第七章　代码优化

代码优化的目的是获取高质量的目标代码，一个实际的编译器在翻译源程序过程中通常包含多遍优化。本章主要讨论与机器无关的面向中间代码的优化技术，首先介绍代码优化的定义及分类，然后通过一个实例来说明可以从哪些方面对中间代码进行优化。介绍了基本块的概念及基本块划分算法，以及基于基本块 DAG 表示的局部优化方法。中间代码优化的重点是循环优化，介绍了在控制流程图构建和回边检测基础上的循环体查找算法。确定循环体之后可以寻找循环不变计算、执行代码外提对循环进行优化。最后简单介绍了数据流信息相关的几个概念及全局优化问题。

7.1 代码优化概述

代码优化指在编译时刻为了改进目标程序的质量而进行的各项工作，包含代码优化模块的编译器又称为**优化编译器**。改进目标程序的质量包括提高目标程序的时间效率和空间效率。时间效率体现为生成的目标代码的运行时间的长短，空间效率体现为目标代码在运行时占用的内存空间的大小，代码优化的目的就是希望最终生成的目标代码运行时间尽可能短，占用的存储空间尽可能小。代码优化可以在中间代码生成之后针对中间代码进行，也可以在目标代码生成之后针对目标代码进行，如图 7-1 所示。

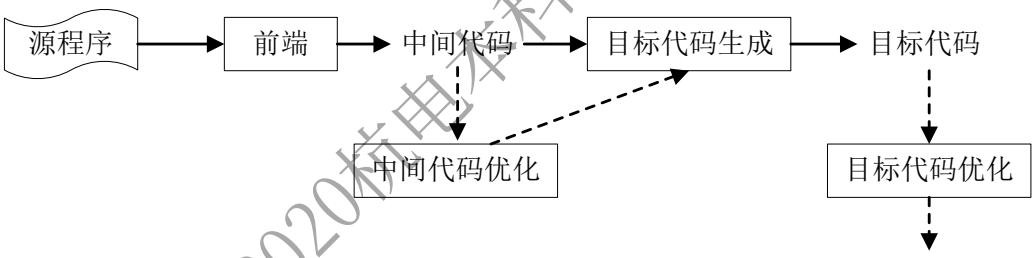


图 7-1 代码优化的阶段

代码优化是对代码的等价变换，即代码优化前和优化后的语义是一样的，即描述的算法是一样的，不能改变程序对给定输入的输出，也不能引起源程序原先不会出现的新错误。另外代码优化需要花费额外的成本，这个成本包括构造优化编译器增加的成本和编译源程序过程中增加的时间与空间成本。代码优化所花的额外成本应该能够从目标程序的运行中得到补偿，否则代码优化就是“得不偿失”的，没有意义。

代码优化涉及的范围非常广泛，编译过程各阶段、各环节可采用的优化技术非常多。根据优化是否和目标代码最终的运行机器相关，可以将代码优化分为两类：

（1）和目标机器相关的优化：即针对目标代码的优化。这类优化需要考虑目标机器的体系结构、CPU 及其指令系统等，优化内容包括寄存器优化、多处理器优化、特殊指令优化、无用指令消除等。

（2）与目标机器无关的优化：即针对中间代码的优化。由于中间语言的通用性，不同编译器可采用同一个中间语言作为源程序的中间表示，针对中间代码的与机器无关的优化更具有普遍意义，可以应用于各种物理平台上优化编译器的构造。本章重点讨论与机器无关的优化。

中间代码优化根据优化的代码范围又可以分为：局部优化、循环优化和全局优化，见图7-2。

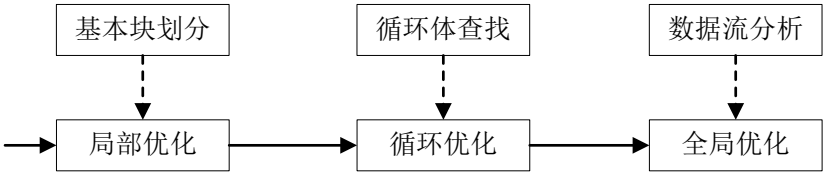


图 7-2 中间代码优化的分类

（1）局部优化：面向程序基本块。基本块是程序运行时一个不可分割的代码序列。要做局部优化先要将中间代码划分为一个一个的基本块。

（2）循环优化：面向循环体。循环体是程序的一次运行过程中可能被反复执行到的基本块集合。要做循环优化首先需要在程序流图的基础上，分析程序的控制流程，识别循环体。

（3）全局优化：面向整个程序。需要进行数据流信息的收集，包括到达-定值、活跃变量与可用表达式等反映程序中变量值的获得和使用情况的数据流信息。

下面通过一个例子来说明中间代码优化的常见内容。

【例 7-1】考察如下源程序的片段：

```
P := 0
for I := 1 to 20 do
        P := P + A[I] * B[I] ;
```

假设每个数组元素占 4 个字节，将以上源代码翻译为三地址代码，代码序列如图 7-3 所示。代码共有 12 条，其中(1)-(2)构成独立的一部分 B_1，(3)-(12)构成另一部分 B_2，B_2 是一个循环体，有向边表示控制流程。对这段中间代码可作如下优化操作。

（1）删除多余运算

在表达式或者相关语句中可能出现完全相同的子表达式，称为**公共子表达式**。公共子表达式的每次出现其计算结果都是相同的，每次都去做计算是没有必要的。图 7-3 代码段中，“4 * I”在(3)、(6) 中都出现了，它就是一个公共子表达式。为优化代码可以将(6)等价变换为“T_4 := T_1”。

（2）代码外提

代码外提主要针对循环体。由于循环体在程序运行过程中可能被反复执行，因此减少循环体的代码数量可以提升整个程序的执行效率。有一类所谓的循环不变运算（即计算结果独立于循环次数的运算）没有必要放在循环体里面每次都去执行，可以提到循环外面，做一次运算就可以了。例中代码(4)和(7)都是循环不变运算，可以从循环体 B_2 中提到 B_1 中去执行。

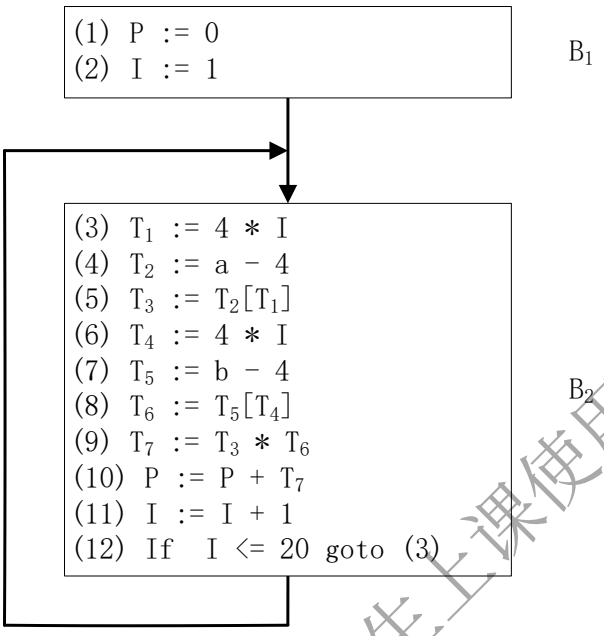


图 7-3 例 7-1 程序的中间代码

（3）强度削弱

对于 CPU 来说，各种运算的强度是不一样的。例如一般的通用 CPU 做乘法运算需要先将其转化为加法，因此乘法的运算强度比加法的运算强度要高。在代码中如果有选择应该尽量采用强度低的运算。例中 T_1 与 I 是线性关系，每做一次循环，T_1 递增 4 。可以将(3)外提，用一次乘法运算计算 T_1 的初值，然后在(12)前增加一条代码“$T_1 := T_1 + 4$”，用加法运算替换乘法运算。

（4）变换循环控制条件

代码(12)中的“I <= 20”是循环控制条件，I 与 T_1 之间是一种线性关系（$T_1 :=$ 4 * I），可以将循环控制条件变换为“T_1 <= 80”。虽然这步变换本身没有对代码进行优化，但是可以为后续的其它代码优化作准备。

（5）合并已知量与复写传播

T_1 的初值在编译时就可以计算出来，故可将(3)直接写为“$T_1 := 4$”，这类操作称为合并已知量。T_4 等于 T_1，于是(8)可以改写为“$T_6 := T_5 [T_1]$”，这类变换称为复写传播。

（6）删除无用代码

经过前面几步的优化后，发现 T_4 在代码段中没有被引用，I 作为控制条件的作用已被 T_1 替代，若同时已知 T_4 和 I 在程序的后面未被引用，那么 T_4 和 I 的相关赋值语句(2)、(6)和(11)可以删除。另外有一些在执行过程中永远无法被执行到的代码，称为死代码。删除死代码不会对程序的运行结果造成影响。

图 7-3 代码段经过优化之后变换为如图 7-4 所示的代码段。优化之后的代码从 12 条减少为 10 条，更有意义的是循环体的代码由 10 条减少为 6 条，这将大大提升代码的执行效率。

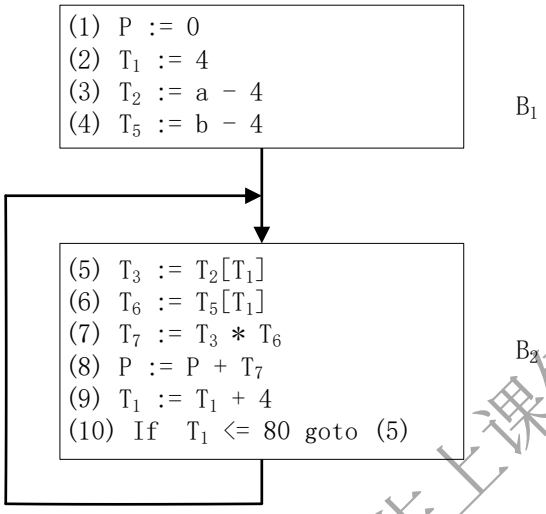


图 7-4 优化后的中间代码

接下来几节将讨论代码优化的具体实现技术。

7.2 基本块与局部优化

基本块是三地址代码形式的程序中的一个连续的代码序列，执行的时候从一个称为**入口语句**的代码进入，从一个称为**出口语句**的代码退出。每一个基本块的入口语句和出口语句都是唯一的。基本块在程序执行的时候是一个不可分割的整体，基本块中的语句要么同时被执行，要么都不执行。对于一个给定的三地址代码形式的程序，可以把它划分为一个个的基本块，针对基本块的优化称为**局部优化**。

基本块划分包括两个步骤，第一个步骤是查找程序中所有的入口语句，第二个步骤是确定每个入口语句对应基本块的代码。

程序中满足以下条件之一的都是入口语句：

（1）程序的第一条代码；

（2）转移语句的目标语句；

（3）紧跟在条件转移语句后面的语句。

查找出程序中所有的入口语句之后，可以按如下步骤为每一个入口语句构造基本块：

（1）入口语句是基本块的第一条语句；

（2）从该入口语句出发，顺次往下查找，到达下一个入口语句（不包括这一入口语句），或到达一个转移语句（包括该转移语句），或到达一个停语句（包括该停语句）之间的代码序列构成基本块。

凡是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，可以删除。删除无用代码本身就是对代码进行优化。

考察图 7-3 的例子，(1)是程序第一条代码，是入口语句，(3)是(12)这条转移语句的目标语句，也是入口语句。从(1)出发，顺次往下查找，到达(3)这条入口语句，不含(3)，(1)-(2)构成第一个基本块 B_1，从(3)出发往下查找，到达转移语句(12)，(3)-(12)构成第二个基本块 B_2。

再看一个例子。

【例 7-2】考察如下程序，把它划分为基本块。

```
(1) read x
(2) read y
(3) r := x mod y
(4) if r = 0 goto (8)
(5) x := y
(6) y := r
(7) goto (3)
(8) write y
(9) halt
```

解：入口语句有 4 条：

（1）程序第一条语句(1)；

（2）转移语句的目标语句(3)；

（3）条件转移语句的下一条语句(5)；

（4）转移语句的目标语句(8)。

根据基本块划分方法，(1)-(2)构成第一个基本块 B_1，(3)-(4)构成第二个基本块 B_2，(5)-(7)构成第三个基本块 B_3，(8)-(9)构成第四个基本块 B_4。

下面讨论基于基本块的 DAG 表示来实现基本块优化的方法。DAG（Directed Acyclic Graph）是有向图的一种，DAG 不允许存在回路，因此被称为有向无环图。可以通过将基本块转换为一个 DAG 表示，并在此基础上重写代码来实现基本块的优化。

基本块的 DAG 表示中涉及两种结点：

（1）叶结点：只有出边没有入边的结点。叶结点标记为变量名字或常数，作为叶结点的变量名字代表变量的当前值，通常加上脚标 0；

（2）内部结点：既有出边又有入边的结点（也可以是只有入边的结点）。内部结点标记为运算符号，代表此运算符号作用于其子结点计算出来的值。所谓子结点是指它的入边关联的结点，也就是它的前驱结点。

每个结点都有一个附加标识符表，可附加一个或多个标识符，表示这些标识符具有该结点所代表的值。

简单起见，只考虑如下 3 种形式的三地址代码：

（1） x := y op z

（2） x := op y

（3） x := y

在构造 DAG 过程中，需用到一个函数 node（id），其中 id 是一个名字。node（id）返回最新建立的与 id 联系的结点。

构造 DAG 的方法是依次考察每一条三地址代码，并根据代码的形式和内容执行一定的操作。考察每条三地址代码之前，如果 node（y）和/或 node（z）没有定义，则首先建立标记为 y 和 z 的结点。对于 3 种不同的三地址代码分别执行如下操作：

（1）对于 x := y op z，寻找是否有一个标记为 op 的结点，它的左子结点为 node（y），右子结点为 node（z）。如果有，在此结点的附加标识符表中增加 x，否则，建立一个这样的标记为 op 的结点，并在此结点的附加标识符表中增加 x。

（2）对于 x := op y，寻找是否有一个标记为 op 的结点，它的唯一子结点为 node（y），如果有，在此结点的附加标识符表中增加 x，否则，建立一个这样的标记为 op 的结点，并在此结点的附加标识符表中增加 x。

（3）对于 x := y，在 node（y）的附加标识符表中增加 x。

要注意的是，以上步骤中在增加标识符 x 之前要删除其它结点上附加标识符中的 x（如果存在的话）。特殊地对于 3 种三地址代码，如果 x 当前可计算得到一个常数，在一个该常数的结点中标记 x，或建立一个标记为 x 的常数结点。

【例 7-3】考察如下三地址代码并构建 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

依次考察每一条代码：

（1）对于 T_0 := 3.14，构建一个常数结点 3.14，并在此结点的附加标识符表中添加 T_0，见图 7-5(a)。

（2）对于 T_1 := 2 * T_0，由于可直接计算出 T_1 为 6.28，于是构建一个常数结点 6.28，并在此结点的附加标识符表中添加 T_1，见图 7-5(b)。

（3）对于 T_2 := R + r，由于 R 和 r 对应的结点还没有且它们都是变量，于是分别构建标记为 R_0 和 r_0 的结点，并构建一个标记为+的结点，其左子结点是 R_0，右子结点是 r_0，并在此结点的附加标识符表中添加 T_2，见图 7-5(c)。

（4）对于 A := T_1 * T_2，调用 node (T_1)和 node (T_2)返回结点 6.28 和+，构建一个标记为*的结点，其左子结点是 6.28，右子结点是+，并在此结点的附加标识符表中添加 A，见图 7-5(d)。

（5）对于 B := A，调用 node (A)返回结点*，在这个结点的附加标识符表中添加 B，见图 7-5(e)。

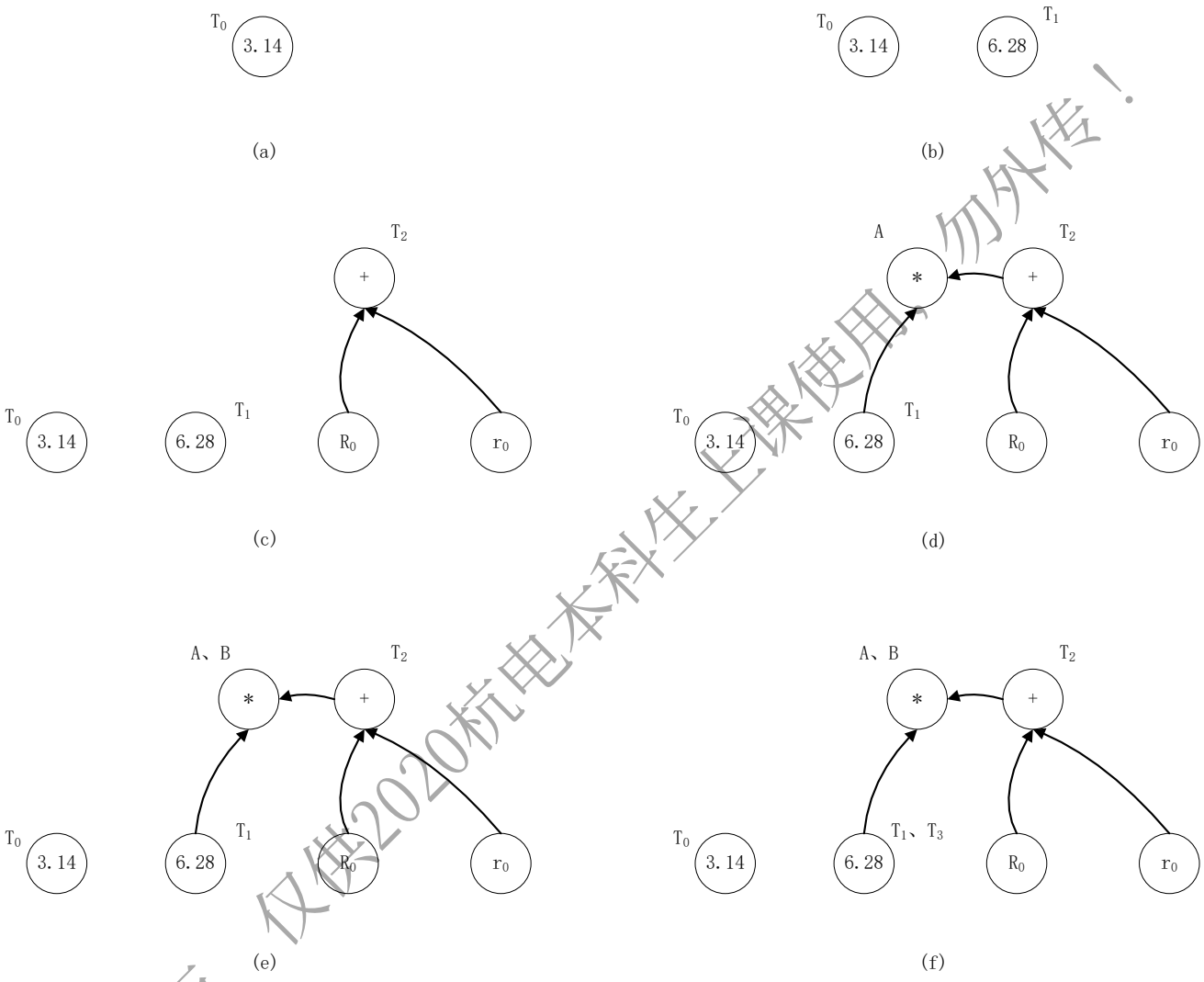
（6）对于 T_3 := 2 * T_0，由于可直接计算出 T_3 为 6.28，于是在常数结点 6.28 的附加标识符表中添加 T_3，见图 7-5(f)。

（7）对于 T_4 := R + r，由于存在一个标记为+的结点，并且它的左子结点是 R_0，右子结点是 r_0，于是在这个结点的附加标识符表中添加 T_4，见图 7-5(g)。

（8）对于 T_5 := T_3 * T_4，由于存在一个标记为*的结点，并且它的左子结点是 node (T_3)，右子结点是 node (T_4)，于是在这个结点的附加标识符表中添加 T_5，见图 7-5(h)。

（9）对于 T_6 := R - r，由于不存在一个标记为-的并且其左子结点是 R_0，右子结点是 r_0 的结点，于是构建一个这样的结点，同时在这个结点的附加标识符表中添加 T_6，见图 7-5(i)。

（10）对于 B := T_5 * T_6，由于不存在一个标记为*的并且它的左子结点是 node (T_5)，右子结点是 node (T_6)的结点，于是构建一个这样的结点，同时在这个结点的附加标识符表中添加 B。在添加 B 之前还要删除第一个*结点附加标识符表中的 B，见图 7-5(j)。



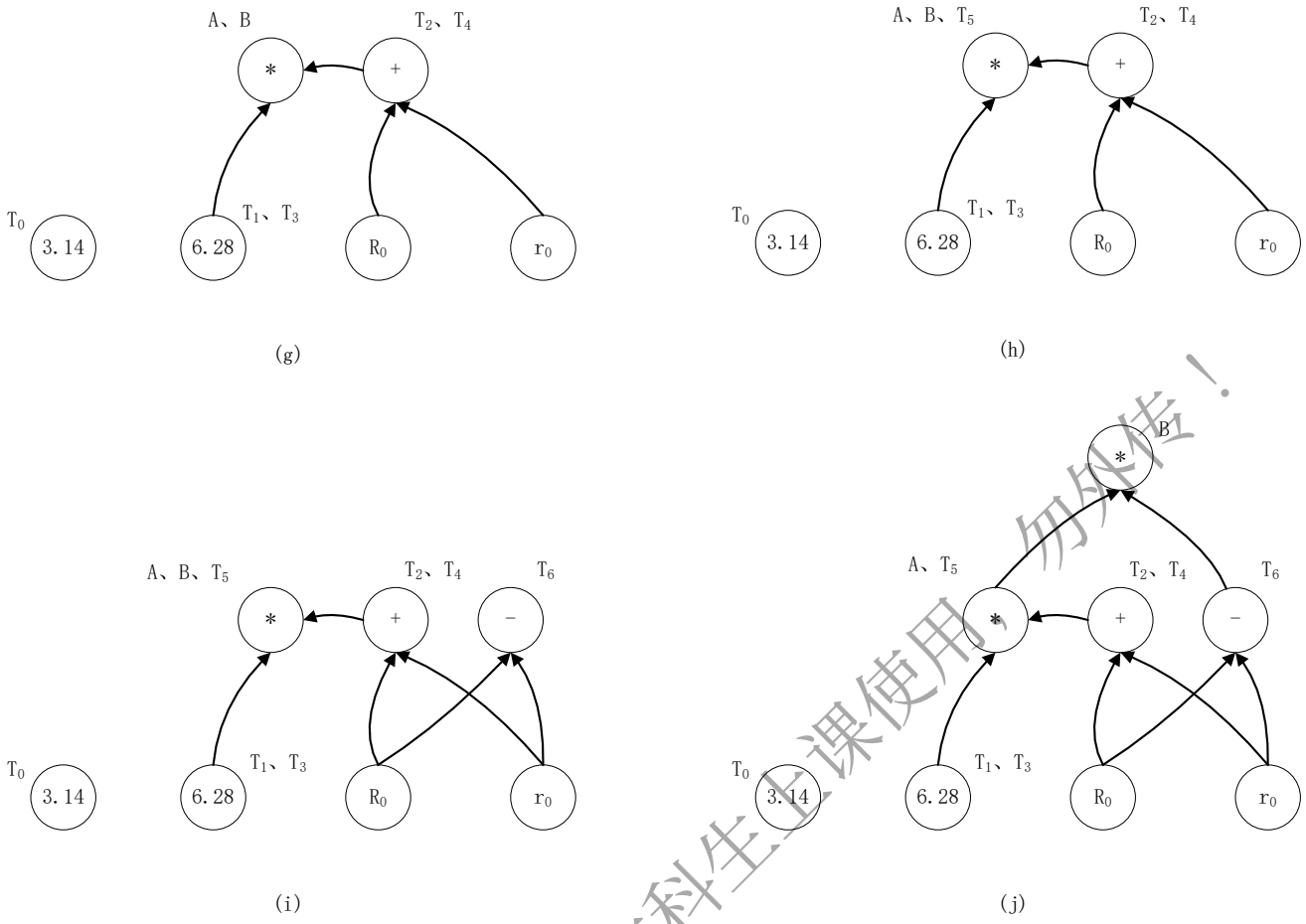


图 7-5 为例 7-3 构造 DAG

在构造 DAG 的过程中已经对代码作了优化。对于 3 种三地址代码，如果 x 计算出一个常数，并不生成一个计算该常数的内部结点，而是生成一个叶结点，或者在一个已有常数结点的附加标识符表中添加 x，这个步骤起到了合并了已知量的作用。在考察一条代码过程中，如果有了一个代表计算结果的结点，并不另外构建结点，只是把这个被赋值的变量标识符附加到这个结点上，这样可以删除多余运算。在一个结点的附加标识符表中增加标识符 x 之前，要删除其它结点上附加标识符中的 x，这个步骤相当于删除了无用赋值。因为该变量在被引用前又被重新赋值了，之前的赋值就是无意义的。

最后，按照构造 DAG 的顺序重写代码可得优化之后的代码。

例 7-3 的三地址代码重写之后得到如下代码：

(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 （7-1）

(6) A := 6.28 * T_2

(7) T_5 := A

(8) T_6 := R - r

(9) B := A * T_6

对于原代码中的(2)和(6)合并了已知量，删除了(5)这个无用赋值，(3)和(7)中存在的公共子表达式只计算了一次。

代码中的 T_0、T_1、...、T_6 是临时变量，临时变量一般具有局部性，如果已知 T_0、T_1、...、T_6 在基本块后面不被引用，则代码可进一步优化为：

(1) S_1 := R + r

(2) A := 6.28 * S_1 （7-2）

(3) S_2 := R - r

(4) B := A * S_2

将基本块转化为 DAG 表示，代价还是比较大的，但这个代价是值得的。除了可以基于 DAG 对基本块作局部优化，还可以从 DAG 中获得其它一些优化所需信息。这些信息包括：

（1）在基本块外被赋值，并在基本块内被引用的所有标识符，就是标记为变量名字的叶结点上标记的那些标识符；

（2）在基本块内被赋值，可以在基本块外被引用的所有标识符，就是 DAG 中各结点附加标识符表中的标识符。

以上信息对于全局数据流分析具有重要作用。

7.3 控制流分析与循环优化

循环是程序中常见的控制结构。所谓**循环**就是在程序的一次执行过程中可能被反复执行的代码序列。因为可能被反复执行，因此针对循环体进行优化对于提升整个目标代码的运行效率意义重大。要作循环优化，首先要分析整个程序的控制流程并识别循环体。我们可以使用程序的控制流程图来描述程序的控制流程，并在此基础上讨论循环的定义和循环的查找。

一个三地址代码形式的程序的**控制流程图**（简称**流图**）是一个有向图，可以用一个三元组 G =（N，E，n_0）来表示。其中，N 是流图的结点集合，流图中的结点是程序基本块；n_0 是 N 中的元素，称为**首结点**，该结点包含程序的第一条语句，n_0 到流图中任何结点都有通路；E 是有向边集合，有向边代表程序的流程，有向边按如下规则构造：

当以下条件之一成立时，从结点 i 到结点 j 画一条有向边：

（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或停语句；

（2）基本块 i 的出口语句是转移语句，并且转移目标是 j 的入口语句。

考察例 7-2 中的程序，程序被划分为 B_1、B_2、B_3、B_4 共 4 个基本块，其中 B_2 紧接在 B_1 之后，于是从 B_1 到 B_2 画一条有向边；B_3 紧接在 B_2 之后，从 B_2 到 B_3 画一条有向边；B_2 的出口语句是转移语句，转移目标是 B_4 的入口语句，故从 B_2 到 B_4 画一条有向边；B_3 的出口语句是转移语句，转移目标是 B_2 的入口语句，故从 B_3 到 B_2 画一条有向边。例 7-2 程序的控制流程图见图 7-6。

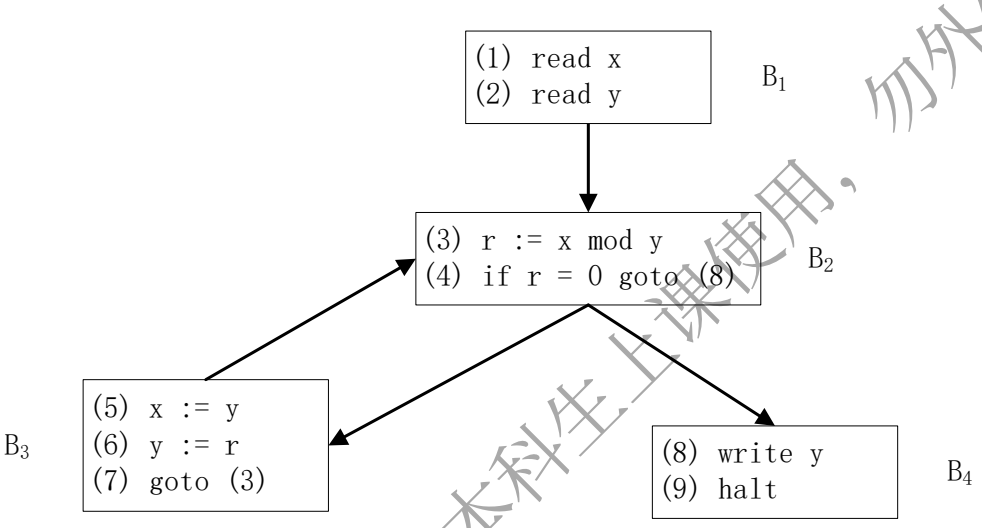


图 7-6 例 7-2 的控制流程图

循环由流图中的结点序列构成，这些结点是强连通的，有且仅有一个称之为入口结点的结点。要在流图基础上查找循环体，需要先定义几个概念。

有流图中的结点 d 和结点 n，如果从流图的首结点 n_0 出发，每条到达 n 的路径都要经过 d，则称 d 是 n 的**必经结点**，记为 d DOM n。特殊地，每个结点是它自身的必经结点，首结点 n_0 是所有结点的必经结点。

【例 7-4】考虑如图 7-7 所示的流图，其中 1 是首结点。令 n 是流图中的任一结点，求 n 的必经结点集 DOM(n)，即求 n 是哪些结点的必经结点。

解：

DOM(1) = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

DOM(2) = { 2 }

DOM(3) = { 3, 4, 5, 6, 7, 8, 9, 10 }

DOM(4) = { 4, 5, 6, 7, 8, 9, 10 }

DOM(5) = { 5 }

DOM(6) = { 6 }

DOM(7) = { 7, 8, 9, 10 }

DOM(8) = { 8, 9, 10 }

DOM(9) = { 9 }

DOM(10) = { 10 }

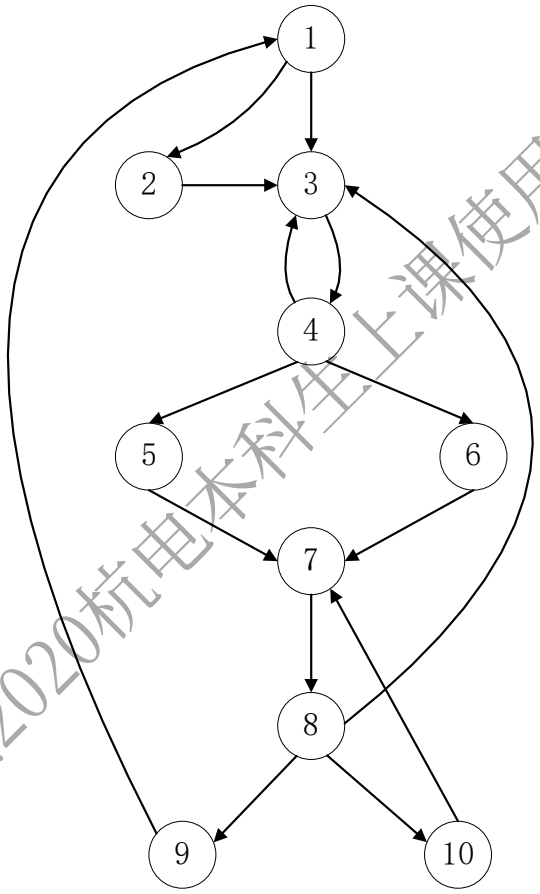


图 7-7 一个流图的例子

下面介绍另一个概念回边。假设 a→b 是流图中的一条有向边，如果同时有 b DOM a，则称 a→b 是流图的一条**回边**。对于一个流图，求出了每个结点的必经结点集，就可以求出所有的回边。

考察图 7-7 中的流图，由于 4 DOM 7，故 7→4 是回边；7 DOM 10，故 10→7 是回边；3 DOM 4，故 4→3 是回边；3 DOM 8，故 8→3 是回边；1 DOM 9，故 9→1 是回边。该流图共有 5 条回边。

根据回边的定义，可知一条回边对应一个循环。假设 a→b 是流图中的一条回边，则 b 是循环的入口，a 是循环的出口之一，如果 a 等于 b，该循环只有一个结点。如果 a 不等于 b，

可按如下规则确定循环中的其它结点：求 a 的前驱结点，以及前驱结点的前驱结点，直到回到 b，在这个过程中求得的结点都是该循环中的结点。

对于例 7-4，它的 5 条回边对应的循环体分别为：

（1）7→4，循环体为 { 4, 5, 6, 7, 8, 10 }；

（2）10→7，循环体为{ 7, 8,10 }；

（3）4→3，循环体为{ 3, 4, 5, 6, 7, 8, 10 }；

（4）8→3，循环体为{ 3, 4, 5, 6, 7, 8, 10 } ；

（5）9→1，循环体为{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 } 。

分析程序的结构，可知整个程序体就是一个大的循环（9→1），记为 X_1。X_1 包含一个小循环（4→3 和 8→3），记为 X_2，这个循环的入口结点是 3，4 和 8 是它的两个出口结点。X_2 又包含一个小循环（7→4），记为 X_3。X_3 包含一个更小的循环（10→7），记为 X_4。程序是一个 4 层循环的嵌套结构。

代码外提和删除归纳变量是常见的循环优化操作，下面介绍具体的实现方法。

代码外提，即将计算结果独立于循环次数的所谓循环不变计算放到循环的前面执行。这样可以减少循环体中代码的数量，提高整个代码的运行效率。代码外提的第一步是要寻找循环不变计算。令 X 是一个循环体，寻找 X 中循环不变计算的算法如下：

（1）考察 X 中所有基本块的每条三地址代码，如果它的运算对象或者是常数，或者对其赋值的语句都在 X 之外，则将它标记为循环不变计算；

（2）重复（3），直到没有新的三地址代码标记为循环不变计算；

（3）考察 X 中未被标记为循环不变计算的三地址代码，如果它的运算对象或者是常数，或者对其赋值的语句都在 X 之外，或者只有一个对其赋值的语句且该赋值语句被标记为循环不变计算，则将该三地址代码标记为循环不变计算。

要实现代码外提，可以在循环的入口结点前面建立一个新结点，并将所有可外提的循环不变计算放置在这个新结点中，这个结点称为**前置结点**。前置结点以循环的入口结点为其唯一的后继结点，流图中原来进入入口结点的有向边，全部改为进入前置结点，如图 7-8 所示。

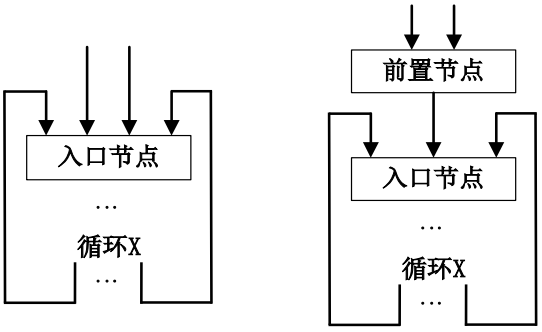


图 7-8 代码外提的流图

并不是所有的循环不变计算都能外提。可外提的循环不变计算 s：x := y + z 需要满足三个条件：

（1）含 s 的基本块是循环中所有出口结点的必经结点，出口结点是指存在不在循环体中的后继结点的结点。见图 7-9，B_2、B_3、B_4 构成一个循环，B_4 是出口结点，B_3 中的代码 i := 2 是循环不变计算，但是 B_3 不是 B_4 的必经结点，将 i := 2 外提的话，对于原先不经过 B_3 的执行过程将产生不同的计算结果。

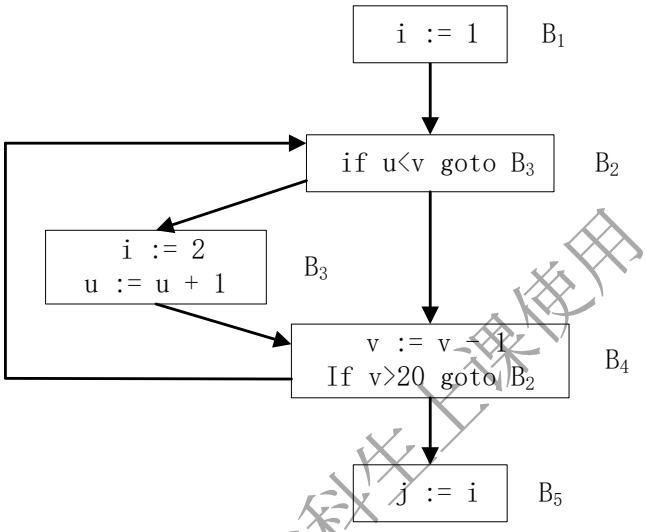


图 7-9 代码外提条件（1）

（2）循环中没有其它语句对 x 赋值。见图 7-10，这个流图与图 7-9 类似，但在结点 B_2 中又增加了一个对 i 赋值的代码 i := 3，这也是循环不变计算，如果把 i := 3 外提，对于原先经过 B_3 的执行过程将产生不同的计算结果。

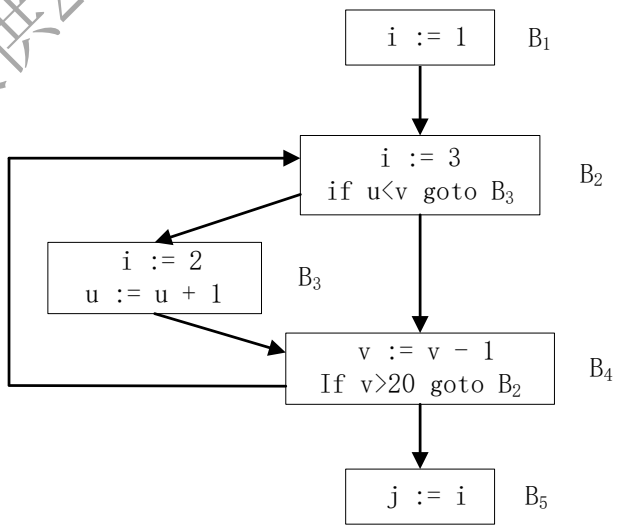


图 7-10 代码外提条件（2）

（3）循环中对 x 的引用，只能是引用 s 对 x 的赋值。见图 7-11，B_4 中的代码 k := i 对 i 的引用可以来自 B_3（i 的值为 2），也可能来自 B_1（i 的值为 1）。若将 i := 2 外提，则 k := i 中 i 的值将一直是 2。

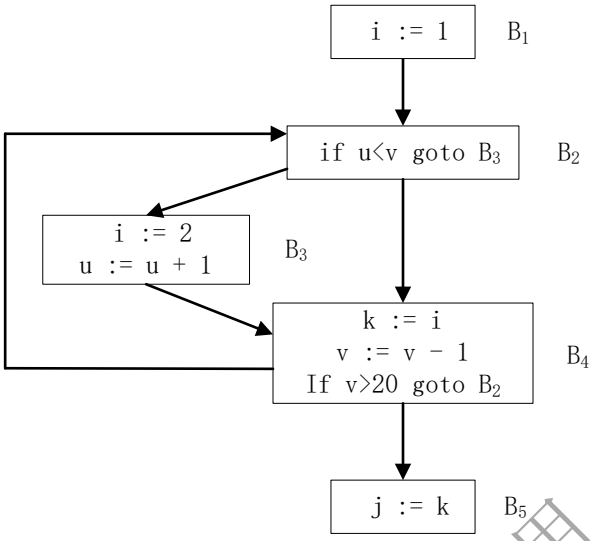


图 7-11 代码外提条件（3）

下面简单介绍删除归纳变量。如果循环中对变量 i 只有唯一的形如 i := i ± c 的赋值，且 c 是循环不变量，则称 i 为循环的**基本归纳变量**。如果变量 j 和基本归纳变量 i 是线性关系，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 为**归纳变量**，并且称它与 i 是同族的。一个基本归纳变量可以用来控制循环，也可以用来计算与它同族的归纳变量。循环控制条件中的基本归纳变量可以用一个与它同族的归纳变量来替换。如果基本归纳变量在循环内没有除作为循环控制条件的其它引用，在循环外的后续的基本块中也没有被引用，则它被替换之后就可以将对它的递归赋值作为无用赋值删除，参见例 7-1 中的强度削弱与变换控制条件。

7.4 数据流分析与全局优化

要从整个程序的视角考虑代码优化问题的话，必须收集相关的数据流信息，包括到达-定值、引用-定值链、定值-引用链、变量活跃信息、变量下次引用信息等。

若标号为 i 的语句是对变量 x 的赋值语句，称 x 在定值**点** i 被**定值**。若标号为 j 的语句引用了 x，且在 i 和 j 之间没有对 x 重新定值，则称定值点 i 可以**到达** j。到达-定值的示例见图 7-12。由于在 j 中引用了 x，也称 j 是 i 中变量 x 的**下次引用信息**，同时称 x 在点 i 是**活跃的**（因为在后面会被引用）。

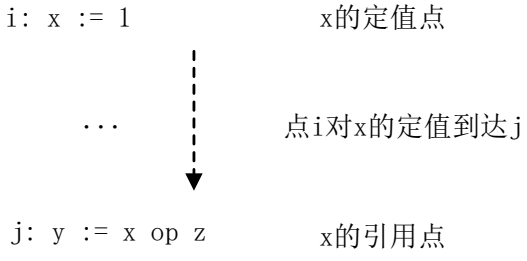


图 7-12 到达-定值示例

若在程序的某点 j 引用了变量 x，则把能到达 j 的 x 的所有定值点的集合称为 x 在引用点 j 的**引用-定值链**，简称 *ud* 链。与之对应的是**定值-引用链**。对一个变量 x 在某点 i 的定值，可以计算该定值能到达的对 x 的所有引用点。这些引用点的集合称为该定值点的定值-引用链，简称 *du* 链。

以上这些数据流信息对于实现各类优化操作是必不可少的。例如，在例 7-3 的基本块优化中，只有已知各临时变量的下次引用信息和活跃信息才可以将（7-1）的 9 条代码进一步优化为（7-2）的 4 条代码。在进行循环优化时，为了求出循环中的所有循环不变计算，需要知道各变量引用点的 *ud* 链信息，前面给出的各种循环优化的算法都是在假设 *ud* 链已知的前提下进行的。无论是在基本块优化还是循环优化中，都可能引起某些变量的定值在该基本块或者该循环内不会被引用；只要这些变量在基本块或者循环出口之后也是不活跃的，则这些变量在基本块或者循环内的定值就是可以删除的无用赋值。因此 *du* 链的获取和活跃变量的分析对于删除无用赋值是很重要的。

数据流信息可以通过建立和解方程来计算获得，这些方程联系程序不同点的信息，典型的方程形式为：

$$out[s] = (in[s] - kill[s]) \cup gen[s]$$

该方程的意思是，当控制流通过一个语句时，在语句末尾的信息是进入这个语句中的信息减去本语句注销的信息并加上产生的新信息。这样的方程称之为数据流方程。

如何建立与求解各类数据流方程以获取各类数据流信息在这里不详述。

7.5 小结

实用的编译器都是优化编译器，未做优化的目标代码将带来极大的运行时刻开销。代码优化与编译各阶段均相关，生成高质量的目标代码是编译器设计的主要目标。代码优化主要分为两类：一类是针对中间代码的，在中间代码生成之后进行；另一类是针对目标代码的，在目标代码生成之后进行。针对目标代码的优化与目标代码运行的物理平台密切相关，每个物理平台都有自己的体系结构、CPU 型号及配套的指令系统。针对目标代码的优化技术通用性较弱。针对中间代码的优化又分为面向基本块的局部优化、面向循环体的循环优化、面向整个程序的全局优化。基本块是一个具有唯一入口语句和唯一出口语句的连续的三地址代码序列，是程序执行的基本单位。三地址代码形式的程序可以划分为一个个的基本块，每个基本块可以表示为一个 DAG，按照构造 DAG 的顺序重写代码可以对基本块进行优化，这就是局部优化。中间代码优化中最重要的是循环优化，循环优化的基础是对循环体的查找，而查找循环体需要分析整个程序的控制流程。通常采用控制流程图（流图）来描述程序的控制流程，流图是一个有向图，节点是基本块，有向边代表程序的执行流程。通过定义必经结点和回边，可以基于流图自动检测程序中的所有循环体，然后执行代码外提等循环优化操作。全局优化需要利用多种数据流信息，包括引用-定值链、定值-引用链、变量的活跃信息与下次引用信息等，要获取这些数据流信息需要建立和求解数据流方程。本章的重点是理解代码优化

的意义、分类与内容；掌握基本块的划分算法，以及基本块 DAG 的构造与基本块优化方法；掌握控制流程图构造方法及循环体查找和循环优化方法。

习题

7.1 设有如下中间代码：

```
        i =1
        j =1
L1:   t1 = 10 * i
L2:   t2 = t1 + j
        t3 = 8 * t2
        t4 = t3 - 88
        j = j + 1
        if j<=10 goto L2
        i = i +1
        if i <= 10 goto L1
        goto L1
        i = 1
L3:  t5 = i - 1
        t6 = 88 * t5
        a[t6] = 1.0
        i = i + 1
        if   i <= 10 goto L3
```

(1) 将上述中间代码划分成基本块；

(2) 试画出该中间代码的控制流程图，并指出所有的循环；

(3) 试对每个循环做代码外提和强度削弱优化。

7.2 对于基本块 B：

$t_0 = 2$

$t_1 = 8 / t_0$

$t_2 = T - C$

$t_3 = T + C$

$R = t_0 / t_3$

$H = R$

$t_4 = 8 / t_1$

$t_5 = T + C$

$t_6 = t_4 / t_5$

$H = t_6 / t_2$

(1) 应用 DAG 进行优化；

(2) 假设只有 R 和 H 在基本块出口是活跃的，写出优化后的四元式序列。