第11章 优化

1：所谓优化

- 一般是指为提高目标程序的质量而进行的各项工作。

➢即对程序或中间代码进行各种等价变换，使得从变换后的程序出发，能生成更有效的目标代码。

➢这里所说的质量，通常是指目标程序所占的存储空间(即程序的静态长度)的大小和运行目标程序所需要的时间(即程序的动态长度)的多少。

➢优化之目的在于，既要设法缩小存储空间，又要尽量提高运行速度，而且常常偏重于提高运行速度。

第11章 优化

2：优化涉及面很广

- 从优化与机器的关系出发，可将优化分为与机器无关的优化和与机器相关的优化。

➢ 与机器无关的优化可在源程序或中间语言程序一级上进行。

➢ 这类优化主要包括：

①常数合并，

②公共表达式的消除，

③循环中不变式的外提，

④运算强度的削弱，等等。

第11章 优化

3：从优化与源程序的关系而言

- 可把优化分为局部优化和全局优化。
  - 局部优化通常是在只有一个入口(程序段的第一条代码)和一个出口(程序段的最后一条代码)的基本块上的优化。
  - 因为只存在一个入口和一个出口，又是线性的；
  - 即逐条顺序执行的，不存在转入转出、分叉汇合等问题；
  - 处理起来就比较简单，开销也较少，但优化效果稍差。
- 全局优化指的是在非线性程序块上的优化。
  - 程序块是非线性的；
  - 因此需要分析比基本块更大的程序块乃至整个源程序的控制流程，需要考虑较多的因素。
  - 这样的优化比较复杂，开销也较大，但效果较好。

11.1　基本块及其求法

1：基本块

- 基本块：
  - 一个入口（第一语句）
  - 一个出口（最后一语句）
- 入口语句集：
  - 程序的第一个语句号；
  - goto语句的下一条语句号；
  - goto语句转到的语句号。
- 基本块划分：
- 入口语句集排序后每一入口语句到下一入口语句号的前一语句。
- 最后一基本块为最后入口语句号到最后语句。

11.1　基本块及其求法

2：把四元式还原成较直观的形式

- 为便于讨论把四元式还原成较直观的形式，我们约定：

(op　　B　C　A)→A:=B op C;

(jump$_\theta$　B　C　L)→if (B θ C) goto　L;

(jump　—　—　L)→goto L;

11.1 基本块及其求法

3：例如，考虑下述四元式序列：

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

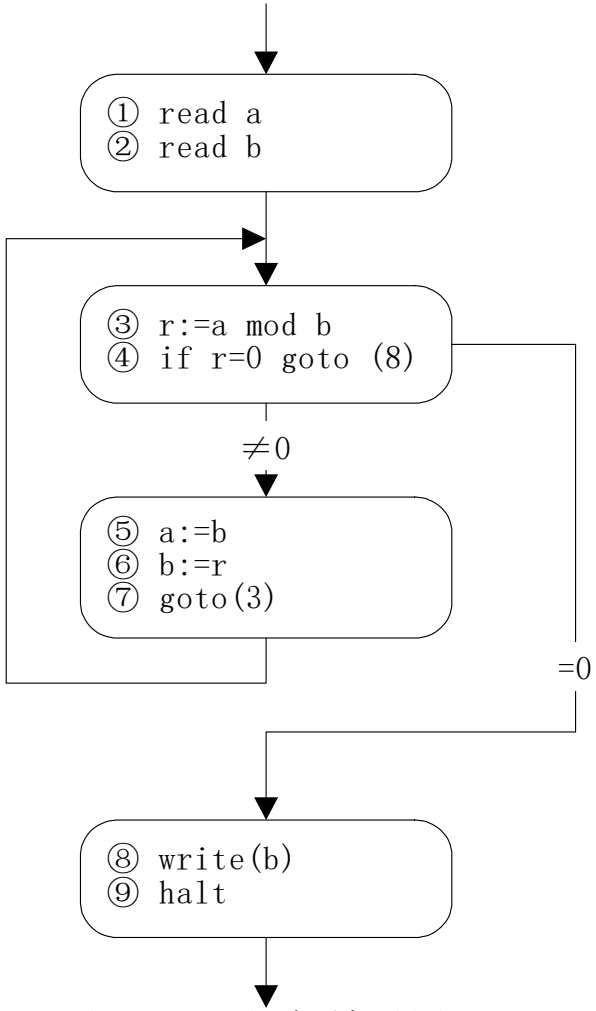


图11.1 基本块举例

- ①、③、⑤、⑧是人口语句；
- ①~②、③~④、⑤~⑦、⑧~⑨分别构成基本块。
- 如图11.1所示。

11.2 优化举例

1:中间代码

- 例如，考虑源程序段

Begin
integer i，j;
array A,B[0..99,0..99];
…
for i:=0 step 1 until 99 do
A[i,2*j]:=B[i,2*j]+5*3.1416;
end

- 其中间语言程序为：

① i:=0;
② T_1:=2*j;
③ T_2:=100*i+T_1;
④ T_3:=5*3.1416;
⑤ R_1:=B[T_2];
⑥ T_4:=2*j;
⑦ T_5:=100*i+T_4;
⑧ A[T_5]:=R_1+T_3;
⑨ i:=i+1;
⑩ if i<100 then goto ②

11.2 优化举例

2:中间代码基本块

- 中间语言程序为:

① i:=1;

② T_1:=2*j;

③ T_2:=100*i+T_1;

④ T_3:=5*3.1416;

⑤ R_1:=B[T_2];

⑥ T_4:=2*j;

⑦ T_5:=100*i+T_4;

⑧ A[T_5]:=R_1+T_3;

⑨ i:=i+1;

⑩ if i≤100 then goto ②

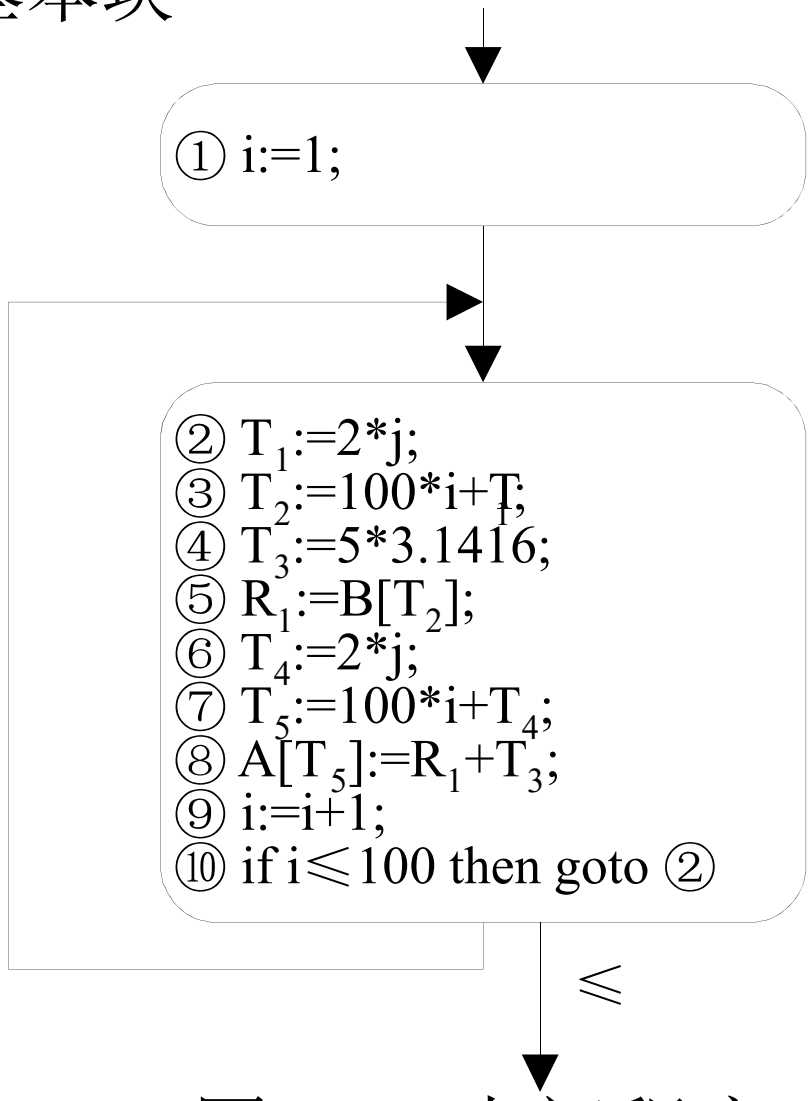


图11.2 中间程序1

11.2 优化举例

3:常量合并

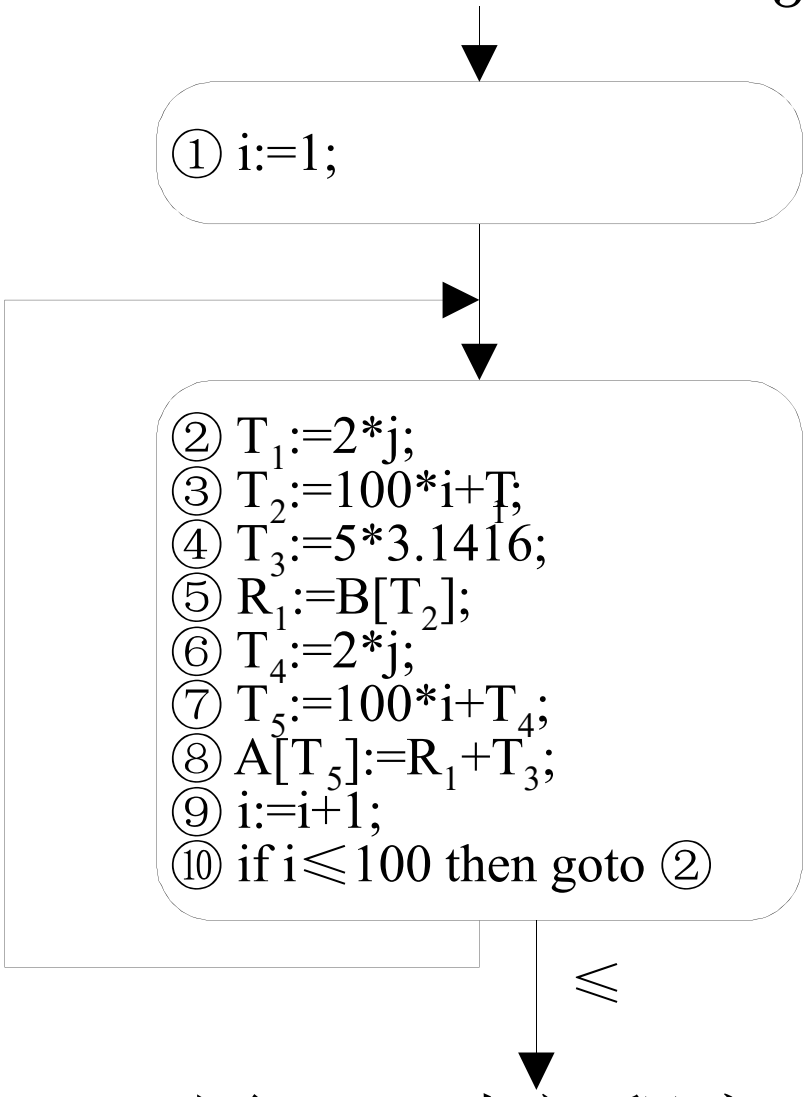


图11.2 中间程序1

- ④中的运算是常量运算，可以在编译阶段完成，归并成常量T3:=15.708。
- 这样做的好处是明显的，就本例可看出，因为它在循环语句中，所以5*3.1416需要重复计算100次。
- 如在编译时归并好，则只要算一次。这就是所谓的常量合并。

11.2 优化举例

4:公共子表达式的消除

- ②和⑥都要计算2*j，而且在②和⑥之间j之值并未发生变化，因此无须计算两次，可以取消对⑥的计算。
- ③和⑦都要计算100*i+t_1，(注意：t_4实质上就有t_1)，而且在③和⑦之间，i和t_1之值都未发生变化，因此，也只要计算一次就够了，可以取消⑦的计算。
- 这些优化是在基本块上进行的，所以是局部优化。经优化后，得到图11.3所示的中间程序。

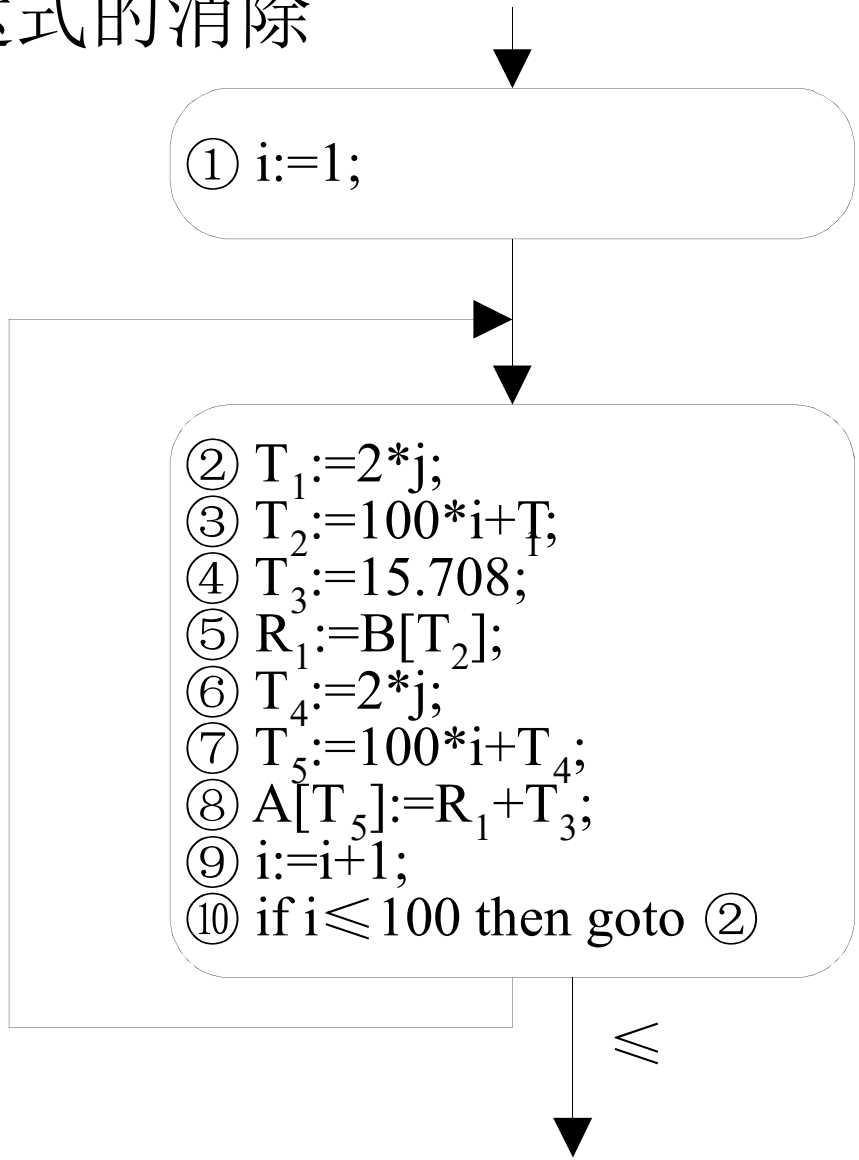


图11.2 常量合并

11.2 优化举例

5:运算强度的削弱

- i之值的变化是有规律的
  - T_1之值在循环中保持不变
  - ③中的100*i+T_1是线性表达式
  - i分别取值1,2,…,100时
  - 该值100+T_1,200+T_1,…, 10000+T_1
  - 把T_2:=100*i+T_1分为T2:=T_1和T_2:=T_2+100
  - 将T_2:=T_1提至循环外面
  - 让T_2:=T_2+100留在循环内
  - 就把乘法运算变成了加法运算
- 由于与循环控制变量i 相关在非线性块上的优化，所以是全局优化。

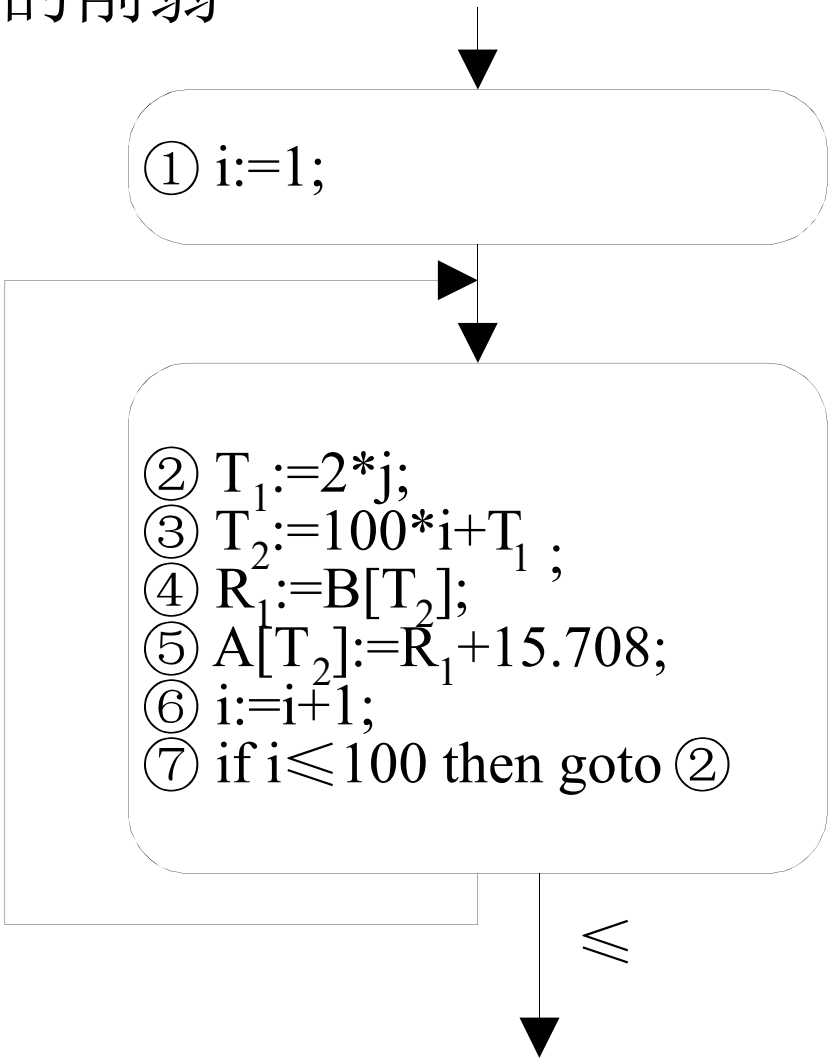


图11.3 公共子表达式的消除

11.2 优化举例

6:循环中不变式的外提

- T_1=2*j外提
- 由于这里的循环中不变式的外提是在非线性块上的优化，所以是全局优化。
- 经这两步优化后，便得到图11.4中所示的中间程序。

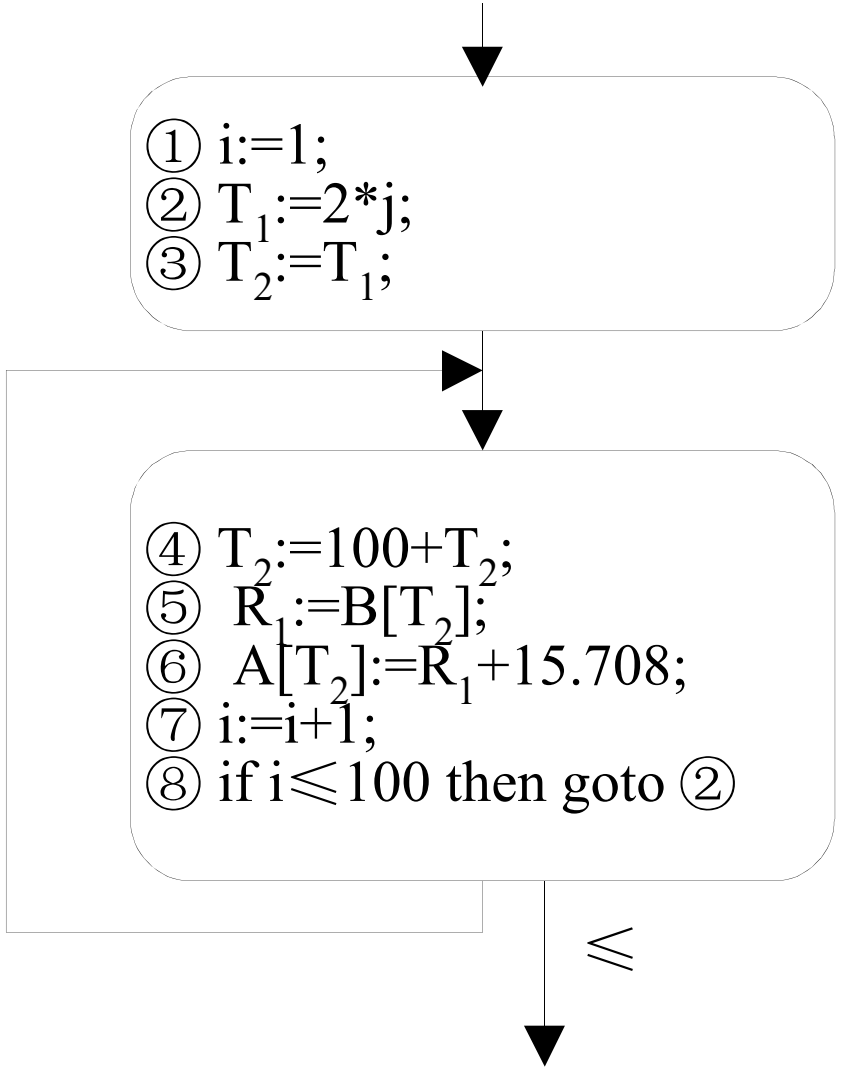


图11.4 运算强度的削弱与常量外提

11.2 优化举例

7:变换循环控制条件

- 可以用终态比较法来替代计数法控制循环的流程
- 从而可以去掉①和⑦两条代码．这种优化称为变换循环控制条件，
- 经此步优化后便得到图11.5所示的简化形式。

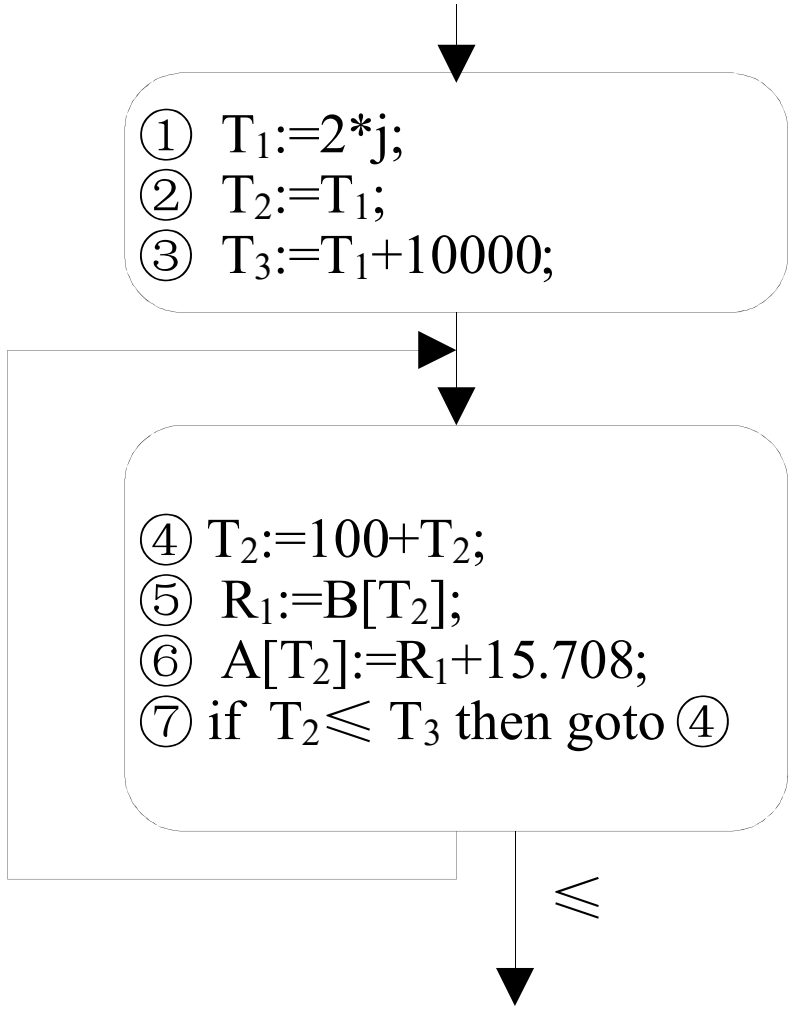


图11.5 变换循环控制条件

优化举例

给出下列中间代码的①入口语句集、基本块；②常量合并；③公共表达式的消除；④循环中不变式的外提；⑤运算强度的削弱。

(1) i:=1;
(2) read j;
(3) T1:=2*j;
(4) T2:=100*i+T1;
(5) T3:=3*3.14;
(6) T4:=2*j;
(7) T5:=T2+T3+T4;
(8) write T5;
(9) i:=i+1;
(10) if i<=100 then goto (3)
(11) halt;

解：① 入口语句集={1，3，11}

基本块={(1,2), （3,10), (11,11)}

② 常量合并：(5) T3:=9.42;

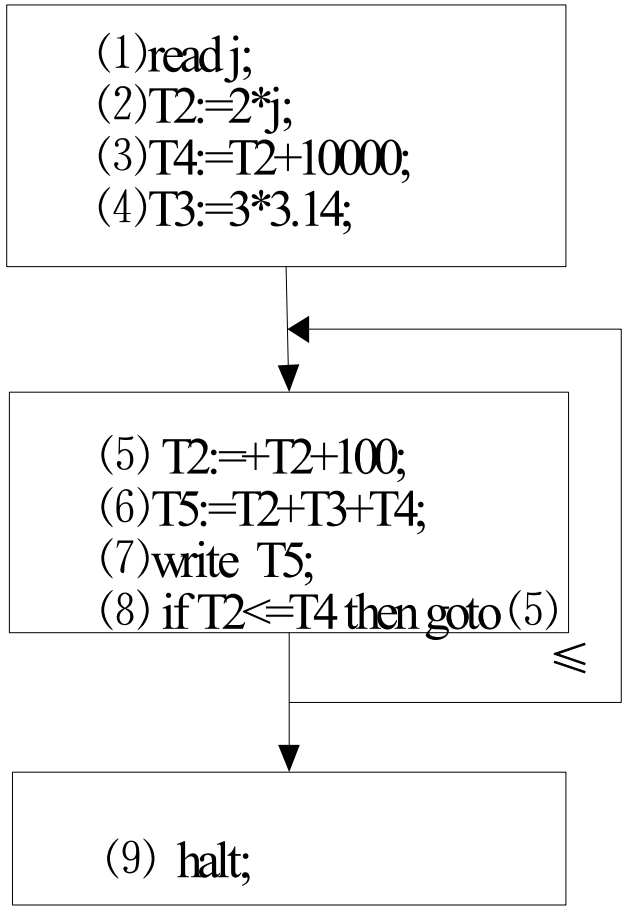
③ 公共表达式的消除：消除 (6) T4:=2*j;

(7) T5:=T2+T3+T1;

④ 循环中不变式的外提；(3) T1:=2*j; 外提

⑤ 运算强度的削弱。 T2:= 2*j; T4=T2+10000;

(4) T2:= T2 +100; (10) if T2<=T4 then goto (4)



11.4 循环优化(1)

- 在编译程序的优化工作中，循环优化占有十分重要的地位.
  - 对于循环次数为n的一个循环，每节省循环体内一条目标指令，运行时就可少执行n条指令；
  - 对于k重循环的内循环而言，每节省一条目标指令，运行时就可少执行$n_1 \times n_2 \times \ldots \times n_k$条指令($n_k$表示第k层循环的次数)。
- 在高级语言的源程序中，数组元素的使用频率较高
  - 数组元素的使用通常是与循环相关联的．对编译程序而言，就得在循环中计算数组元素的地址，而每次直接按数组元素的地址计算公式来计算数组元素的地址，其效率是很低的.
  - 特别是当数组元素位于多重循环的内循环中时，由于其计算量很大，运行时间就会大为增加.

11.4 循环优化(2)

- Bauer和Samelson就提出了“循环中数组元素地址计算的优化”这一课题，并给出了一种解决办法，其主要思路是：

①以较少的指令替代直接转“数组元素地址计算子程序”.

②把地址计算的工作拆开，能在编译时计算就在编译时计算(常量合并)；能在循环外面或较外层循环处计算，就尽量拿到外面计算(外提不变式).

③在内循环中，用加改变量的办法，使数组元素的地址随着循环控制变量值的变化而变化，不必每次都从头算起(强度削弱).

- 大多数编译程序中的循环优化工作差不多都是按此思路实现的.

11.5 借助DAG进行优化

1:无环有向图

- DAG(Directed Acyclic Graph)即无环有向图，它是有向图中的一种。
- 在一有向图中，称任一有向边序列$n_1 \rightarrow n_2$, $n_2 \rightarrow n_3$, …, $n_{i-1} \rightarrow n_i$为从结点n_1到结点n_i的一条通路，
- 若其中$n_1=n_i$，则称该通路为环路。例如，图11.6中所示的(n_2,n_2)和(n_3,n_4,n_3)就是环路。
- 若一有向图中任一通路都不是环路，则称该有向图为无环有向图(DAG)。

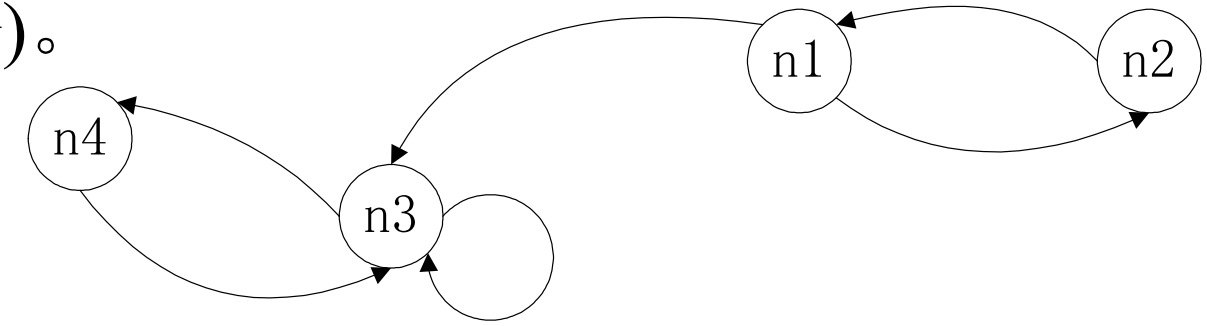


图11.6 环路图

11.5 借助DAG进行优化

2: 无环有向图描述四元式

- 可用无环有向图来描述四元式。
- 例如，与四元式(op B C A)对应的DAG如图11.7所示。
- 利用DAG来进行优化的主要思想是：
- 将一基本块中的每一个四元式依次表示成对应的一个DAG，
- 该基本块就对应一较大的DAG(即其中各个四元式的DAG的合成)。
- 再按原来构造DAG结点的顺序重写四元式序列，便可得到“合并了已知量”、“删除了无用赋值”、“删除了多余运算”的等价的基本块——优化了的基本块。

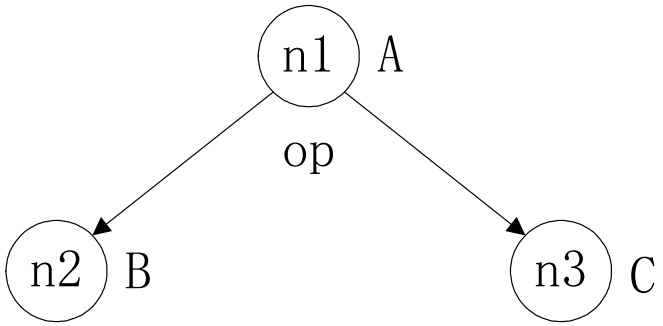


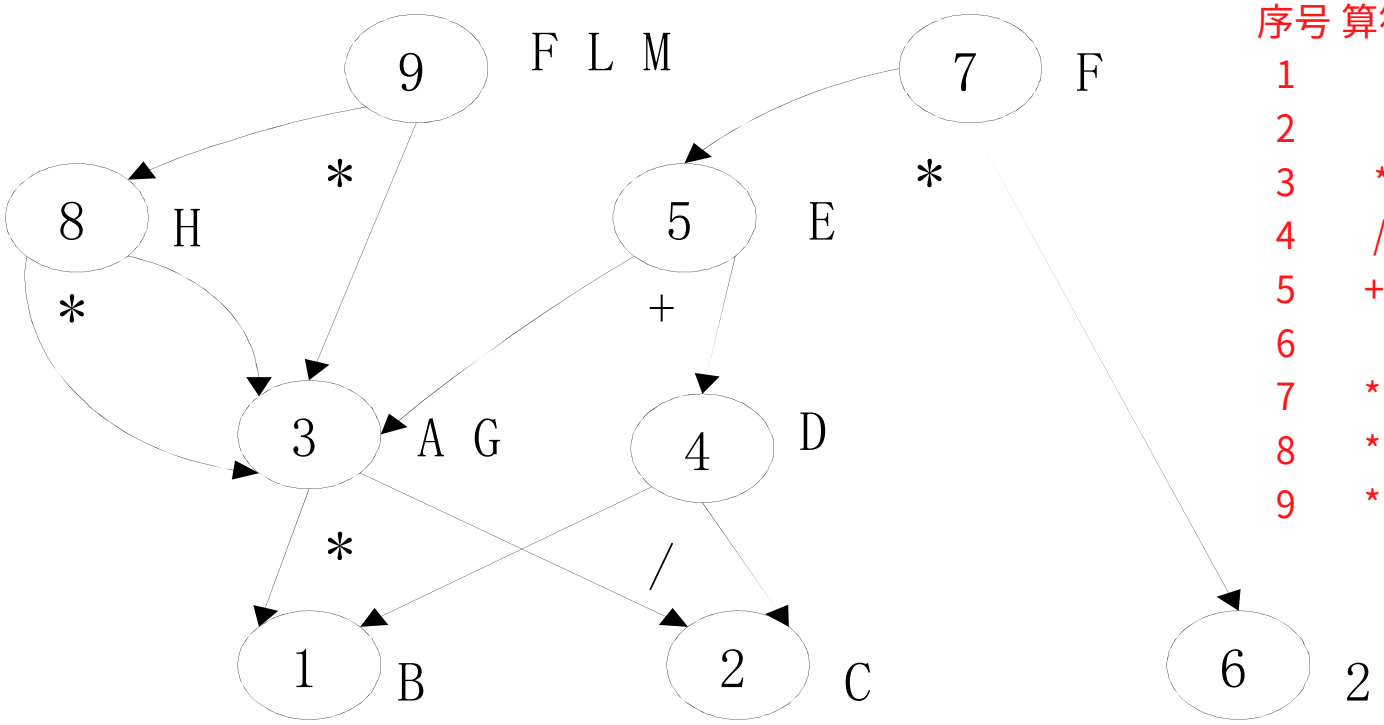
图11.7　四元式的无环有向图

11.5 借助DAG进行优化

- 如下的基本块:

A:=B*C
D:=B/C
E:=A+D
F:=E*2
G:=B*C
H:=G*G
F:=H*G
L:=F
M:=L

3: 无环有向图描述基本块



序号	算符	名称	左儿	右儿
1		B		
2		C		
3	*	AG	1	2
4	/	D	1	2
5	+	E	3	4
6		2		
7	*	F	5	6
8	*	H	3	3
9	*	FLM	8	3

四元式基本块的DAG

- 试利用DAG对其进行优化，并就以下两种情况分别写出优化后的四元式序列：

①假设只有G、L、M在该基本块后面还要被引用；

②假设只有F在该基本块后面还要被引用。

1：
G:=B*C
H:=G*G
L:=H*G
M:=L

2:
S1:=B*C
S2:=S1*S1
F:=S2*S1

11.6 并行分支的优化(1)

- 在具有多处理机(或多处理部件)的机器上，把彼此无关的程序段编成并行分支，就可同时执行，从而既发挥了机器的效率又提高了程序的执行速度。

  例如：x_1:=x+sqrt(d)-r*cos(t)　　(11.1)

  y_1:=y+log(d)+r*sin(t)　　(11.2)

- 在式(11.1)中，x+sqrt(d)和r*cos(t)属并行分支；在式(11.2)中，y+log(d)和r*sin(t)属并行分支. 事实上，式(11.1)和式(11.2)本身也是两个并行分支，它们都可以进行并行分支的优化，即把它们指派到不同的处理机(处理部件)上并行执行。当然，这需要根据具体机器的特点而定。

11.6 并行分支的优化(2)

- 如前所述，一个串行程序总可以划分成一些基本块．若把每个基本块看做一个结点，把块与块之间的联系用有向连线表示，那么一个静态程序又可看做一个有向图．于是，并行分支的识别就可用图论，逻辑代数中的关系论等工具来进行。F.E.Allen的“控制流程分析”就是基于这种方法建立的一种优化理论。
- 直观地讲，假定两个基本块A与B彼此独立，即A既不直接也不间接地依赖于B，B也不依赖于A，用图论的话讲就是彼此互不抵达，那么，A和B就是并行分支，它们是可并行执行的．并行分支识别的主要工作在于寻找这种彼此独立的基本块，问题的难点在于：

① 任何两个基本块彼此独立的充要条件是什么?

② 如何确切定义基本块?

11.7 窥孔优化

- MeKeeman提出了一种窥孔优化技术。

➢这种优化可在中间代码级上进行，

➢但更多的是在目标代码级上进行。

➢所谓窥孔优化是指：

✓每次只查看所生成的目标代码中相邻的几条指令，并对它们进行优化，以获得等价的、更短的代码序列。

✓这种优化通常包括删去多余的存取指令、删去决不会执行的代码、充分利用机器指令特点等等。

11.8 小结(1)

- 优化的目的在于节省运行时间和缩减存储空间；
- 不过这两方面常常是难以兼得的。要根据具体情况的折中办法。
- 例如，对于循环优化，主要侧重于节省时间．几条原则是：

➢ 第一，在编译阶段能计算的量决不留到运行时刻去做。常量合并。

➢ 第二，能在外层循环中计算的量决不放在内层去计算。

➢ 第三，能够公用存储单元(特别是寄存器)的尽量让其公用。临时工作单元的分配就是实现这条原则的例子

11.8 小结(2)

- 在全局优化中，现今多采用数据流分析技术
  - 即把程序表示成以基本块为结点的控制流程图
  - 应用必经结点和回边等概念来查找程序中的循环
  - 在整个程序范围内通过分析变量的定值和引用之间的关系
  - 建立若干相关的数据流方程，通过求解这些方程来进行基本块内的优化和循环优化.
- 优化的好处是明显的，但却增添了编译程序本身的复杂性。
- 当今，优化追求的“小存储空间，高运行速度”的目标，已不再是主要问题。