执行是很重要的。必须尽可能找出一条或若干条指令（与被取值无关），在取值指令之后能立即执行，如果找不到相应的指令，这些周期就会被浪费。

指令调度算法可以是局限于基本块范围内，也可以是更大范围的全局指令调度（global code

scheduling）；可以是仅静态地完成指令执行顺序的调整，也可以是实现动态指令调度（dynamic code scheduling）。指令调度的更具体内容超出本课范围，这里不去进一步讨论。

4.2 简单的代码生成算法

这一小节里，我们介绍非常简单的目标代码生成的例子，面向单个基本块的代码生成，采用简易的寄存器分配算法，使大家能够基本了解代码生成的完整过程。

我们选择了两个基本块内代码生成的简单例子，一个采用了与当前 MiniDecaf 实验框架类似的算法，另一个例子的代码生成算法等效于龙书中所用的算法。

在本节的例子中，我们假设基本块中只有形如 A:=B op C 和 A:=B 的 TAC 语句序列。其中，op 为二元运算（如加法和减法运算）。为简化讨论，还假定 A、B 和 C 均为变量（容易扩展至含有常量的情形）。同时，假设生成目标代码时仅用到下列格式的指令：

（1）OP　reg0, reg1, reg2

其中，reg0, reg1, reg2 处可以是任意的寄存器。运行这些指令时，对 reg1 和 reg2 的值做二元运算，或者对 reg1 的值做一元运算，结果存入 reg0。

（2）LD　reg, mem　　/* Load 指令，取内存量 mem 的值到寄存器 reg */

（3）ST　reg, mem　　/* Store 指令，存寄存器 reg 的值到内存量 mem */

（4）MV　reg0, reg1　　/* Move 指令，将寄存器 reg1 的内容复制寄存器 reg0 */

假设指令选择可以通过简单对应来完成，这样代码生成算法的核心是处理好在基本块范围内如何合理有效地使用寄存器的问题，以下是部分需要遵循的基本原则：

（1）生成某变量的目标对象值时，尽量让变量的值或计算结果保留在寄存器中，必要时才用 ST 指令将其 Spill 到内存。

（2）尽可能引用变量在寄存器中的值（对于 CISC 目标机）

（3）保证语义一致的前提下，尽可能在取值相同的变量间共享寄存器。

（4）在同一基本块内，后面不再被引用的变量所占用的寄存器应尽早释放。

不同的算法使用寄存器的方式不同，其合理性和有效性会有不同程度的差异。这些算法往往会借助于在基本块范围内建立的变量待用信息链和活跃信息链。

此外，当处理到基本块出口时，需要将变量的值存放在内存中。因为一个基本块可能有多个后继结点或多个前驱结点，同名变量在不同前驱结点的基本块内，出口前存放的寄存器可能不同，或没有定值，所以应在出口前把寄存器的内容放在内存中，这样从基本块外进入的变量值都在内存中。

4.2.1 简单代码生成算法的一个例子

我们先看一个与 MiniDecaf 实验框架（C++版）中的简单代码生成算法相类似的例子。

算法中将会用到以下寄存器描述数组及变量位置查询函数两组信息：

（1）寄存器描述数组 VALUE。VALUE[R] 描述寄存器 R 当前存放哪个变量。（注意：该算法中，一个寄存器仅存放单个变量）

（2）变量位置查询函数 lookupReg。lookupReg (V) 表示查询变量 V 存放在哪个寄存器中，若不存放在寄存器中则返回 NULL。（注意：该算法中，一个变量至多存放在一个寄存器中）

另外，寄存器分配函数 allocReg(V) 表示为变量 V 分配新的寄存器。函数 allocReg 随后给

出。

下面是该算法的描述：

（1）对每个 TAC 语句 I: A:=B op C 或 i: A:=B，依次执行下述步骤：

- 调用 lookupReg(A) 确定目的寄存器，若 A 已分配寄存器，则使用已分配的寄存器，命名为 A`；若未分配，则调用 allocReg(A) 为变量 A 分配新的寄存器，同命名为 A`。
- 调用 lookupReg(B) 和 lookupReg(C)，确定 B 和 C 现行值存放位置；如果其现行值在寄存器中，则把寄存器取作 B` 和 C`；如果 B (C) 的现行值不在寄存器中，则调用 allocReg(B) (allocReg(C)) 为 B (C) 分配寄存器 B` (C`)，且使用 LD 指令将 B (C) 从内存中取出，存到寄存器 B` (C`) 中。
- 按照上述寄存器分配结果更新 VALUE[A`]、VALUE[B`]、VALUE[C`] 。
- 分两种情形生成目标代码：

a) 对于 I: A:=B op C，分情况生成如下代码序列：

若 B 和 C 不在寄存器中，则生成

LD B`, B

LD C`, C

OP A`, B`, C`

若 B 和 C 中有一个在寄存器中，则参考以上目标代码，去掉对应的 LD 指令即可；

若 B 和 C 都在寄存器中，则生成

OP A`, B`, C`

b) 对于 I: A:=B，分情况生成如下代码序列：

若 B 在寄存器中，则生成

MV A`, B`

若 B 不在寄存器中：

LD B`, B

MV A`, B`

- 如 B (C) 的现行值在基本块中不再被引用，也不是基本块出口之后的活跃变量（由语句 I 上的附加信息得知），且其现行值在某个寄存器 R_k 中，则删除 VALUE[R_k] 中的 B (C)，同时使该寄存器不再被 B (C) 所占用。

（2）处理完基本块中所有 TAC 语句之后，对现行值在某寄存器 R_i 中的每个变量 M_i，若它在出口之后是活跃的，则生成 “ST R_i , M_i”，将其存入内存。

下面是函数 allocReg 的描述。

allocReg 功能：以变量 V 为参数，返回一个寄存器。

（设{ R_k } 为当前所有寄存器的集合）

步骤：

- 若 $\{R_k\}$ 中存在未被使用的寄存器，则直接返回任一未被使用的 R_i，且置 $VALUE[R_i] = V$。
- 若 $\{R_k\}$ 中不存在未被使用的寄存器，即所有寄存器都被占用的情况下：

  若 $\{ VALUE[R_k] \}$ 中存在不再被该基本块引用的变量 $VALUE[R_i]$，且该变量不是该基本块出口之后的活跃变量，则返回该变量对应的寄存器 R_i，且令 $VALUE[R_i] = V$ 。

  若 $\{ VALUE[R_k] \}$ 中不存在上述“非活跃”变量，则选择从分析该基本块到当前时刻最久未被使用的寄存器 R_i (LRU 算法)，使用 ST 指令将该寄存器中存储的变量存入内存，返回 R_i 作为新的可使用的寄存器，且令 $VALUE[R_i] = V$。

下面看一个简单的例子。设有以下 TAC 语句序列组成的基本块

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

假定在该基本块出口处，b 和 d 是活跃的，其他变量均不活跃。

为简单，设 a, b, c 和 d 在内存中的位置用同样的符号表示。

以该基本块的语句序列为输入，利用上述算法所生成的代码序列如图 20 第 2 列所示。算法执行过程以及相关描述信息的变化情况均反映在图 20 中。

语句	生成的代码	说明

t: = a－b	LD R_1, a
	LD R_2, b
	SUB R_0, R_1, R_2
		VALUE[R_0]=t
		VALUE[R_2]=b

a := b	MV R_1, R_2	VALUE[R_0]=t VALUE[R_1]=a VALUE[R_2]=b
u: = a－c	LD R_4, c SUB R_3, R_1, R_4	VALUE[R_0]=t VALUE[R_2]=b VALUE[R_3]=u
v: = t＋u	ADD R_1, R_0, R_3	VALUE[R_1]=v VALUE[R_2]=b VALUE[R_3]=u
d: = v＋u	ADD R_0, R_1, R_3 ST R_0, d ST R_2, b	VALUE[R_0]=d VALUE[R_2]=b 将活跃变量b和d存入内存

图 20 一个简单的目标代码生成过程举例（对应 4.4.1 节）

4.2.2 简单代码生成算法的另一个例子 （供课后参考）

下面的算法与龙书所介绍的算法类似，可参考较新版龙书[Aho07]的第 8.6 节。本学期课堂上不介绍这一算法，仅供大家课后参考。这一代码生成算法比 4.2.1 中的算法对寄存器的使用更加合理和有效。

假设为 TAC 语句 *I* 中的内存变量选择寄存器的函数为 *getreg*(*I*)。我们首先介绍这一简单代

码生成算法的基本步骤，然后再来讨论 *getreg*(*I*)函数。我们假定有足够的寄存器，使得在把值存放回内存、释放所有寄存器后，空闲的寄存器足以完成任何 TAC 语句的运算。

算法中将会用到以下两组信息：

（1）寄存器描述函数 RVALUE。RVALUE[R] 描述寄存器 R 当前存放哪些变量。

（2）变量地址描述函数 AVALUE。AVALUE[A] 表示变量 A 的值存放在哪些位置中，这些位置可以是寄存器、也可以是存储位置。一个变量可以存放在一个或多个位置。

下面是该算法的描述：

（1）对每个 TAC 语句 *I*: A:=B op C 或 *I*: A:=B，依次执行下述步骤：

- 以 *I* 为参数，调用 getreg(*I*); 从 getreg 返回时，得到作为存放 A、B 和 C 现行值所选择的寄存器 R_a、R_b 和 R_c；函数 getreg 随后给出。
- 分两种情形生成目标代码：

  （a) 对于 *I*: A:=B op C。

  如果 RVALUE[R_b]不包含 B，则生成一条指令“LD R_b, mem_b”，这里 mem_b 为 B 的一个内存位置。同时，修改 RVALUE[R_b]使之包含 B，修改 AVALUE[B]使之包含 R_b，以及对于每个 X≠B，从 AVALUE[X]中删除 R_b。

  类似地，如果 RVALUE[R_c]不包含 C，则生成一条指令“LD R_c, mem_c”，这里 mem_c 为 C 的一个内存位置。同时，修改 RVALUE[R_c]使之包含 C，修改 AVALUE[C]使之包含 R_c，以及对于每个 X≠C，从 AVALUE[X]中删除 R_c。

  然后，生成相应二元运算的指令。假设 op 对应的指令为 ADD，则生成“ADD R_a, R_b, R_c”。另外，修改 RVALUE[R_a]使之仅包含 A，修改 AVALUE[A]使之仅包含 R_a，以及对于每个 X≠A，从 AVALUE[X]中删除 R_a。

  （b) 对于 *I*: A:=B。

  此时，getreg(*I*)总是为 A 和 B 返回同一个寄存器，即 R_a=R_b。

  如果 RVALUE[R_b]不包含 B，则生成一条指令“LD R_b, mem_b”，这里 mem_b 为 B 的一个内存位置。同时，修改 RVALUE[R_b]使之包含 A 和 B，修改 AVALUE[B]使之包含 R_b，修改 AVALUE[A]使之仅包含 R_b，以及对于每个不同于 A 和 B 的变量 X，从 AVALUE[X]中删除 R_b。

  如果 RVALUE[R_b]已包含 B，则修改 RVALUE[R_b]使之也包含 A，以及修改 AVALUE[A]使之包含 R_b。

（2）处理完基本块中所有 TAC 语句之后，对现行值在某寄存器 R 中的每个变量 X（RVALUE[R]包含 X），若它在出口之后是活跃的，但 AVALUE[X]中不包含 X 的任何内存位置，则生成 Store 指令 “ST R, mem_x”，将寄存器 R 中的 X 值保存到它自己的内存位置 mem_x 中。

下面是函数 getreg 的描述。

getreg(*I*) 功能：以 *I*: A:=B op C 或 *I*: A:=B 为参数，返回为存放 A、B 和 C 现行值所选择的寄存器 R_a、R_b 和 R_c。

步骤：

- 对于 *I*: A:=B op C。

  首先，必须为 B 和 C 分别选择一个寄存器，我们以 B 为例（C 类似），通过以下步

骤返回寄存器 R_b（对于 C，为 R_c）。

1）若存在 R，B∈RVALUE[R]，则返回 R_b 为 R；

2）否则，若 B 不在寄存器中,若当前存在可分配的空闲寄存器 R，则返回 R_b 为 R；

3）否则，若既不在寄存器中也不存在空闲寄存器，则选择一个的可行寄存器 R，使得对任何 X∈RVALUE[R]，能够确保 X 的值不会再次使用，即符合以下情形之一：

（a）若 AVALUE[X]中除 R 之外，还有其他位置保存 X；

（b）若 X 是 A，但不是另一操作数 C；

（c）若 X 在语句 *I* 之后不再被使用；（通过代用信息链获知）

（d）若上面的条件均不满足，则生成 Store 指令 “ST R, mem_x”，即把寄存器 R 中保存的 X 的值复制到它自己的内存位置 mem_x 上去,称之为“spill”操作；修改 AVALUE[X]使其包含 mem_x。

在符合上述情形的寄存器中选择一个“spill”操作最少的 R，并返回 R_b 为 R。

R_c 的选择与 R_b 类似，可将上述步骤中的 B 改为 C，以及 C 改为 B。

对于 R_a 的选择，可通过如下步骤：

1）若存在 R，A∈RVALUE[R]，且 RVALUE[R]中除 A 之外没有其他变量，则返回 R_a 为 R；

2）否则，若 B 在语句 *I* 之后不再被使用，且（在必要时加载 B 之后）R_b 仅保存了 B 的值，则返回 R_a 为 R_b。对于 C 和 R_c，也有类似的选择。

3）否则，可按照上面选则 R_b（或 R_c）的步骤 3）来确定 R_a。

- 对于 *I*: A:=B。

  可按照上面描述的步骤先选定 R_b，然后再令 R_a=R_b。

同样，我们看一个与 4.2.1 一样的例子，即以下 TAC 语句序列组成的基本块

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

假定在该基本块出口处，a, b, c 和 d 是活跃的，其他变量均不活跃。注：这里与 4.2.1 的例子有些差别，4.2.1 的例子中仅 b 和 d 在基本块出口处是活跃的。

同样，为简单起见，设 a, b, c 和 d 在内存中的位置用同样的符号表示。另外，假定这一基本块所生成的代码仅使用 3 个寄存器 R0, R1 和 R2。大家可以对比一下，在出口处 a, b, c 和 d 均活跃的前提下，若使用 4.2.1 的算法，3 个寄存器是绝对不够的。

以该基本块的语句序列为输入，利用上述算法所生成的代码序列如图 20 第 2 列所示。算法执行过程以及相关描述信息的变化情况均反映在图 21 中。初始时，各个寄存器描述为空，每条 TAC 语句的代码生成后寄存器描述的变化如第 3 列所示，而变量地址描述的变化见第 4 列。

语句	生成的代码	寄存器描述	变量地址描述

t: = a－b	LD R_0, a
	LD R_1, b
	SUB R_0, R_0, R_1
		R_0 包含 a, t
		R_1 包含 b
			a 在 R_0 和 a 中
			t 在 R_0 中
			b 在 R_1 和 b 中

a := b		R_0 包含 t R_1 包含 a, b	t 在 R_0 中 a 在 R_1 中 b 在 R_1 和 b 中
u: = a－c	LD R_2, c SUB R_2, R_1, R_2	R_0 包含 t R_1 包含 a, b R_2 包含 u	t 在 R_0 中 a 在 R_1 中 b 在 R_1 和 b 中 c 在 c 中 u 在 R_2 中
v: = t＋u	ADD R_0, R_0, R_2	R_0 包含 v R_1 包含 a, b R_2 包含 u	a 在 R_1 中 b 在 R_1 和 b 中 c 在 c 中 u 在 R_2 中 v 在 R_0 中
d: = v＋u	ADD R_0, R_0, R_2 ST R_1, a ST R_0, d	R_0 包含 d R_1 包含 a, b R_2 包含 u	a 在 R_1 和 a 中 b 在 R_1 和 b 中 c 在 c 中 d 在 R_0 和 d 中

图 21 另一个简单的目标代码生成过程举例（对应 4.4.2 节）

4.3 高效使用寄存器

如前所述，可供分配的寄存器数目极其有限，因而如何高效使用寄存器是目标代码生成时重点考虑的问题。一方面要尽可能地让变量的值保留在寄存器中，尽可能引用变量在寄存器中的值，而另一方面则需要尽可能早地释放寄存器，而让其他变量可以获得寄存器。这是很难调和的两个方面。

关于如何有效使用寄存器的话题有许多。下面我们介绍一个关于使用最少的寄存器进行表达式求值的方法，对于在基本块内高效使用寄存器非常有用。这一方法适用于诸如 MIPS 之类的 RISC 机器。

假设在一个简单的基于寄存器的机器上进行表达式求值，除了 load/store 指令用于寄存器值的装入和保存外，其余操作均由下列格式的指令完成：

```
OP    reg0, reg1, reg2
OP    reg0, reg1
```

其中，reg0, reg1, reg2 处可以是任意的寄存器。运行这些指令时，对 reg1 和 reg2 的值做二元运算，或者对 reg1 的值做一元运算，结果存入 reg0。对于 load/store 指令，假设其格式为：

```
LD    reg, mem          /* 取内存或立即数 mem 的值到寄存器 reg */
ST    reg, mem          /* 存寄存器 reg 的值到内存量 mem */
```

该方法首先对表达式树（表达式的抽象语法树）的每个结点用所谓的 **Ershov 数**（Ershov number）进行标记。如果不考虑寄存器的泄漏，并假设不考虑可能的优化因素（如公共子表达式删除），那么每个结点的 Ershov 数就是对应这个结点的表达式求值时所需寄存器数目的最小值。用 Ershov 数标记表达式树结点的算法是：

（1）用 1 标记所有叶子结点；

（2）对仅有一个孩子的内部结点，其标记沿用孩子结点的标记；

（3）对于有两个孩子的内部结点，若两个孩子的标记不同，则用较大的一个来标记该结点；若两个孩子的标记相同，则将这个标记数加 1 后对该结点进行标记。

例如，图 22 是一个用 Ershov 数标记表达式树结点的例子，对应的表达式为 (a+b)*((d+e)-c)。

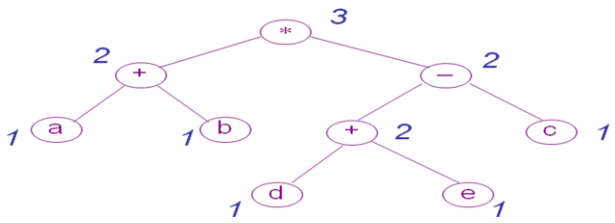

图 22　用 Ershov 数标记表达式树结点

在完成标记后，就可以生成使用寄存器最少的目标代码了。我们来介绍一种实现这个过程的基本算法。假设表达式树结点 n 的 Ershov 数为 k（>0），表达式求值过程使用 k 个伪寄存器 R_0，R_1，…，R_{k-1}，并且求值结果存在 R_0 中。这种代码生成算法的思想是：

（1）如果 n 的左子树根结点的标记大于右子树根结点的标记，那么先递归求值左子树，求值过程使用伪寄存器 R_0，R_1，…，R_{k-1}，结果存放于 R_0；然后再递归求值右子树，求值过程使用伪寄存器 R_b，…，R_{k-1}（0<b<k，右子树求值所需寄存器的数目为 k-b，少于 k），结果存放于 R_b；最后生成一条形如 OP R_0, R_0, R_b 的指令。

（2）如果 n 的左子树根结点的标记小于右子树根结点的标记，那么情况和（1）刚好相反，即先递归求值右子树，求值过程使用伪寄存器 R_0，R_1，…，R_{k-1}，结果存放于 R_0；然后再递归求值左子树，求值过程使用伪寄存器 R_b，…，R_{k-1}（0<b<k，左子树求值所需寄存器的数目为 k-b，少于 k），结果存放于 R_b；最后生成一条形如 OP R_0, R_b, R_0 的指令。

（3）如果 n 的左子树根结点的标记与右子树根结点的标记相等，子树求值的次序不重要，比如我们可以先递归求值左子树，求值过程使用伪寄存器 R_0，R_1，…，R_{k-2}，结果存放于 R_0；然后再递归求值右子树，求值过程使用伪寄存器 R_1，…，R_{k-1}，结果存放于 R_1；最后生成一条形如 OP R_0, R_0, R_1 的指令。

（4）如果 n 只有一个孩子，那么先递归求值这个为根结点的子树，求值过程使用伪寄存器 R_0，R_1，…，R_{k-1}，结果存放于 R_0；最后生成一条形如 OP R0, R0 的指令。

（5）如果 n 是叶子结点，则生成一条形如装入到结果寄存器 reg 的 load 指令 LD reg, mem。

根据这一算法，由图 22 的表达式树生成的代码为：

```
LD    R0, a
LD    R1, b
ADD   R0, R0, R1
LD    R1, d
LD    R2, e
ADD   R1, R1, R2
```

LD R_2, c
SUB R_1, R_1, R_2
MUL R_0, R_0, R_1

以上算法中，我们假设了实际物理寄存器的数目不少于 Ershov 数。如果是少于 Ershov 数的情况，则要对这个算法进行调整，在适当的地方插入 store 指令，将相应的伪寄存器泄漏到（spilled into）内存。**Sethi–Ullman 算法**（Sethi–Ullman algorithm [SU70]）完整描述了这个过程，限于篇幅，这里不作进一步介绍。

4.4 图着色寄存器分配

前面也提到过，寄存器分配可分为两个部分，即分配和指派。因此，可以将其认为是一个两遍的过程：

（1）第一遍先假定可用的通用寄存器是无限数量的，完成指令选择和生成。

（2）第二遍将物理寄存器指派到伪寄存器。物理寄存器数量不足时，会将一些伪寄存器泄露到（spilled into）内存。图着色算法的核心任务就是使得泄露的伪寄存器数目最少。

下面介绍一个基本的图着色全局寄存器分配算法。它基于特定的**寄存器相干图**（register interference graph）。本课中的寄存器相干图是一个无向图，每个伪寄存器是图中的一个结点；如果程序中存在某点，一个结点在该点被定值，而另一个结点在紧靠该定值之后的点是活跃的，则在这两个结点间连一条边。

图 23 的流图中给出了每个定值点之后的活跃变量信息。据此，可以给出该流图对应的寄存器相干图，如图 23 所示。

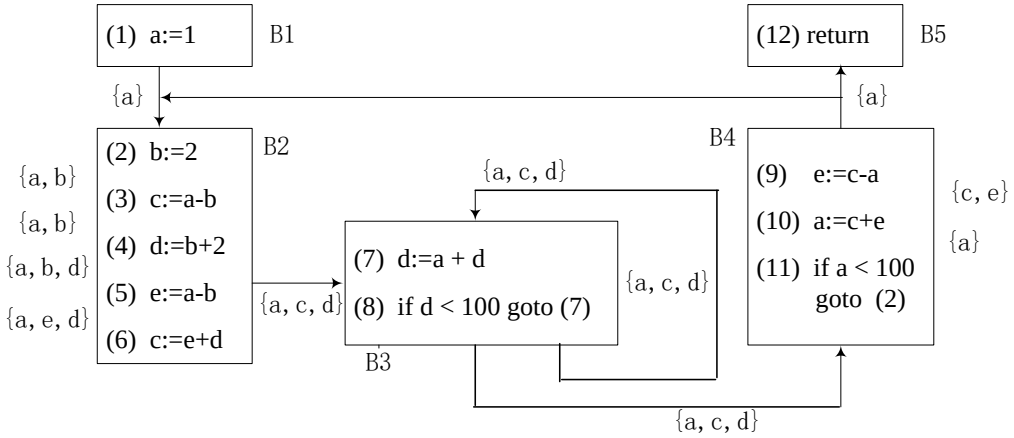

图 23　定值点之后的活跃变量信息

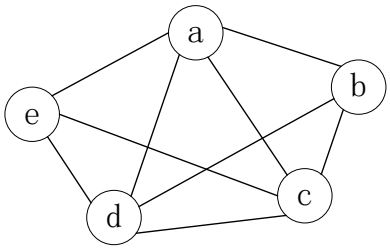

图 24　寄存器相干图

对相干图进行**着色**（coloring），是指使用 k（对应物理寄存器的数量）种颜色对相干图进行着色，使得任何相邻的结点均具有不同的颜色（即两个相干的伪寄存器不会分配到同一个物理

寄存器）。这样，我们就把物理寄存器指派的问题转化成了图伦问题。

“一个图是否能用 k 种颜色着色”是一个 NP-完全问题。以下是一个简单的启发式 k-着色算法思想：

（1）假设图 G 中某个结点 n 的度数小于 k，从 G 中删除 n 及其邻边得到图 G’，对 G 的 k-着色问题可转化为先对 G’进行 k-着色，然后给结点 n 分配一个其相邻结点在 G’的 k-着色中没有使用过的颜色。

（2）重复（1）的过程从图中删除度数小于 k 的结点。如果可以到达一个空图，说明对原图可以成功实现 k-着色；否则，原图不能成功实现 k-着色，可从 G 中选择某个结点（作为泄露候选）将其删除，算法可继续。

对于图 24 的寄存器相干图，取 k=4，则可以成功着色。倘若真实的可分配物理寄存器数目不足 4，则必将会选某些结点所代表的伪寄存器泄漏到内存中去。

最最后要指出的是，以上寄存器相干图仅适用于特定情况。每个结点对应于流图范围内需要分配寄存器的变量，每个变量都是在定值时将被分配寄存器，寄存器分配算法面向整个流图范围（比如，可以在前述的 getreg 函数基础上进行扩展）。此外，若是先进行某些代码优化（比如，在图 23 的流图范围内去掉无用定值点），然后再生成寄存器相干图，可能会有不同的着色效果。

对于不同的目标指令集，不同的寄存器分配算法，不同的优化目标和范围，可根据需要定义不同的寄存器相干图。

练习

1. 图26是图25的C代码的部分三地址代码序列。

```
void quicksort(m,n)
int 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);
}
```

图25

(1) 请将图26的三地址代码序列划分为基本块并给出其流图。

(2) 将每个基本块的公共子表达式删除。

(3) 找出流图中的循环，将循环不变量计算移出循环外。

(4) 找出每个循环中的归纳变量，并在可能的地方删除它们。

(1)	i := m-1	(16)	t_7 := 4*i
(2)	j := n	(17)	t_8:= 4*j
(3)	t_1 := 4*n	(18)	t_9:= a[t_8]
(4)	v := a[t_1]	(19)	a[t_7] := t_9
(5)	i := i+1	(20)	t_{10}:= 4*j
(6)	t_2 := 4*i	(21)	a[t_{10}]:= x
(7)	t_3 := a[t_2]	(22)	goto (5)
(8)	if t_3<v goto (5)	(23)	t_{11} := 4*i
(9)	j := j-1	(24)	x := a[t_{11}]
(10)	t_4 := 4*j	(25)	t_{12} := 4*i
(11)	t_5 := a[t_4]	(26)	t_{13} := 4*n
(12)	if t_5>v goto (9)	(27)	t_{14} := a[t_{13}]
(13)	if i>=j goto (23)	(28)	a[t_{12}] := t_{14}
(14)	t_6 := 4*i	(29)	t_{15} := 4*n
(15)	x := a[t_6]	(30)	a[t_{15}] := x

图26

2. 在图27的程序流图中，B_3 中的 i:=2 是循环不变量，可以将其提为前置结点吗？你还能举出一些例子说明循环不变量外提的条件吗?

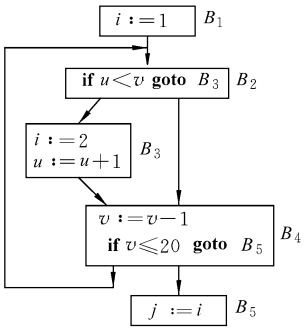

图27

3. 试对以下基本块B_1和B_2：

B_1:	A := B*C	B_2:	B:=3
	D := B/C		D:=A+C
	E := A+D		E := A*C
	F := 2*E		F := D+E
	G := B*C		G := B*F
	H := G*G		H := A+C
	F := H*G		I := A*C
	L := F		J := H+I
	M := L		K := B*5
			L := K+J
			M := L

分别应用DAG对它们进行优化，并就以下两种情况分别写出优化后的TAC语句序列：

(1) 假设只有G、L、M在基本块后面还要被引用；

(2) 假设只有L在基本块后面还要被引用。

4. 对于图28所给出的流图，

1）为基本块 B2 构造 DAG 图。

2）假设 B4 出口处的活跃变量集合为空，求出 B3 出口处、B3 入口处以及 B2 出口处的活跃变量集合，即 LiveOut(B3)，LiveIn(B3) 以及 LiveOut(B2) 的值。

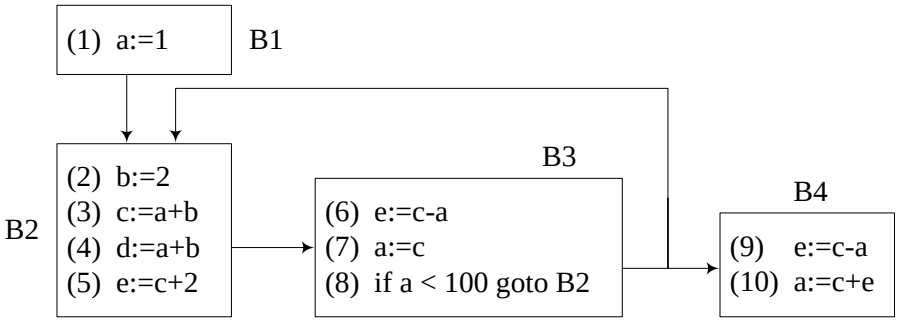

图28

5. 分别对图29和图30的流图：

(1) 求出流图中各结点n的必经结点集D(n)；

(2) 求出流图中的回边；

(3) 求出流图中的循环。

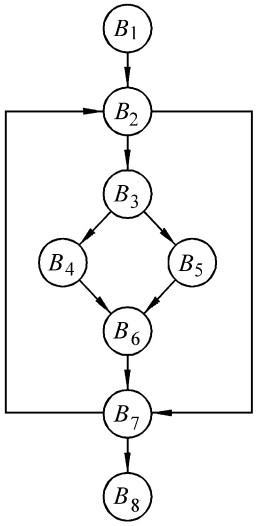

图29

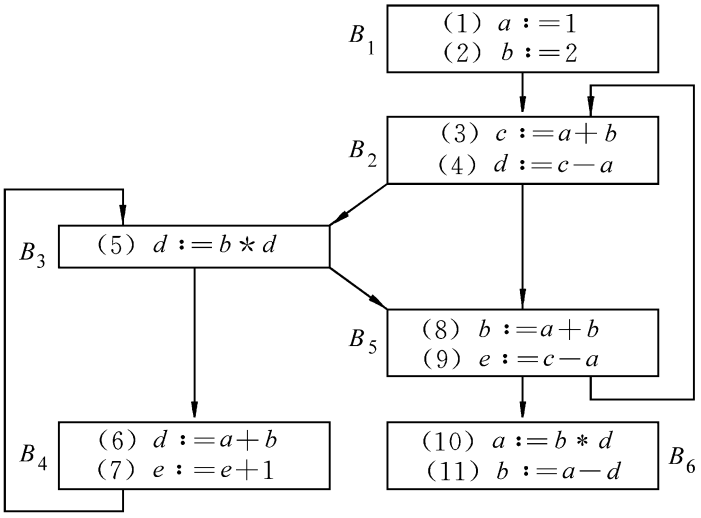

图30

6. 图31是包含 7 个基本块的流图，其中 B1 为入口基本块，B7 为出口基本块：

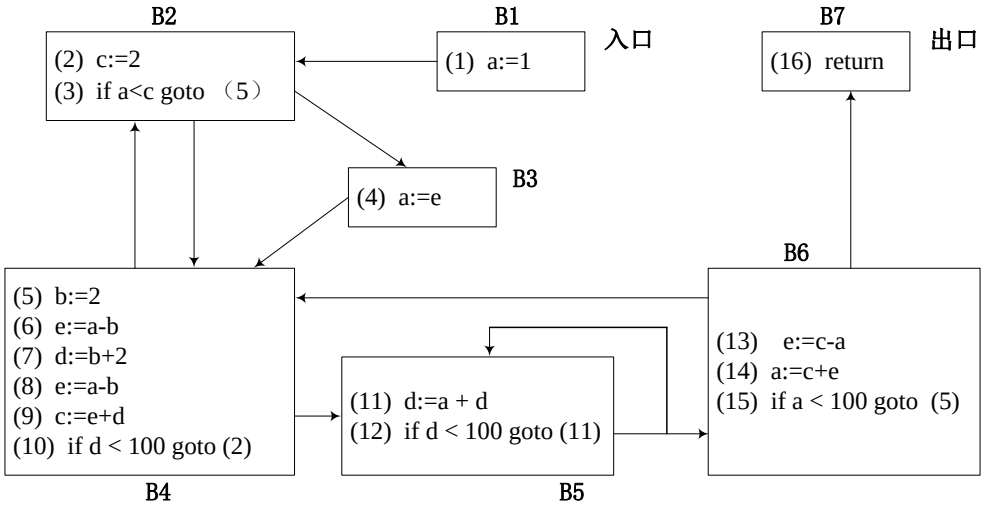

图31

1）指出在该流图中，基本块 B4 的支配结点（基本块）集合，始于 B4 的回边，以及基于该回边的自然循环中包含哪些基本块？

2）采用迭代求解数据流方程的方法对活跃变量信息进行分析。假设B7的 LiveOut信息为 ∅，迭代结束时的结果在下图所示表中给出。试填充该表的内容。

	LiveUse	DEF	LiveIn	LiveOut
B1				
B2				
B3				

B4				
B5				
B6				
B7				∅

3）对于该流图，根据采用迭代求解数据流方程对到达-定值（reaching definitions）数据流信息进行分析的方法。假设 B1 的 IN 信息为 ∅，迭代结束时的结果在下图所示表中给出。试填充该表的内容。

	GEN	**KILL**	**IN**	**OUT**
B1			∅	
B2				
B3				
B4				
B5				
B6				
B7				

4）指出该流图范围内，变量 a 在（11）的 UD 链。

5）指出该流图范围内，变量 c 在（2）的 DU 链。

7. 一个编译程序的代码生成需考虑哪些问题?

8. 图24右边的DAG图也是一棵表达式树。试对该表达式树的每个结点用Ershov数进行标记，并根据标记结果以及4.3节所介绍的算法，针对4.3节所假设的基于寄存器的简单机器，生成该表达式的目标代码。

9. 对于图30和图31中的流图，分别给出相应的寄存器相干图。若要保证图着色过程中不会出现将寄存器泄漏到内存中的情形，那么可供分配的物理寄存器的最小数目分别是多少？。

参考文献

[AGT89] Alfred V.Aho, Mahadevan Ganapathi, and Steven W.K.Tjiang. Code generation using tree matching and dynamic programming. In ACM transactions on Programming Languages and Systems, Vol.11, No.4, pp.491-516, 1989.

[PLG88] E.Pelegrí-Llopart and S.L.Graham. Optimal code generation for expression trees: an application burs theory. In Proceedings of ACM SIGPALN-SIGACT Sympisiun on Principles of Programming Languages, pp.294-308, 1998.

[SU70]Sethi, R., and Ullman, J. The generation of optimal code for arithmetic expressions. J.ACM

17,4(1970), 715-728.

[Aho07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Addison Wesley, 2007.