第 6 章 中间代码优化

引言故事：

中间代码的优化是提升编译生成的代码的运行效率的重要途径之一，这类似将文言文翻译为普通话后对翻译内容进行的润色和加工。与文言文的翻译相比，中间代码优化还可以对原始代码进行进一步的处理，以改进可维护性、执行效率等方面的表现。

文言文的用词习惯和现代汉语有很大差异[1]，通常，将文言文直接翻译成现代汉语需要进行润色，从而使翻译结果更符合现代汉语的表达习惯和语法规范，以提高读者的阅读体验和理解度。文言文和现代汉语在语法结构上有很大的差异，因此需要进行相应的调整。例如，在文言文中，“吾”、“尔”等代词在现代汉语中可以用“我”、“你”等代替；“者”、“乎”等助词可以省略或者改为“的”、“吗”等。

文言文发轫于诗，肇于史，经于骈，成熟定形于文言文[2]。文言文与现代汉语在句式、语法上也有很大差别，因此在翻译过程中需要对其句式和语法进行改写，使其更符合现代汉语的表达习惯。例如：“父亲年已八十有六，身居高位，子女纷多，忧劳难免，今为新法所逼，欲自缢，以绝后患。”润色后汉语可为“父亲已经 86 岁了，担任了较高的职位，有很多子女，日常不免有许多烦心事。现在受到新法规的压迫，想要自杀，以免日后带来麻烦。”

由于文言文的表达方式较为简洁，常常需要通过上下文来理解其含义。因此在翻译过程中，需要根据上下文信息，对文言文中未明确表达的意思进行补充，让读者更加容易理解。例如：“李白一生好饮，少有清名，每到一处，必先醉数日。”润色后为“李白喜欢喝酒，年少时就有很高的声望，每到一个地方，总是要先喝醉几天。”

---

[1] 吴小宁.论文言文中的词类活用现象[J].现代商贸工业,2018,39(27):167-168.DOI:10.19311/j.cnki.1672-3198.2018.27.083.

[2] 王文元.论文言文与白话文的转型[J].天中学刊,2007,No.147(06):91-95.

文言文翻译时的润色过程和编译器的中间代码优化过程是非常相似的。具体来说，前者可以看作是对原始文言文的语言、文化、历史等方面的理解和转化，以符合现代汉语的表达习惯和语法规则；后者可以看作是对源代码进行语义分析、优化和转换，以提高代码的执行效率和性能。

在文言文翻译到普通话的润色过程中，译者需要根据读者的背景、理解能力、语言水平等因素，选择适当的词汇、语法、词序等，以使得翻译后的文本更加易懂、通顺、准确。同样地，在编译器的中间代码优化过程中，编译器需要分析源代码的数据依赖、控制流、内存使用等方面，进行各种优化手段的应用，如常量折叠、循环展开、函数内联等，以提高代码的执行效率和性能。

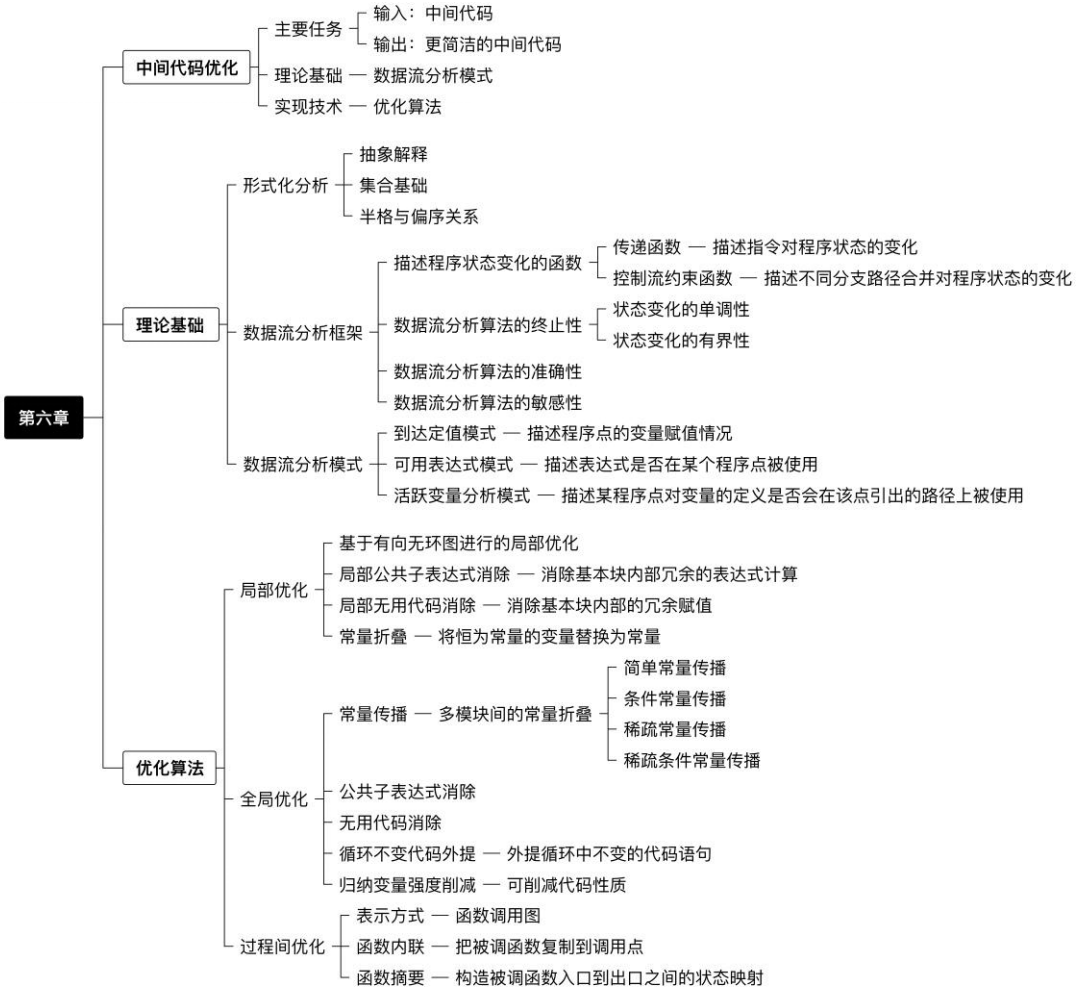
此外，中间代码优化过程是在原始语言的基础上进行的，保留了原始语言的逻辑结构和功能实现。中间代码优化过程的目标是优化代码的表现，使其更加易懂、高效、优雅。

本章要点：

中间代码优化是编译器设计中的重点与难点。在该步骤中，编译器将生成的中间代码转换为更加简洁、高效且语义不变的中间代码。编译器设计优化管道与管道内的各个优化模块，通过依次执行优化管道里的每个模块，消除中间代码内部的冗余代码，改变代码执行的顺序，降低执行过程中的程序开销，并且将代码转化为更加便于目标代码相关优化的形式。本章内容主要涵盖了在编译过程中进行中间代码优化的理论和实现方法。为便于读者理解中间代码优化的相关理论，本章首先介绍用于中间代码优化的数据流分析框架，使用通俗易懂的方式形式化地推演数据流分析框架的单调性与有界性，及数据流分析结果的准确性。然后，本章介绍三种通用的数据流分析模式及其算法的构造方式，以便于读者针对不同优化场景实现优化模块。另外，本章也讨论了具体的实践技术内容。本章基于三元式中间代码表示形式进行优化，将中间代码优化分为三种情况：局部优化、全局优化与过程间优化，着重介绍全局优化中公共子表达式消除、常量传播、无用代码消除、循环不变代码外提与归纳变量强度削减等优化的实现方法，帮助读者在所提供的技术指导下，完成中间代码的优化工作。

需要注意的是，由于本次实践内容的代码会与之前实践内容中已经写好的代码进行对接，因此保持一个良好的代码风格、系统地设计代码结构和各模块之间的接口对于整个实践内容来说是相当重要的。

思维导图：


第六章
中间代码优化
主要任务
输入：中间代码
输出：更简洁的中间代码
理论基础 — 数据流分析模式
实现技术 — 优化算法
理论基础
形式化分析
抽象解释
集合基础
半格与偏序关系
数据流分析框架
描述程序状态变化的函数
传递函数 — 描述指令对程序状态的变化
控制流约束函数 — 描述不同分支路径合并对程序状态的变化
数据流分析算法的终止性
状态变化的单调性
状态变化的有界性
数据流分析算法的准确性
数据流分析算法的敏感性
数据流分析模式
到达定值模式 — 描述程序点的变量赋值情况
可用表达式模式 — 描述表达式是否在某个程序点被使用
活跃变量分析模式 — 描述某程序点对变量的定义是否会在该点引出的路径上被使用
优化算法
局部优化
基于有向无环图进行的局部优化
局部公共子表达式消除 — 消除基本块内部冗余的表达式计算
局部无用代码消除 — 消除基本块内部的冗余赋值
常量折叠 — 将恒为常量的变量替换为常量
全局优化
常量传播 — 多模块间的常量折叠
简单常量传播
条件常量传播
稀疏常量传播
稀疏条件常量传播
公共子表达式消除
无用代码消除
循环不变代码外提 — 外提循环中不变的代码语句
归纳变量强度削减 — 可削减代码性质
过程间优化
表示方式 — 函数调用图
函数内联 — 把被调函数复制到调用点
函数摘要 — 构造被调函数入口到出口之间的状态映射

6.1 中间代码优化的理论方法

经过词法分析、语法分析、语义分析后，编译器前端的工作告一段落。中端将源代码翻译成中间代码表示形式，并对中间代码进行机器无关的优化。出于安全性、可维护性或编程习惯的考量，程序开发者编写的源代码中通常有大量冗余代码，并且，一部分冗余可能来源于中间代码生成过程，例如添加了多余的中间变量，重复的多维数组访问的指针运算等。程序的冗余通常导致编译器执行效率低下，时空开销高昂等问题，并且最终导致生成的目标代码运行效率低下。区别于目标机器相关的优化，机器无关优化不考虑目标机器的寄存器与机器指令。基于中间代码的优化不需考虑源语言与目标语言存在的差异，是独立于前端和后端进行优化的全部过程。有哪些程序片段需要被优化，应当运用哪些优化方法，应当以什么顺序进行代码优化？这些问题决定了优化的时空开销，及最终中间代码优化的效果。

由于程序语言的复杂性、程序性质的多样性，针对程序的分析往往具有相当大的难度。根据 Rice 定理：对于程序行为的任何非平凡属性，都不存在可以检查该属性的通用算法。因此，没有一种优化的方式能够宣称其优化的结果到了最佳。但是，这一消极的结论在另一方面有着积极的影响：总能够提出更好的优化方式，使得代码更加简洁与高效。

6.1.1 中间代码优化概述

在程序执行的任何位置，当前程序使用的数据在内存中的存储位置与该数据的内容（符号值或实际值）构成了程序状态。程序命令的顺序执行，本质上是对内存中存储的数据的定义与使用操作，操作的执行改变了程序状态。比如，我们声明了一个变量 x 并定义，就会开辟一段内存来存储 x 所代表的值。

中间代码优化的基础在于跟踪并分析程序状态的改变。比如，在执行某条指令时，当前的变量存储的值是否具有唯一的常量，如果是，那么我们就可以将这个变量替换成一个常量。又或许，在执行某条指令时，一个变量存储的值是否会在被使用之前就被覆盖掉，如果是，我们就不需要在内存或是寄存器里存储这个值。

一系列地连续重写中间代码以消除效率低下和无法轻易转换为机器代码的代码片段的方法或

函数构成了编译器的中间代码优化模块。这些方法或函数通常被称为**趟（Pass）**。机器无关优化模块排列趟的执行顺序，构成编译器的**优化管道（Pipeline）**。在某些编译器中，IR 格式在整个优化管道中保持固定，在另一些编译器中，格式在某些趟执行完毕后会发生改变。对于格式固定的优化管道，趟的顺序是相对灵活的，可以运行大量优化序列，并不会导致编译错误或是编译器崩溃。

根据处理粒度的不同，优化通常分为局部代码优化（基本块内部）、全局代码优化（多个基本块或函数内部）和过程间代码优化（跨越函数边界）三种。这三种优化在优化管道中按顺序执行。根据冗余原因的不同，优化通常分为子表达式削减、常量传播优化、循环相关优化、无用代码消除、别名相关优化、内存相关优化等。根据优化需求的不同，优化通常使用不同的数据流分析模式进行分析。常用的数据流分析模式有到达定值（针对循环优化、常量传播等），可用表达式（针对全局公共子表达式消除等），活跃变量分析（针对无用代码削除等）。

优化管道中趟的执行顺序由编译器开发人员设计，合理的优化顺序能够使优化模块在合理的时空开销下获得更好的优化效果，而不合理的顺序使得优化管道需要反复执行同一个优化方法。因此，编译器研究中一个重要的研究课题是构造更好的优化管道。

在实践中，应当生成与中间代码生成时格式相同但是更加简洁高效的中间代码。而本书讨论设计的优化管道中趟的构造与执行顺序、生成中间代码的语义一致性、代码运行时间、执行操作次数等，将作为代码优化的评估指标。

在中间代码的实践内容中，生成的中间代码包含了大量的跳转语句，程序的执行没有明显的顺序，使得阅读代码很难分辨其中的执行逻辑。为了更好地描述程序中的值被定义和使用的顺序，我们可以使用控制流图的形式，对程序代码进行划分。控制流图是一个有向图，其中节点表示一个基本块，有向边表示基本块之间的跳转。

图 6.1 中的代码贯穿局部优化与全局优化实例，其控制流图如图 6.2 所示。

我们前面章节中已经介绍过，程序执行过程中，只能从基本块的第一个指令进入该块，从最后一个指令离开该块。每次程序执行过程中访问基本块时，必须按顺序从头到尾执行其中每一条指令。

```
1   a = read();                21      if(a > b){
2   b = read();                22          f = f + d;
3   c = read();                23          h = a + 3 * y;
4   d = a + b;                 24      }else{
5   e = c * b;                 25          while(j > 0){
6   f = a + b;                 26              j = j - 1;
7   f = 5;                     27              g = e / (f - 1);
8   g = e + d;                 28              h = 4 * j;
9   h = c * f;                 29              if(y < d){
10  x = b - 3;                 30                  g = f + h;
11  y = 2;                     31                  x = a * c;
12  a = x - y;                 32              }
13  b = e - h;                 33          }
14  i = 0;                     34      }
15  j = 10;                    35  }
16  while(i < 10){             36  write(x);
17      i = i + 1;             37  write(h);
18      d = b - a * c;         38  write(g);
19      e = 4 * i;
20      g = x + 4;
```

图 6.1 源代码示例

基本块内部的每一条指令，都代表了一种程序操作或行为，程序在基本块代码的执行过程中，依次执行这些行为。我们可以使用程序状态图对程序内部行为进行抽象描述，状态图记录每一个指令所代表的程序行为和对程序状态变化造成的影响。基本块中代码按照顺序改变程序状态这种受限的形式，使得基本块非常易于分析。

例如，对于图 6.2 中的基本块 B2，当程序执行时，总是从基本块 B1 的出口跳转到基本块 B2 的入口，按照从头到尾的顺序依次执行基本块 B2 中的程序代码，然后从基本块 B2 的出口跳转到 B4 或 B5 的入口。

为了表示程序运行中会可能会遍历到的所有路径，以及路径中基本块之间的跳转及执行的关系，研究者通常使用**控制流图（Control-Flow Graph，CFG）**来抽象表达代码执行的顺序，及代码执行时必然成立的性质。基于控制流图，我们可以分析程序的状态，从而根据程序状态进行优化与静态检查。

基于控制流图，编译器设计者得以分析程序内部数据流动，并且基于流图进行代码的转换。

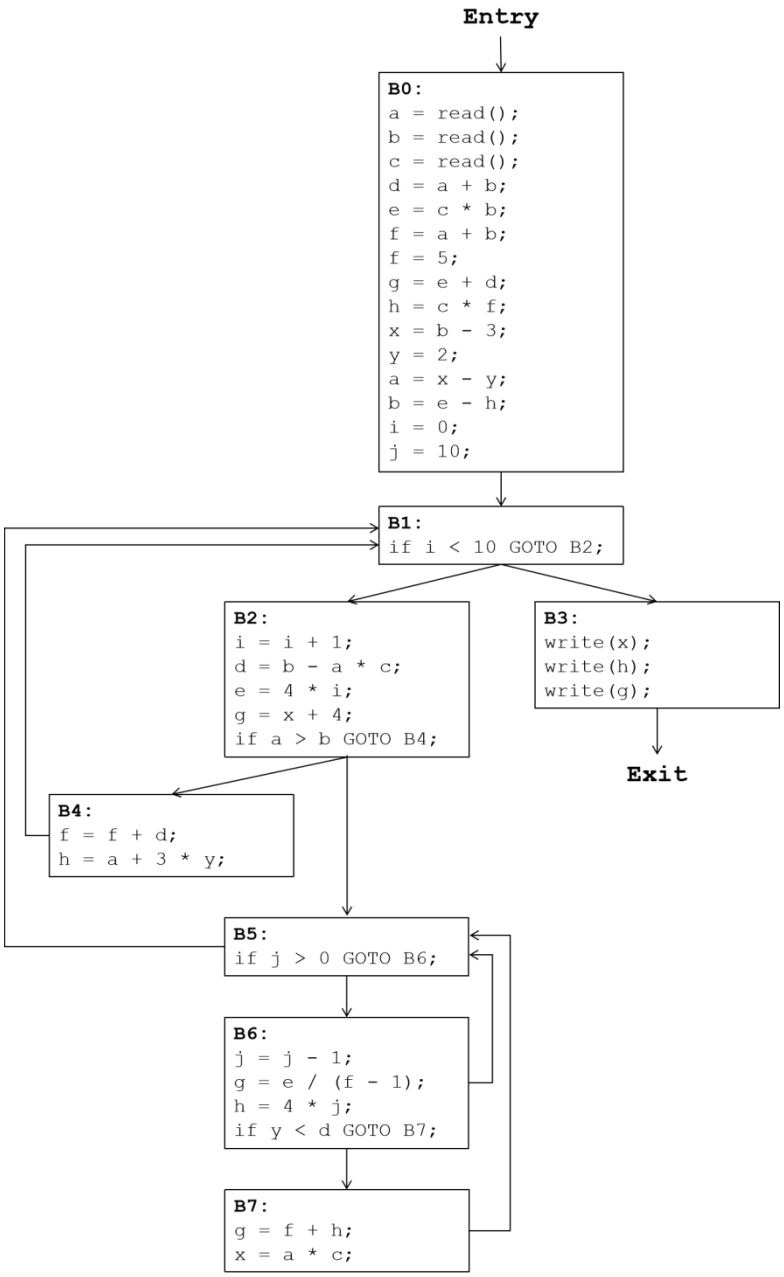


图 6.2 控制流图

6.1.2 数据流分析理论与框架

在基本块中，数据的流动是线性的，而全局的情况下，因存在循环、函数调用等情况，数据

的流动变得复杂。比如，全局公共子表达式消除就需要确定在程序的任何可能执行路径上，是否都存在内容相同且值未改变的表达式。

因此，一个好的用于程序分析的框架是我们一切工作的基石。单调框架的诞生使得进行程序分析的人能够构造出一个精确的、数学形式化的分析，基于这个框架，我们能够有效地描述程序内部的状态变化。

为了在复杂的控制流中发掘这种数据流动的关系，编译器的设计者将**数据流分析（Data-Flow Analysis）**技术引入到全局优化工作之中。基于前文所述，我们可以将程序执行的过程看作是一系列程序状态的转换，而针对特定的问题，我们只需要进行**抽象解释（Abstract Interpretation）**，从程序状态中抽取对应的信息，来解决特定的数据流分析问题。

抽象解释（Abstract Interpretation）理论与框架，相对于**具体解释（Concrete Interpretation）**，旨在通过对我们所感兴趣的问题进行近似的抽象，取出其中的关键部分进行分析，从而使得程序内部的状态是有限的。基于单调框架的抽象解释，使得我们的分析有两个特征：程序状态是有限的并且状态的变化是单调的，单调而有限的变化使得我们的分析总能在不动点停止，从而获取相对精确的分析结果。

我们用一个例子来比较抽象解释与具体解释之间的差距：数学中的考拉兹猜想，对于任意的正整数 n，($n \in N^+$)，若为偶数则除以 2，若为奇数则乘 3 再加 1，如此反复，其终将到达一个不动点，此时 n 的值为 1。虽其形式看似简单，但作为一个猜想，尚未被证实或者证伪。此时，若我们需要写一个程序寻找一个正整数，若能找到一个以上猜想的反例，则计算停止并且输出结果，那么此时，我们是不知道机器是否会停止的，事实上，计算机目前验证了 $5x10^{18}$ 以内的正整数均能满足以上猜想。

具体解释即是记录在计算过程中所有可能获得的 n 的值，而在程序执行的过程中，有时我们并不需要获取这么多信息。抽象解释即是从程序中抽取我们所需要的信息，比如，如果将 n 的值用于条件判断，如 if(n>0)，则此时我们只需得知 n 是否大于零。那么，对于以上的所有运算过程，虽然我们不知道 n 的所有可能的取值，但是我们可知，n 永远为一正整数，所以该跳转语句将永远跳转到 true 分支。所以，抽象解释在舍弃部分精度的情况下，使我们花费有限的时空开销获得

所需的分析结果。

集合论知识基础

（1）集合的定义。在《集合论初步》中，对集合进行了一个刻画：“吾人直观或思维之对象，如为相异而确定之物，其总括之全体即谓之集合，其组成此集合之物谓之集合的元素。”我们通常用大写字母表示集合，如 A、B、C 等，用小写字母表示集合中的元素，如 a、b、c 等。

（2）集合的描述方法。集合有两种描述方法。分别为外延法和概括法：

外延法：

V={a,b,c,d,e}

用枚举的方式列举集合 V 中的所有元素。

概括法：

$\boldsymbol{Z}^{+}=\{x\in\boldsymbol{Z}\mid x>0\}$

描述正整数集 $\boldsymbol{Z}^{+}$中所有元素所包含的性质，对于所有 $\boldsymbol{Z}^{+}$中的元素 x，x>0。

（3）属于。对于集合 $\boldsymbol{Z}^{+}$，若一元素属于集合 $\boldsymbol{Z}^{+}$，如 1，则记为 $1\in\boldsymbol{Z}^{+}$，若一元素不属于 $\boldsymbol{Z}^{+}$，如-1，则记为 $-1\notin\boldsymbol{Z}^{+}$。

（4）集合相等与子集关系。

集合相等当且仅当两个集合拥有同样的元素：

$A=B$ 当且仅当$\forall x\,(x\in A\leftrightarrow x\in B)$

集合 A 是集合 B 的子集，即集合 A 包含于集合 B，记作 $A\subseteq B$：

$\forall x\,(x\in A\rightarrow x\in B)$

如果 $A\subseteq B$ 但是 $A\neq B$，则 A 是 B 的真子集。

（5）空集与幂集。

空集是没有任何元素的集合，表示为Ø。空集是任何集合的子集。

对于集合 S，S 的幂集是 S 所有子集所构成的集合，记作 P(S)：

$P(S)=\{x\mid x\subseteq S\}$

例如，对于集合 V={1,2,3}，其幂集为：

$P(V)=\{Ø,\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$

（6）集合运算。集合有多种运算，此处我们只关注其中的两种运算，并和交。

对于集合 A={a,b}，B={b,c}，有：

集合 A 与集合 B 的并，是 A 和 B 中所有元素的集合，记为 A∪B。

A∪B={a,b,c}

集合 A 与集合 B 的交，是 A 和 B 中均包含的元素的集合，记为 A∩B。

A∩B={b}

（7）最大下界与最小上界。

下界：对于集合 X，若 X 所包含的所有元素均包含于集合 A 和 B 中，则集合 X 被称为集合 A 和集合 B 的下界，即：

$\forall x\,(x\in X\rightarrow x\in A, x\in B)$

最大下界：设 X 是 A 和 B 的下界，即 X⊆A，X⊆B，若对于任何 X，有 A 和 B 的一个下界 Y，X⊆Y，则称 Y 为 A 和 B 的最大下界。

最大下界的唯一性：若存在集合 A 与集合 B 的两个最大下界 X 和 Y，则根据定义，X⊆Y 且 Y⊆X，可知 X=Y。集合 A 与集合 B 的最大下界为 A∩B。

上界：对于集合 X，若集合 A 和 B 中包含的所有元素均包含于 X 中，则集合 X 被称为集合 A 和集合 B 的上界，即：

$\forall a, b\,(a\in A, b\in B\rightarrow a\in X, b\in X)$

最小上界：设 X 是 A 和 B 的上界，即 A⊆X，B⊆X，若对于任何 X，有 A 和 B 的一个上界 Y，Y⊆X，则称 Y 为 A 和 B 的最小上界。

最小上界的唯一性：若存在集合 A 与集合 B 的两个最小上界 X 和 Y，则根据定义，X⊆Y 且 Y⊆X，可知 X=Y。集合 A 与集合 B 的最小上界为 A∪B。

（8）集合的关系。

有序对：有序对(a, b)表示由元素 a 和 b 按照一定顺序排列而成的二元组。

笛卡尔积：对于任意集合 A、B，笛卡尔积 $A\times B=\{(a, b)\,|\,a\in A, b\in B\}$

{1,2,3}×{a,b}={(1, a),(2, a),(3, a),(1, b),(2, b),(3, b)}

关系的定义：如果 A、B 是集合，由 A 到 B 的一个关系是笛卡尔积 A×B 的一个子集，即对于由 A 到 B 的一个关系 R，R⊆A×B。若 A=B，则关系 R 称为集合 A 上的关系。

关系的性质：

①自反性与反自反性：

自反的（Reflexive）: ∀a∈A, (a, a)∈R

反自反的（Irreflexive）：∀a∈A, (a, a)∉R

若 A={a, b}, R⊆A×A, 则：

R={(a, a), (a, b), (b, b)}是自反的；

R={(a, b), (b, a)}是反自反的。

②对称性与反对称性：

对称的（Symmetric）：∀(a, b)∈R, (b, a)∈R

反对称的（Antisymmetric）：若(a, b)∈R, (b, a)∈R, a=b

若 A={a, b}, R⊆A×A, 则：

R={(a, b), (b, a)}是对称的；

R={ (a, b), (a, a)}是反对称的。

③传递性：

传递的（Transitive）：若(a, b)∈R, (b, c)∈R, 则(a, c)∈R

若 A={a, b, c}, R⊆A×A, 则：

R={(a, b), (b, c), (a, c)}是传递的。

偏序关系与半格：

（1）偏序关系定义。非空集合 A 上具有自反性、反对称性和传递性的关系称为集合 A 上的偏序关系，记为≤。若非空集合 A，集合 A 上有偏序关系 R，可知

自反性：若 x∈A，则(x, x)∈R，即 x≤x。

反对称性：若 x, y∈A，(x, y)∈R，则(y, x)∉R，即当 x, y∈A 且 x≤y，则 R 中不存在 y≤x。

传递性：若 x, y, z∈A，(x, y)∈R 且(y, z)∈R，则(x, z)∈R，即当 x, y, z∈A 且 x≤y, y≤z，则 x≤z。

我们看一个集合上的偏序关系的例子，设集合 A={1,2,3,4,5,6}

对于集合 A 上的整除关系 R，有：

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

自反性：对于 A 中的元素 x，x 能够整除 x，故 R 中存在所有有序对(x, x)。

反对称性：对于 A 中的元素 x, y，若 y 能整除 x 且 x≠y，x 不能整除 y。

传递性：对于 A 中的元素 x, y, z，若 y 能整除 x 且 z 能整除 y，则 z 能整除 x。

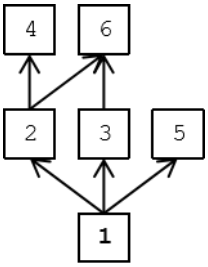


图 6.3 哈斯图示例

（2）哈斯图。哈斯图是一种图形形式的对偏序集的传递简约。对于集合 A 上的偏序集合(R, ≤)，把 R 的每个元素表示为平面上的顶点，省略其中所有的环（有序对中的两个元素相同）和能够以传递关系引出的边（当 R 中存在(x,y)，(y,z)与(x,z)，省略其中用于代表(x,z)的边），根据偏序关系将所有的节点由下而上排列。

比如，对于偏序关系 R，有哈斯图如图 6.3 所示。

我们回忆一下在集合基础中提到的有关幂集的内容。对于集合 A，幂集 P(A)为 A 的所有子集所构成的集合。对于集合 A={a,b,c}，其幂集为：

P(A)={Ø,{a},{b},{c},{a,b},{a,c},{b,c},{a,b,c}}

对于集合 P(A)上的关系⊆，有：

自反性：对于集合 P(A)中的某元素 x，可知 x 是 x 的子集，即 x⊆x 成立。

反对称性：若 P(A)中的某元素 x 和 y，当且仅当 x=y 时，x⊆y 且 y⊆x。

传递性：若 P(A)中的某元素 x，y 和 z，若 x⊆y，y⊆z，则 x⊆z。

因此，我们可知⊆是集合 P(A)上的偏序关系，我们构造哈斯图如图 6.4 所示。

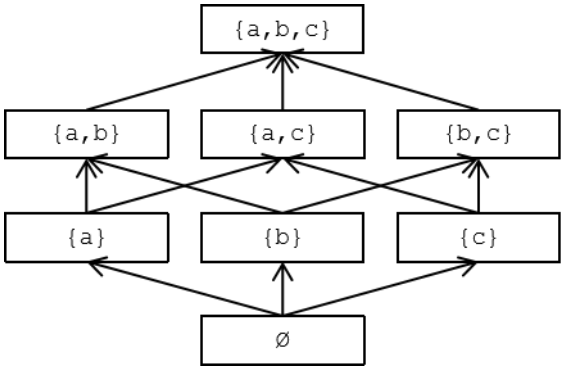


图 6.4 P(A)上的偏序关系⊆的哈斯图

（3）偏序集。若在集合 A 上给定一个偏序关系 R，则将集合 A 中的元素按偏序关系 R 构成一个偏序集，记作(A,R)。

（4）极大元与极小元。对于集合 A 中的元素 a，若不存在(A,R)中的元素 b（a≠b），在偏序关系 R 下，a≤b，则 a 被称为偏序集(A,R)中的极大元。同样的，对于集合 A 中的元素 a，若不存在(A,R)中的元素 b（a≠b），在偏序关系 R 下，b≤a，则 a 被称为偏序集(A,R)中的极小元。如图 6.5 中，a 和 b 是极大元，而 d 和 e 是极小元。

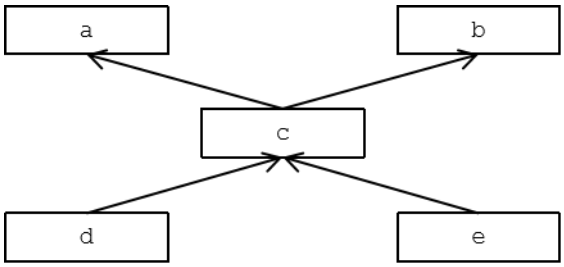


图 6.5 极大元与极小元

（5）最大元与最小元。对于集合 A 中的元素 a，若对于(A,R)中的任意元素 b（a≠b），在偏序关系 R 下，均有 b≤a，则 a 被称为偏序集(A,R)中的最大元。同样的，对于集合 A 中的元素 a，若对于(A,R)中的任意元素 b（a≠b），在偏序关系 R 下，均有 a≤b，则 a 被称为偏序集(A,R)中的

最小元。

如图 6.4 中，对于 P(A)中的任意元素 x，在偏序关系⊆下，均有 x⊆{a,b,c}且Ø⊆x，故{a,b,c}是(P(A),⊆)上的最大元，Ø是(P(A),⊆)上的最小元。因偏序关系的反对称性可知，偏序集上的最大元与最小元是唯一的。

（6）偏序集的下界和上界。对于偏序集(A,R)中的一个子集 B，若存在元素 a∈A，对 B 中的所有元素 b，均有 b≤a，则 a 称为 B 的一个上界。同样的，若存在元素 a∈A，对 B 中的所有元素 b，均有 a≤b，则 a 称为 B 的一个下界。偏序集的子集不一定存在上界或者下界，例如，在图 6.5 中，对于偏序集(A,R)中的元素 a 和 b，不存在一个元素 c∈(A,R)，使得 c 是(A,R)的子集{a,b}的上界。

（7）偏序集的最大下界与最小上界。对于偏序集(A,R)和其子集 B，B 的上界集合为 C，若 C 不为空且对于 C 中的任何元素 y，存在 C 中的一个元素 x，x≤y，则 x 称为集合 B 的最小上界。同样的，若存在 B 的下界集合 D，若 D 不为空且对于 D 中的任意元素 y，存在 D 中的一个元素 x，y≤x，则 x 称为集合 B 的最大下界。根据前文所述，我们可知，若一个集合有最大下界或最小上界，最大下界或最小上界唯一。

（8）半格。如果一个偏序集的每对元素都有最大下界或都有最小上界，就称这个偏序集为半格。以图 6.3 为例，对于集合 A 及 A 上的整除关系 R（前文我们已知其为 A 上的偏序关系），有

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

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

其中,A 中的每对元素均能在 A 中找到其最大下界，最大下界在此处的意义是均能被两个元素整除的最大的元素，例如，对于元素 4 和 6，其均能被 1 和 2 整除，而 2 是 4，6 的最大下界。

因元素 1 的存在，集合 A 中的每对元素均能找到最大下界，因此偏序集(A,R)构成半格。

进行数据流分析时，我们关注：1. 在一条语句执行的前后，程序状态发生的变化；2. 在不同执行路径的交汇处，程序状态的合并应采取什么样的方式。在抽象解释中，我们将程序状态的抽象表示为格中的元素，而每次进行数据流分析时，我们通常只使用其中的一半。为了说明程序状

态的变化情况，此处举一个较为简单的例子（图 6.6）。

我们关注图 6.6 中执行每一条语句前后，变量 x 所有可能的取值及其变化。对 x 的赋值语句会导致其值的改变，如当执行 B5 中的赋值语句 x=c 后，其将 x 的值（从 b）变为 c。执行不同的路径可能导致 x 的取值有多种，如执行路径 B1-B2-B4-B6 或 B1-B3-B5-B6，x 的值分别为 a 和 c。

除对 x 的赋值语句外，其他的语句均不可能改变 x 的取值（不考虑别名关系和指针等），如执行 B4 中的语句 y=x，x 的值不变。

我们设 x 的可能取值构成的集合为 X，图 6.6 中，程序内部存在三条对 x 的赋值语句 x=a，x=b 和 x=c，A={a,b,c}，X 为幂集 P(A)的一个子集：

$X \subseteq \{Ø,\{a\},\{b\},\{c\},\{a,b\},\{a,c\},\{b,c\},\{a,b,c\}\}$

如果将执行每一条语句看成一次对程序状态的变化，程序内部的五条语句分别对状态产生了如下的改变：

B2 : x=a，X : Ø→{a}，对变量的初次赋值。

B3 : x=b，X : Ø→{b}，对变量的初次赋值。

B4 : y=x，X : {a, b}→{a, b}，未对变量 x 赋值，不改变程序状态。

B5 : x=c，X : {b}→{c}，对先前的赋值的覆盖。

B6 : z=x，X : {a, b, c}→{a, b, c}，未对变量 x 赋值，不改变程序状态。

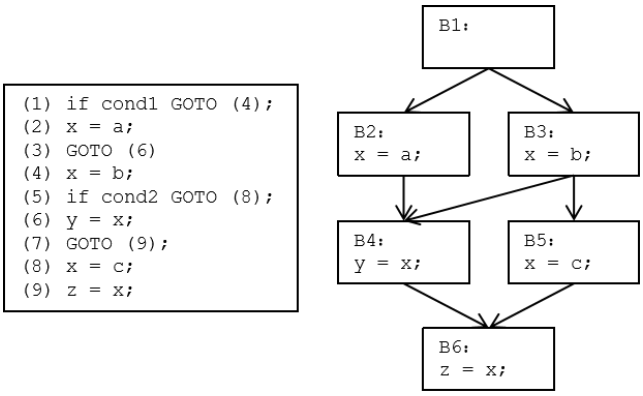


图 6.6 变量 x 的可能值

我们还需关注在程序的不同执行路径的交互处，程序状态的合并应采取什么样的方式。因为这里我们关注的是变量 x 所有可能的取值，所以在交互处，我们把所有 x 的取值都添加到 X 里。

在基本块 B4 的入口处，由基本块 B2 传递过来的 x 的值集为{a}，由基本块 B3 传递过来的 x 的值集为{b}。程序可能执行路径 B1-B2-B4，也可能执行路径 B1-B3-B4，因此，在 B4 的入口处，需要对两个值集进行并运算，即 IN[B4]={a}∪{b}={a,b}，意为 x 可能为 a 或 b。

在进行数据流分析时，首先要保证**正确性（Soundness）**，然后尽量保证**准确性（Truth）**，例如，在分析 B4 入口处 x 的取值时，如果得出的结论是 IN[B4]={a}，那么在后续的优化中就可能直接使用定值 a 替代变量 x（和 y），从而改变了程序语义（执行路径 B1-B3-B4 的结果产生变化），导致了错误的结果，违背了正确性。而如果我们将程序中所有位置上，x 的可能取值均设为{a,b,c}，在后续的优化中，优化器会因 x 均不能被定值所替代，不做任何改变，从而无功而返。虽然程序的优化结果“正确”了，但与优化的预期（Truth）相去甚远。因此，我们此处使用并运算，寻找到值集的最小上界，以尽可能保证准确性。

我们需说明，在幂集 P(A)上的关系⊆是偏序关系。

自反性：对于集合 P(A)上的子集 p，p 是自己的子集，因此 p≤p 成立。

反对称性：对于集合 P(A)上的元素 p 和 q，若 p⊆q，q⊆p，可知 p=q。

传递性：对于集合 P(A)上的子集 p、q 和 r，若 p⊆q，q⊆r，可知 p 是 q 的子集，q 是 r 的子集（p≤q，q≤r），则 p 是 r 的子集（p≤r）。

因此，⊆是集合 P(A)上的偏序关系。

然后，我们说明偏序集(P(A), ⊆)是半格。

因 P(A)是集合 A 的幂集，即 A 的所有子集构成的集合，则可知，对于其中的每对元素 p 和 q，均可找到 r∈P(A)，r⊆p，r⊆q，即其必定为集合{a,b,c}的一个子集，因此偏序集(P(A), ⊆)是半格。

然后，我们需证明，a=p∨q 是 P(A)中任意元素 p，q 的最小上界。

集合运算具有结合性，可交换性与等幂性，因 a∨p=p∨q∨p=p∨q=a，可知 a∨p=a，p≤a。

同理可得 q≤a。

若对任意 P(A)中元素 b，p≤b 且 q≤b，p，q 均为 b 的子集，p∨b=q∨b=b，则 a∨b=(p∨q)∨b=p∨(q∨b)=p∨b=b，即 a∨b=b，a 是 b 的子集。可知 a 是 p，q 的最小上界。

因对偏序集中元素的运算是∪，即并运算，我们也可称这样的半格为并半格。

对于集合的∩运算，读者应自行进行证明。我们称这样的半格为交半格。如果一个偏序集的每对元素都同时具有最大下界和最小上界时我们称该偏序集为**格（Complete Lattice）**。

数据流分析框架

在介绍了诸多概念之后，我们得以介绍数据流分析框架。在学习数据流分析框架之前，我们可能对程序中有多少状态缺乏一个直观的感受：程序执行会产生什么样的结果，程序在执行的过程中会有哪些中间结果。在之前，我们只能通过执行一次程序，动态地获得我们想要的信息，而当学习了数据流分析框架之后，我们可以通过静态的方式，从框架中获取数据流信息，并进一步进行程序验证、优化与缺陷检测。

基于抽象解释理论构造的数据流分析框架有两个非常重要的性质：单调性与有界性，这能确保算法在有限的时空成本下计算出相对精确的结果。

为理解数据流分析框架下，程序状态的有界性，我们首先关注图 6.6 中的例子。在图 6.6 中，我们初步地感受到，在不考虑别名和指针的情况下，变量 x 的可能取值和程序内部对 x 的赋值语句有着直接的联系：当程序内部存在 n 条不同的赋值语句，x 的取值情况即存在 2^n 种。即，对于程序内部的任意位置，x 的可能取值情况为 2^n 中的一种，是有界的。

数据流分析框架会使用迭代算法，迭代地遍历控制流图，模拟程序执行过程，并记录程序状态的变化，我们用以下的例子说明：

我们首先考虑基本块内部程序状态的单调性。假设在基本块的入口处和出口处分别维护 x 的可能值的集合，记作 IN[B]和 OUT[B]，在基本块的内部，程序状态的变化（x 的值集的改变）仅存在两种可能：

基本块内部没有对 x 的赋值语句，则 IN[B]=OUT[B]。

基本块内部存在多条对 x 的赋值语句，最后一条是 $x=x_1$，则无论 IN[B]是什么，OUT[B]={x1}。

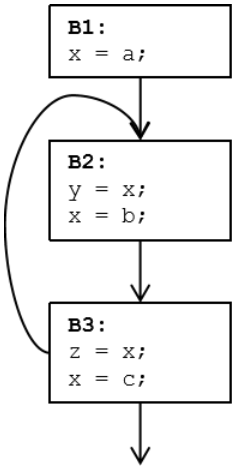
那么，在基本块内部，当 IN[B]相同时，在依次执行基本块内部的语句之后，OUT[B]也相同。

然后，我们关注数据流框架所使用的迭代算法中每次迭代后程序状态的单调性。图 6.7 中展示了每次迭代后每个基本块入口和出口处变量 x 的可能取值。根据前文所述，基本块内部程序状态的变化仅存在两种可能。那么，如果用下标来表示迭代的次数，如 $IN[B]_i$ 代表基本块入口处在第 i 轮迭代时维护的 x 的值集结果，有：

基本块内部没有对 x 的赋值语句，即 $IN[B]_i=OUT[B]_i$，$IN[B]_j=OUT[B]_j$。

基本块内部存在多条对 x 的赋值语句，即 $OUT[B]_i=OUT[B]_j$。

则设 $i<j$，若 $IN[B]_i \leqslant IN[B]_j$ 成立，$OUT[B]_i \leqslant OUT[B]_j$ 也成立，若 $IN[B]_j \leqslant IN[B]_i$ 成立，$OUT[B]_j \leqslant OUT[B]_i$ 也成立。



	Iteration 1	Iteration 2	Iteration 3
IN[B1]	{ }	{ }	{ }
OUT[B1]	{a}	{a}	{a}
IN[B2]	{a}	{a, c}	{a, c}
OUT[B2]	{b}	{b}	{b}
IN[B3]	{b}	{b}	{b}
OUT[B3]	{c}	{c}	{c}

图 6.7 迭代算法中程序状态的单调性

最后，我们关注在程序内部，不同路径的交互处，进行程序状态合并时的单调性。设基本块 X 有两个前驱基本块 A 和 B（如 B2 的前驱基本块 B1 和 B3）。设第 i 轮后基本块 A 的出口处状态为 a，基本块 B 的出口处状态为 b，第 j 轮后基本块 A 的出口处状态为 a’，基本块 B 的出口处状态为 b’，$i<j$，则有：

$IN[X]_i=a \cup b$，$IN[X]_j=a' \cup b'$

又若 $a \leqslant a'$，$b \leqslant b'$，即 $a'=a \cup a_1$，$b'=b \cup b_1$，a_1，b_1 为满足条件的集合。

根据集合运算的性质，有 $a' \cup b'=a \cup a_1 \cup b \cup b_1=(a \cup b) \cup a_1 \cup b_1$。

即当 a⩽a’，b⩽b’时，a∪b⩽a’∪b’成立，满足单调性。

a’≤a，b’≤b 时同理。

对以上三种情况的合并，我们可知：在迭代的过程中，程序状态的变化是单调的。

单调有界准则：

对于一数列$\{X_n\}$，若其从第一项开始，满足 $X_i \le X_{i+1} \le X_{i+2} \ldots$，可称该数列是单调递增的。且若存在一项 T，使得 $X_i \le T$ 恒成立，则称该数列是有上界的。单调递增且有上界，或是单调递减且有下界，该数列是有极限的，即当 i 趋近于正无穷时，$X_i = X_{i+1} = X_{i+2} \ldots$。

那么，假设 x 有 n 种取值情况，其构成的集合 $A=\{x_1, x_2, \ldots, x_n\}$，对于基本块 X 入口处的程序状态 $IN[X]_i$，必为幂集 P(A)中的一个子集且 $IN[X]_i \le \{x_1, x_2, \ldots, x_n\}$，又从前文可知，$IN[X]_i$ 在迭代的过程中单调递增，故其必有一个极限 $IN[X]_N \le \{x_1, x_2, \ldots, x_n\}$，在第 N 轮迭代之后，IN[X]的结果不再改变。

基于迭代算法，程序内部各位置 x 的取值集合的结果将到达一个不动点，作为算法的输出，以便于我们进一步的分析。

先前的理论较为晦涩，难以理解，我们基于图 6.7 进行直观的展示。

程序中 x 的赋值语句共有 3 条：x=a，x=b 和 x=c，其构成的集合 A={a, b, c}，集合 A 的幂集 P(A)={Ø, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}。程序内部 x 的可能取值一定是幂集 P(A)的一个子集。

我们分析 x 的所有可能取值，使用∪运算进行状态合并。前文所述，(P(A), ∪)是偏序集且为并半格，且任意程序内部基本块 B 入口与出口处程序状态 X 的改变具有单调性，有如下的性质：

1. 对于∀a, b∈P(A)，a∪b∈P(A)

2. 对于∀a∈$IN[B]_i$且 j⩾i，a∈$IN[B]_j$（单调递增）

3. 迭代后程序状态到达不动点（值集不再改变）

不动点指，在迭代到第 n 轮后，程序内部基本块入口与出口，变量 x 的可能值集不再改变，

设基本块 B 有两个前驱 B1 和 B2，迭代第 i 轮时 B1 的出口处 x 的值集为 a，B2 的出口处 x 的值集为 b，第 i+1 轮时 B1 的出口处 x 的值集为 a’，B2 的出口处 x 的值集为 b’，i≥n，有：

a=a’，b=b’

则在基本块 B 的入口处程序状态 a∪b=a’∪b’，故程序状态在 B 入口处也不变。

在图 6.7 中，程序在迭代到第 2 轮后到达不动点，在获取了数据流分析的结果之后，我们可以基于结果进行一定的优化或者静态检查，如，在基本块 B3 的开头，x 的值集只有一个元素{b}，即经过数据流分析计算后，x 在基本块 B3 的开头只可能是定值 b，那么，我们可以将对 z 的赋值语句 z=x 替代为 z=b，并且对 x 的赋值 x=b 也可以被消除。

基于上述格的理论，我们介绍本章所使用的数据流分析框架。

一个数据流分析框架(D,V,R,F)由下列元素所组成：

（1）一个数据流方向 D，它的取值包括**前向（Forward）**和**后向（Backward）**。根据前文，我们知道，基本块内部的代码应当按顺序从头到尾执行一遍，因此，前向数据流分析即从头到尾分析执行每一条语句后程序状态的变化，后向数据流分析是前向数据流分析的逆向分析，即从尾到头分析程序状态的变化。后文中介绍的到达定值与可用表达式分析是前向分析，活跃变量分析是后向分析。

（2）一个半格(V,R)，V 代表程序状态的集合，R 是集合的交运算或并运算，用于表示在基本块入口处对不同的前驱（或在出口处对不同的后继）的程序状态的合并。

（3）一个从 V 到 V 的传递函数族 F，用于刻画基本块内部每条语句对程序状态造成的变化。

程序状态的改变存在两种情况，即基本块内部的指令对程序状态的改变和控制流带来的程序状态的改变。用于描述基本块内部指令对程序状态的改变的函数即传递函数族 F：V→V。

对于程序内部的一条指令 s 及其对应的传递函数 f_s∈F，有：

前向数据流分析：OUT[s]=f_s(IN[s])；

后向数据流分析：IN[s]=f_s(OUT[s])。

传递函数描述了在进行数据流分析的过程中，一条语句对程序状态产生的变化。通常，传递函数由三部分组成：

（1）传递前的程序状态 x，此处 x 代表前向分析的 IN[s]或后向分析的 OUT[s]。

（2）因语句 s 生成的程序状态 gen。

（3）因语句 s 失效的程序状态 kill。

通过对传递函数的设计，能够描述不同的抽象情况下，我们所关注的程序状态的改变情况，如，当我们想知道变量的赋值变化的情况，我们构造的传递函数可以是这样的：

$f_s=gen_s \cup (IN[s]-kill_s)$

其中，gen_s表示当前语句生成的赋值关系（define），$kill_s$表示因当前语句而失效的赋值关系。

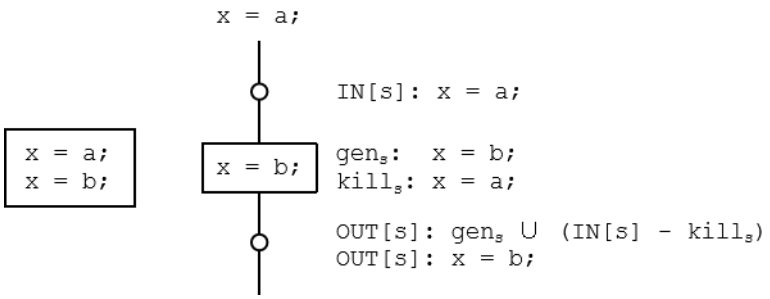


图 6.8 传递函数示例

在图 6.8 中，当进行到达定值分析时，进行的是前向数据流分析，那么，当分析到 s：x=b 时，s 使得对 x 的赋值语句 x=b 生效，gen_s：x=b，此时，对 x 的其他赋值均失效，因此，此时，对 x 的赋值语句 x=a 失效，$kill_s$：x=a。

传递函数族 F 有如下的性质：

（1）F 有一个单元函数 I，使得对于 V 中的所有元素 x，有 I(x)=x。

（2）F 对函数组合运算封闭，即，对于$\forall f, g \in F$，若 h(x)=g(f(x))，则 $h \in F$。

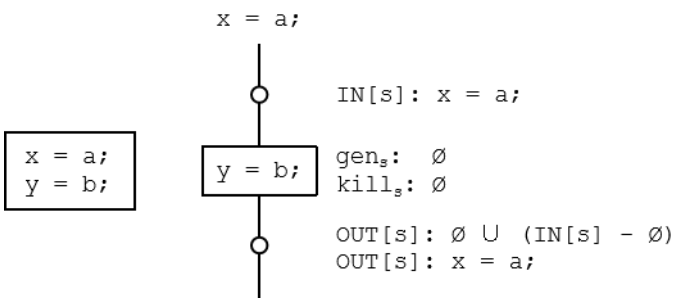


图 6.9 单元函数示例

对于单元函数 I，其实际表示的是执行一条语句不会对程序状态产生变化，如图 6.9 所示。

若我们只需要分析 x 的赋值问题，则，当我们执行 s: y=b 时，指令 s 既没有使得某一个 x 的赋值关系生效，也没有使得某一个 x 的赋值关系失效，则 $f_s(IN(s))=Ø \cup (IN[s] - Ø)$，即只要 gen 和 kill 均为空集，即存在单元函数，使得 I(x)=x。

对于函数的封闭性，设我们有两个传递函数：

$f(x)=G_1 \cup (x-K_1)$

$g(x)=G_2 \cup (x-K_2)$

则：

$h(x)=g(f(x))=G_2 \cup (G_1 \cup (x-K_1)-K_2)$

其等价于：

$(G_2 \cup (G_1-K_1)) \cup (x-(K_1 \cup K_2))$

使 $G=G_2 \cup (G_1-K_1)$，$K=K_1 \cup K_2$，则 $h(x)=G \cup (x-K)$，$h(x) \in F$。

即基本块内部的传递函数，其可在框架下反映程序状态的 gen-kill 过程。

控制流带来的程序状态的改变被称为控制流约束函数，对于不同的数据流分析模式，我们需要设定不同的控制流约束函数，进行程序状态的合并，因此，在状态合并时分为两种情况：

May 分析与 Must 分析

May 分析用于分析在某一程序点上所有可能存在的程序状态。例如，对于到达定值分析，我们想要知道在某一程序点 p 上变量的赋值情况，那么，我们使用 May 分析，对所有包含程序点 p

的路径进行分析。

Must 分析用于分析在某一程序点上一定存在的程序状态。例如，对于可用表达式分析，我们想要知道在某一程序点 p 上，已被求值的表达式的情况，那么，对于经过程序点 p 上的所有路径，该表达式均已被求值，且构成表达式的变量均未被赋值，那么在该程序点上表达式才是可用的。

在幂集上∪运算与∩运算的单调性已在上文证明，因此此处不再赘述。后文所述的数据流分析算法就是基于设计的传递函数与控制流约束函数，迭代地计算程序状态，直到程序状态不再改变为止。

在分析结束之后，我们需要关注数据流分析的一些性质。

数据流分析敏感性

因现代软件代码规模的急剧增长，使用数据流分析技术进行程序状态的跟踪与计算时，常常需要进行精度与时空成本之间的博弈，在追求较小的时间与空间成本进行程序状态的分析时往往也意味着较低的分析精度。我们用对不同情况的敏感性来描述我们采取的分析方法的预期精度。

（1）**流敏感与流不敏感（Flow-Sensitive/Insensitive）：**程序内部数据随着程序的执行顺序流动，流敏感的分析会根据程序的执行顺序，跟踪程序状态的变化，而流不敏感的分析通过代码扫描报告所有可能出现的情况。

```
x=1;
x=2;
```

对于以上的两条语句，使用流敏感的分析方法，分析结果会告诉我们：在执行第一条语句后，x 被赋值为 1，在执行第二条语句之后，x 被赋值为 2。使用流不敏感的分析方法，分析结果会告诉我们：x 的值可能为 1 或 2。

（2）**路径敏感与路径不敏感（Path-Sensitive/Insensitive）：**路径敏感分析与路径不敏感分析的区别在于：路径敏感分析将构造的程序**控制流图（Control-Flow Graph）**扩展为**扩展图（Exploded Graph）**，跟踪程序内部所有可能路径，分析程序状态变化，而路径不敏感分析通常只基于控制流图进行分析。

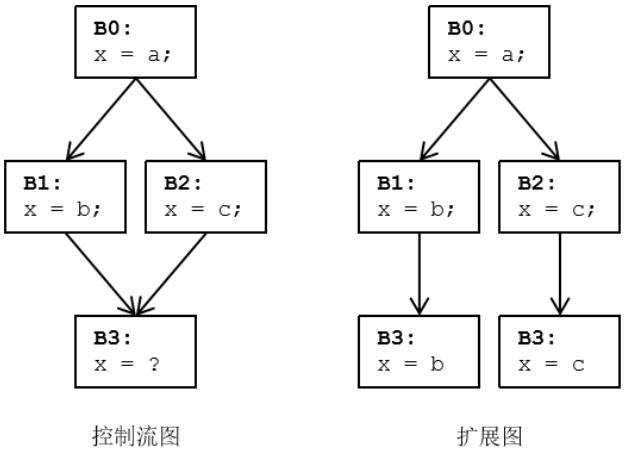


图 6.10 控制流图与扩展图

以图 6.10 为例，当使用路径不敏感的分析时，分析结果会告诉我们：x 的值可能为 b 或 c，而基于扩展图进行分析时，能够直观展现在不同的执行路径上对 x 的值的改变。虽然路径敏感的分析精度更高，但是可能存在**路径爆炸（Path Explosion）**的问题。例如，对于循环的分析，每次执行循环都会多出大量的可执行路径，极大地增加了分析成本。

（3）**上下文敏感与上下文不敏感（Context-Sensitive/Insensitive）：**上下文敏感性与函数调用相关。上下文敏感的分析方法关注在函数调用时调用点的程序状态，而上下文不敏感的分析方法不考虑函数调用点的信息。

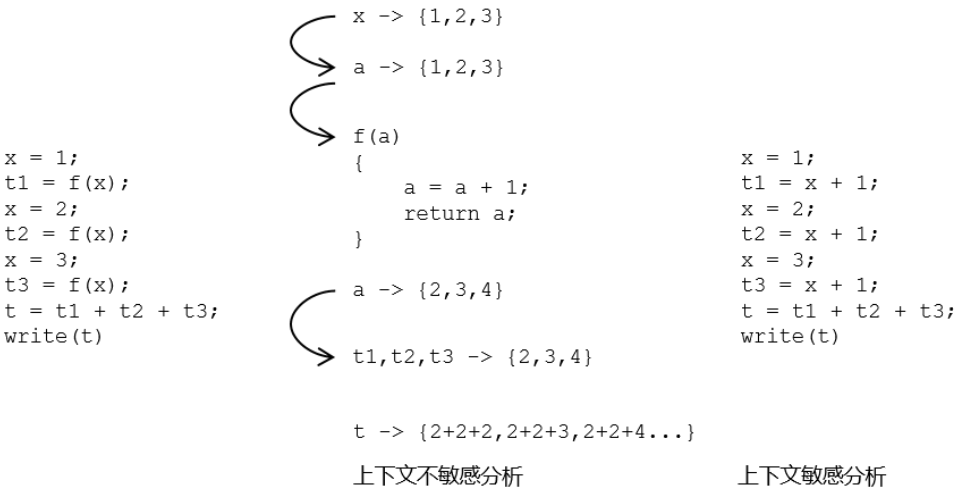


图 6.11 上下文敏感性

以图 6.11 为例，在上下文不敏感分析中，输入值存在三种可能性：1，2，3，从而对于函数的返回值也有三种可能：2，3，4，于是 t 的最终取值可能为 6 到 12 中的任何值。对于上下文敏感分析，使用函数内联的形式进行体现，能够精准得出 t 值为 2+3+4=9。

本章中介绍的数据流分析模式是流敏感、路径不敏感且上下文不敏感的，路径敏感采取的符号执行技术与上下文敏感中采取的过程间数据流分析等，请学有余力的读者自行查阅相关材料。

数据流分析准确性

在进行数据流分析时，我们有时会需要知道能否获得想要的结果。如果我们设计了一个优化算法，改变了代码的语义，消除了不应消除的语句，会直接改变程序执行的结果，使得执行不符合预期，这样就违反了我们优化的初衷。如果我们设计的优化算法，对程序没有进行任何优化，那么，虽然程序语义没有改变，我们的优化算法也失去了意义。

那么我们选择的分析方式能够精确地去除冗余代码（或尽可能去除冗余代码），降低程序开销，而不改变程序语义吗？

在后文所述的数据流分析模式中，我们使用**迭代算法（Iterative Algorithm）**或**工作表算法（Worklist Algorithm）**计算程序状态的变化。通过以下三种分析方式的比较，我们简单地衡量所使用的算法的准确性：

IDEAL Solution=合并所有可执行的路径上的程序状态

MOP（Meet Over All Paths）=合并所有路径上的程序状态

MFP（Maximal Fixed Point）=迭代算法的结果

我们定义优化结果的五种状态：

safe optimization: 优化后不改变程序语义

unsafe optimization: 优化后改变程序语义

truth: 优化掉所有冗余代码，优化后程序开销最小且语义不变

optimize nothing: 不优化任何代码

optimize everything: 优化任何代码

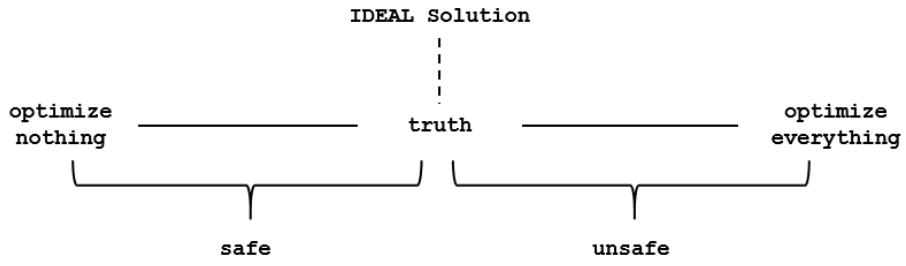


图 6.12 优化结果

而易见，IDEAL Solution 是理想化的分析解，是对实际运行过程中的可执行路径上的程序状态的分析，其分析结果为 truth。

那么，我们首先把 MOP 的结果与 IDEAL Solution 比较。图 6.13 使用公共子表达式消除举例，使用 Must 分析对其进行分析。

对于图 6.13 所示的这段代码，在程序实际执行的过程中，B2-B3-B5 这条路径是不可执行的。但是，如果我们在对代码进行常量传播分析之前，先进行公共子表达式消除，分析基本块 B5 中的表达式 x-y 是否已被计算过，因不可执行路径（B2-B3-B5）上不存在对该表达式的计算，因此不认为 x-y 在基本块 B5 中可以作为公共子表达式消除。

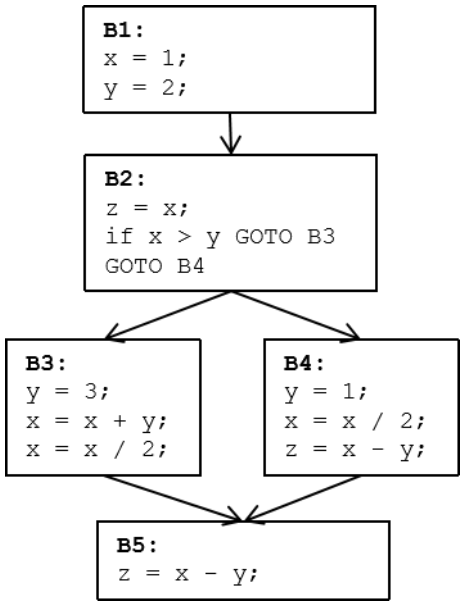


图 6.13 不可达路径的示例

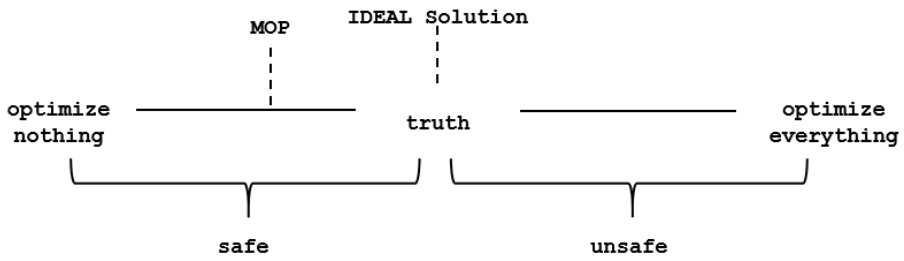


图 6.14 MOP 与 IDEAL Solution 的比较

因此，IDEAL Solution 中应当消除 x-y 这一公共子表达式，而相比 IDEAL Solution 而言，MOP 分析了更多路径上的程序信息，使得 x-y 没有被消除，因此 MOP 的分析结果比 IDEAL Solution 更为保守。

然后，对于 MOP 与 MFP 的比较，我们使用一个常量传播的例子解释其中的差别：

对于图 6.15 所示的这段代码，MOP 跟踪不同的执行路径并进行计算，对于两条路径，z 均为一定值 10，因此可以使用常量进行替换，而 MFP 在基本块的入口使用控制流约束函数，合并不同前驱的程序状态，当判断 z 是否是常量时，x 的可能取值有两种，y 的可能取值也有两种，那么

当进行常量传播分析时，不能将 z 转化为常量值 10。

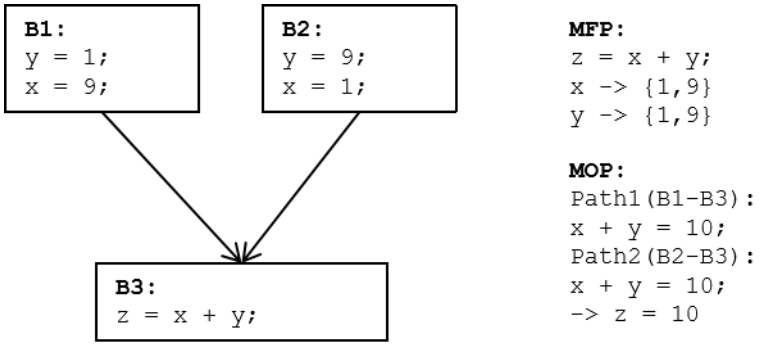


图 6.15 MOP 与 MFP 的差别比较的示例

使用前文所述的敏感性来表示两种方法的不同，MOP 是路径敏感的分析，而 MFP 是路径不敏感的，因此使用 MFP 的方式进行分析，其结果比 MOP 的更为保守。

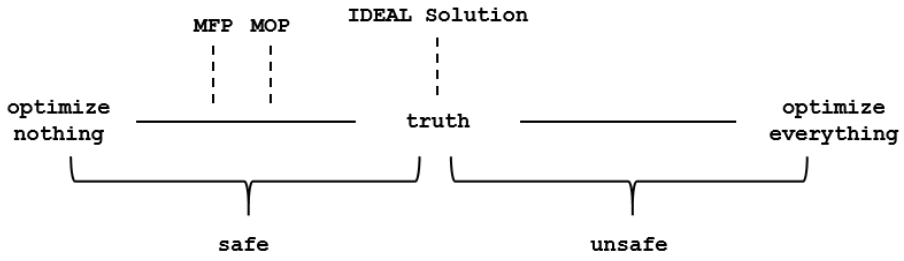


图 6.16 MFP 与 MOP 的比较

我们分析的目的，就是一定要达到 IDEAL Solution。然而，任何路径敏感的分析，都无法避免地会遇到上文所述的路径爆炸问题。同时，进行路径敏感的分析，需要记录所有可能路径上程序状态，分析时的时空成本也极大，不适于在大规模的程序上分析程序状态。

我们将要学习的数据流分析模式是**过近似（Over-Approximate）**的、**追求保守解（Sound Static Analysis）**的，这代表分析结果得出的优化策略，在优化之后不会改变程序的语义的同时，对于我们的分析模式，也不能确保分析的结果是最好的。

因此，我们使用的分析，无法保证所有可被消除的代码都已被消除。实际上，根据**莱斯定理（Rice's Theorem）**，不存在通用、高效的算法，使得优化达到最好的效果，即没有任何一种优化

的算法是最好的。

在实践中，我们需要针对不同的优化目的，对程序进行抽象，构造程序状态传递函数，描述并且记录程序状态的变化情况，基于获取的状态信息，完成优化工作。

6.1.3 到达定值分析

在数据流分析中，我们通常需要了解，数据在哪里被定义，在哪里被使用。在使用时，当前使用的变量是否已经被定义？使用未被定义的变量会诱发“未定义的引用”问题。当前使用的变量是否是一个定值？若是，那么在编译时就可以用一个定值代替这个变量。是否存在定义而从未被使用的情况？若有，该定义是“无用”的。在局部优化中，我们已经在基本块内部探究了变量的 define-use 关系，而数据流分析则试图揭示多个基本块乃至多个函数之间变量的 define-use 关系。

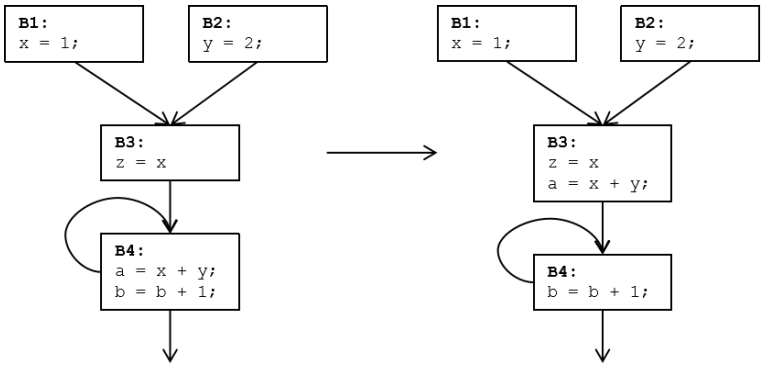


图 6.17 循环不变代码外提

到达定值（Reaching Definition）是最基础的数据流分析模式之一，描述了在程序内部的每个程序点上变量可能的赋值情况，是与程序内部的变量的 define-use 关系最相关且最简单的分析模式。

到达定值分析的主要用途有以下几项：

循环不变代码外提（Loop-Invariant Code Motion，LICM）：

对于循环内部的赋值语句，若构成赋值表达式的变量均在循环外部定义，则可以将该语句移动到循环外侧，降低执行时的开销（如图 6.17 所示）。

常量折叠（Constant Folding）：

我们可以迭代式地将程序内部可转变为常量的变量转化为常量，降低执行的开销。

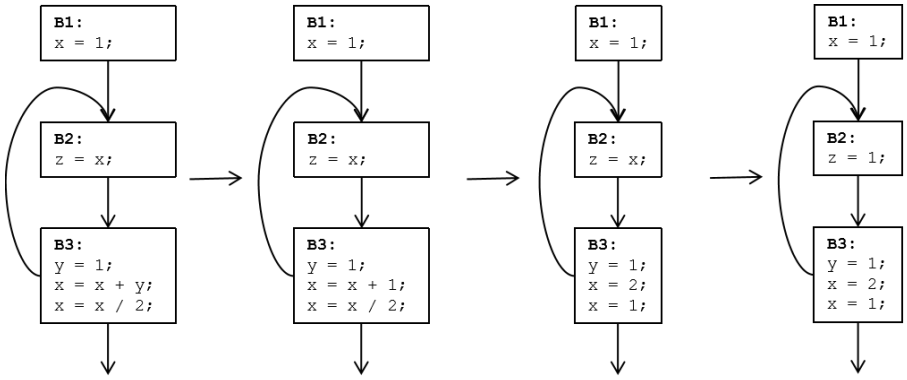


图 6.18 常量传播

对于图 6.18 这段代码中的变量 x，y，z，使用到达定值模式和数据流分析算法进行分析，可以将其中的所有变量都替换为常量。

到达定值模式的算法

我们首先要选择应当使用 May 分析还是 Must 分析进行程序状态的分析。对于到达定值问题，为了分析在某一程序点 p 上的变量 v，对于所有经过程序点 p 的路径，变量 v 的值是否恒为一常量，即，我们需先获取 v 在程序点 p 上所有可能的值的情况，再判断所有可能值是否均为一相同定值，为此，我们使用 May 分析的方式进行程序状态的分析。

然后，我们要构造针对到达定值问题的传递函数与控制流约束函数。前文我们简要地介绍了程序状态的传递函数，接下来我们深入地剖析一下传递函数各部分的语义，以便于我们对传递函数的理解。

传递函数用于描述程序内部的状态变化，那么，对于到达定值问题，我们只需关注其中那些针对变量的赋值语句。对于程序内部的赋值语句 s，我们将其状态转换的传递函数定义为：

$f_s=gen_s \cup (IN[s] - kill_s)$

我们将该传递函数运用到如下的程序中，描述程序状态的变化：

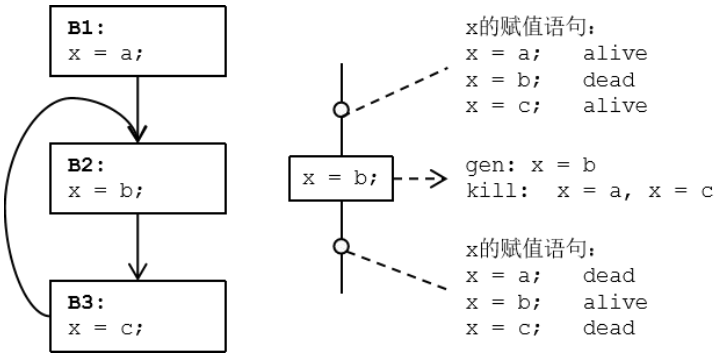


图 6.19 程序状态变化

程序内部对变量 x 的赋值语句共有 3 条：x=a，x=b，x=c，在基本块 B2 的入口，x 的取值有两种可能性：x→{a,c}。待分析的基本块 B2 中的语句 s: x=b，因对变量 x 进行了重新赋值，之前对 x 的赋值全部失效了，于是，我们认为，对变量 x 的赋值语句 x=b，“生成”了赋值情况 x→b，“杀死”了其他所有对 x 的赋值情况。

那么传递函数中 gen_s，IN[s]，$kill_s$ 分别为：

gen_s：语句 x=b 中，对于被赋值的变量 x，生成了赋值情况 x→b。

IN[s]：执行语句之前所有变量可能的赋值情况。

$kill_s$：执行语句之后，对于被赋值的变量 x，杀死了 x 当前所有的赋值情况。

则传递函数 $f_s=gen_s \cup (IN[s]-kill_s)$的语义为，对于待分析的赋值语句 s，执行语句之前变量赋值关系记为 IN[s]，若其对变量 x 进行赋值，则先使得 x 的所有赋值关系失效$(IN[s]-kill_s)$，然后生成当前赋值语句对应的赋值关系 gen_s，最后将 gen_s 加入到程序状态中。

对于控制流约束函数而言，到达定值模式描述的是在某一程序点上的变量 x 是否可能恒为一定值，问题可被转化为：x 的所有可能存在的赋值情况是否相同。为此，我们通常将控制流约束函数构造为如下的形式：

$IN[B]=\cup_{P\text{是}B\text{的一个前驱}} OUT[P]$

我们继续将该控制流约束函数运用到图 6.20 的例子中，描述程序状态变化。

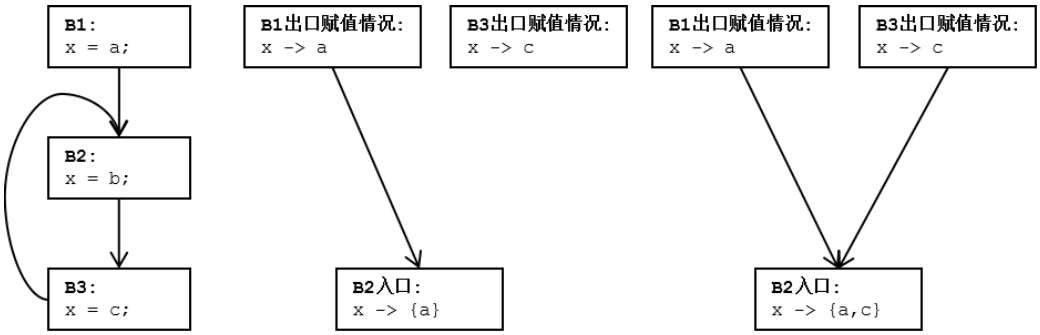


图 6.20 状态合并

基本块出口变量 x 的赋值情况集合的语义为：对于集合内的任一值 d，存在至少一条路径，该路径在基本块的出口位置，x 的赋值情况为 x→d。

B2 的前驱基本块共有两个：基本块 B1 与 B3，那么，要分析在基本块 B2 入口处 x 可能的赋值情况，需要合并两个前驱基本块 B1 和 B3 在出口处的程序状态。程序内部路径的跳转，可能从 B1 跳转到 B2，也可能从 B3 跳转到 B2，因此 B2 入口处 x 的赋值集合，应当为 B2 所有前驱基本块出口位置 x 赋值的并集。

构造了传递函数与控制流约束函数，接下来我们应当基于函数构造算法，计算我们想要的分析结果。我们用一个简单的程序举例，对其进行到达定值分析。

迭代算法与不动点：

迭代算法是迭代式地进行程序状态计算，直到程序状态到达不动点的算法。先前我们进行了 MFP、MOP、IDEAL Solution 方法的比较，我们知道，迭代算法的结果较之 MOP 与 IDEAL Solution，其获得的程序状态更为保守。

迭代算法的思路是：根据一定的顺序，对程序内部的基本块进行遍历，基于传递函数与控制流约束函数，进行程序状态的改变与记录，当程序状态到达不动点时，迭代算法结束，给出程序内部每个程序点上的程序状态情况。

描述到达定值的迭代算法如下：

OUT[ENTRY]=Ø;

for(除 ENTRY 之外的每个基本块 B) OUT[B]=Ø;

while(某个 OUT 值发生了改变)

for(除 ENTRY 之外的每个基本块 B){

IN[B]=$\cup_{P\text{是}B\text{的一个前驱}}$OUT[P];

OUT[B]=$gen_B \cup (IN[B]-kill_B)$;

}

其中，ENTRY 表示程序的入口，算法基于传递函数与控制流约束函数，对程序进行遍历，计算程序状态，直到程序内部的每个基本块在出口时的程序状态均不再改变时停止。

在运用迭代算法计算到达定值之前，我们首先需要关注算法的以下几个性质：构造的迭代算法一定能到达一个不动点吗？这关乎算法能不能停止，并且得到想要的结果。

为了解释这一问题，我们使用计算到达定值的迭代算法作示例。

程序状态的有界性：

图 6.21 中的这段代码，总共有 9 条赋值语句，与赋值语句相对应的赋值情况也有 9 条，每一条赋值情况有两种可能状态：alive 和 dead，如果用一个一维数组来表示每个赋值情况的状态，用 0 代表 dead，用 1 代表 alive，那么(0,0,0,0,0,0,0,0,0)表示 9 条赋值情况均失效，(1,1,1,1,1,1,1,1,1)表示 9 条赋值情况均生效，每一个程序点上的状态，至多有 2^9 种情况，程序的状态数量是有界的。

程序状态的单调性：

对于到达定值问题，一程序点 p 上对应的程序状态中的一条赋值情况 x→a，其语义为：存在至少一条路径，使得到达该程序点 p 时，x 的值为 a，使用上文所述的一维数组来表示，即在程序点 p 上，x→a 的状态为 1。那么，在迭代遍历的过程中，若当次分析完毕后，某一程序点 p 上该赋值情况的状态为 1，在之后的每一次迭代中，该赋值情况恒为 1（存在路径使得赋值情况生效），因此，对于每一条赋值情况，状态的改变仅可能由 0 变为 1，状态的变化是单调的。

因单调而有界，我们的迭代算法最终会到达一不动点，并给出分析结果。

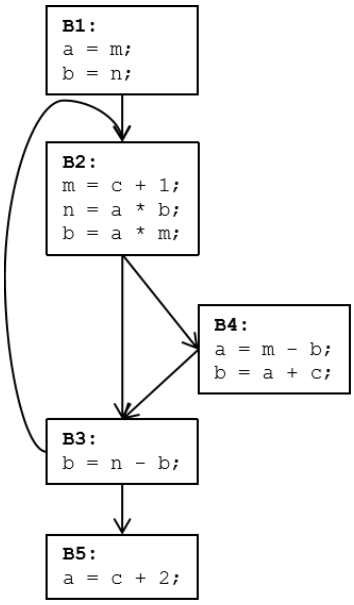


图 6.21 到达定值模式所使用的程序片段示例

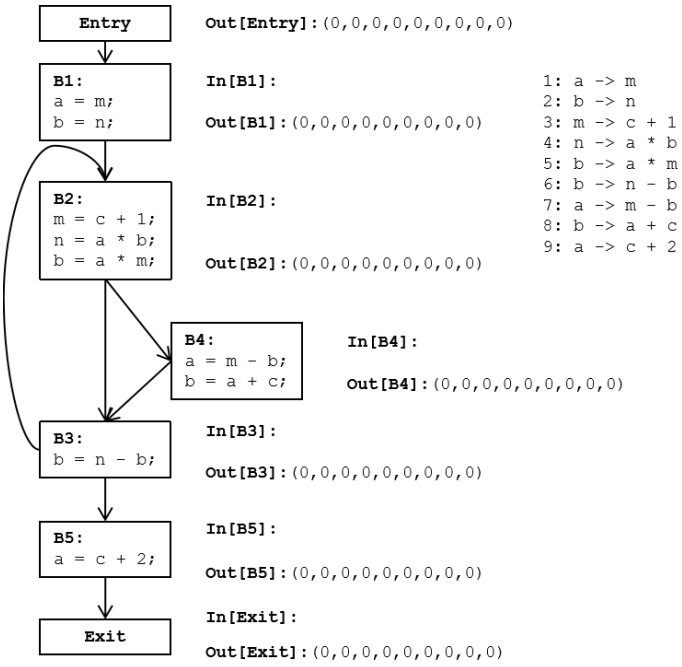


图 6.22 到达定值模式初始化

迭代算法计算到达定值：

了解了迭代算法的相关性质，我们接下来用一个例子说明迭代算法进行程序状态计算的整个过程。我们使用上文所述的一维数组来表示程序状态的情况。因我们进行到达定值计算，每个程序点上的程序状态的语义为：对于状态为 1 的赋值情况，存在至少一条经过该程序点的路径，使得该赋值情况生效。因此，在进行分析之前，将程序内的赋值情况均初始化为 0，如图 6.22 所示。

此处程序的遍历顺序为：B1，B2，B4，B3，B5。在遍历的过程中，使用传递函数与控制流约束函数进行程序状态的计算，以下是运用传递函数与控制流约束函数的示例：

图 6.23 中，IN[s1]与 OUT[s1]表示在执行语句 s1 之前和之后的程序状态。在执行 s1 之前，赋值情况 1、3、4、5 生效，s1 语句对变量 a 赋值，首先杀死所有变量 a 的赋值情况（1、9），然后生成 s1 对应的赋值情况 7，因此，在执行 s1 后，赋值情况 3、4、5、7 生效。

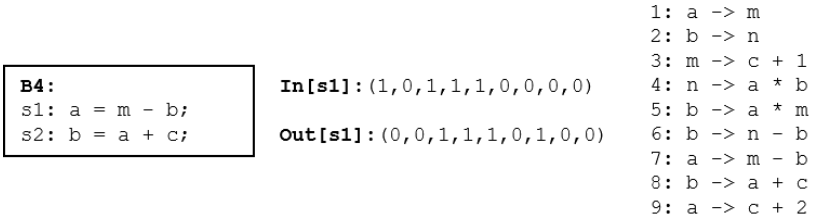


图 6.23 状态传递示例

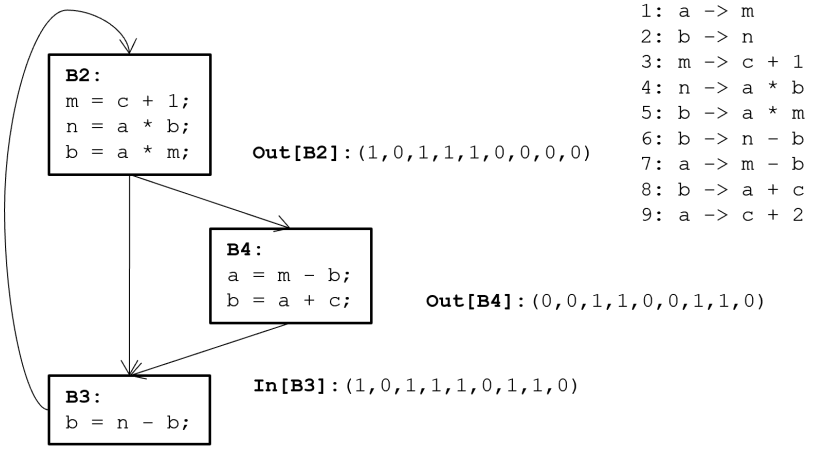


图 6.24 状态合并示例

IN[B3]表示在基本块 B3 入口处的程序状态。到达定值模式需考察一个程序点上所有可能生效

的赋值情况，所以在基本块入口处采用∪运算合并程序状态。对于基本块 B3，其有两个前驱基本块 B2 和 B4，由 B2 向 B3 传递的程序状态为（1,0,1,1,1,0,0,0,0），由 B4 向 B3 传递的程序状态为（0,0,1,1,0,0,1,1,0），使用 OR 运算合并程序状态，得出在 B3 入口处，程序状态为（1,0,1,1,1,0,1,1,0）。

那么，第一次迭代后记录的程序状态如图 6.25 所示。根据迭代算法，每一次迭代后若有基本块的 OUT 发生了变化，则进行下一次迭代。

第二次迭代如图 6.26 所示。第二次迭代后，仍有基本块的 OUT 发生了变化，因此进行第三次迭代。

在第三次迭代后，相较于第二次迭代的结果，没有任何基本块的 OUT 值发生了改变。因此，迭代算法结束，输出分析结果。从记录的程序状态中，能够获取我们想要的程序性质，例如，对于基本块 B4 中的语句 a=m-b，赋值语句右侧的表达式由变量 m 与 b 构成，对于变量 m，生效的赋值关系为 m→c+1，对于变量 b，生效的赋值关系为 b→a*m，若均为常量，则可将变量 a 也用一常量替换。

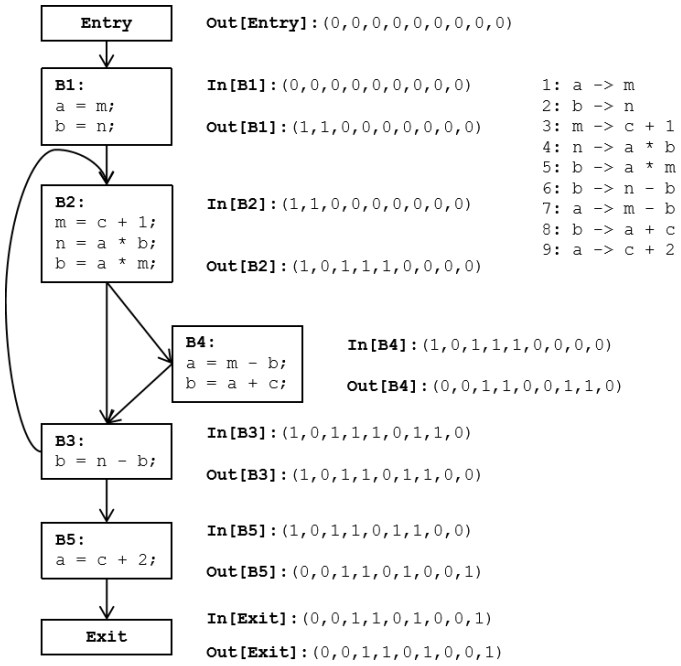


图 6.25 第一次迭代结果

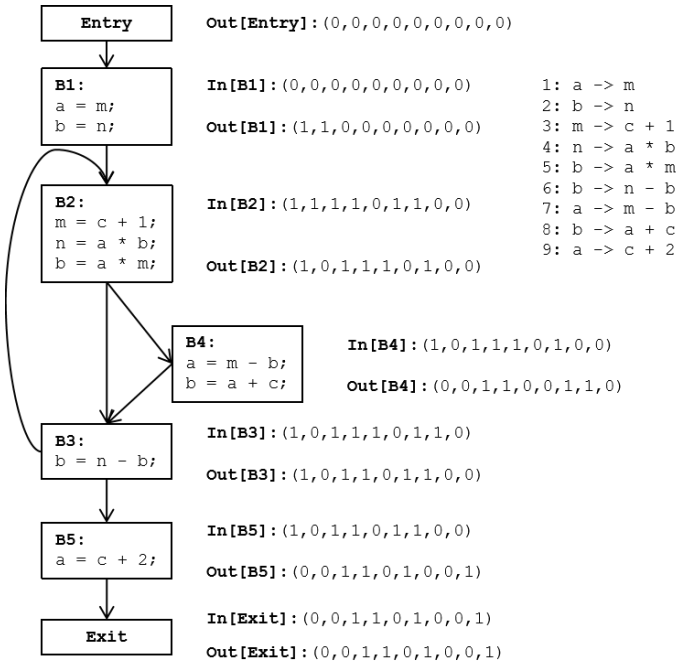


图 6.26 第二次迭代结果

为了去除算法内部的部分冗余计算，我们将迭代算法修改为**工作表算法（Worklist Algorithm）**，使用一个 Worklist 记录需要进行状态计算的基本块，算法修改如下：

OUT[ENTRY]=Ø;

for(除 ENTRY 之外的每个基本块 B) OUT[B]=Ø;

Worklist←所有的基本块;

while(Worklist 非空){

从 Worklist 中选择一个基本块 B

OLD_OUT=OUT[B];

$IN[B]=\cup_{P\text{ 是 }B\text{ 的一个前驱}}OUT[P]$;

$OUT[B]=gen_B \cup (IN[B]-kill_B)$;

if (OLD_OUT≠OUT[B])

把基本块 B 的所有后继加入到 Worklist 中

}

相较于迭代算法，工作表算法多了添加基本块进入工作表的语句，当一个基本块的输出发生改变时，需要将其后继加入到工作表中，进行程序状态的更新与计算，算法的停止条件是工作表为空。

相比迭代算法遍历程序的每个基本块，工作表算法第一次遍历计算了基本块 B1、B2、B3、B4、B5，第二次计算了基本块 B2、B3、B4、B5，第三次仅需计算 B3 与 B4 之后，算法终止，输出结果。

在实践中，我们需要合理地使用到达定值模式，进行部分优化任务的分析与计算。

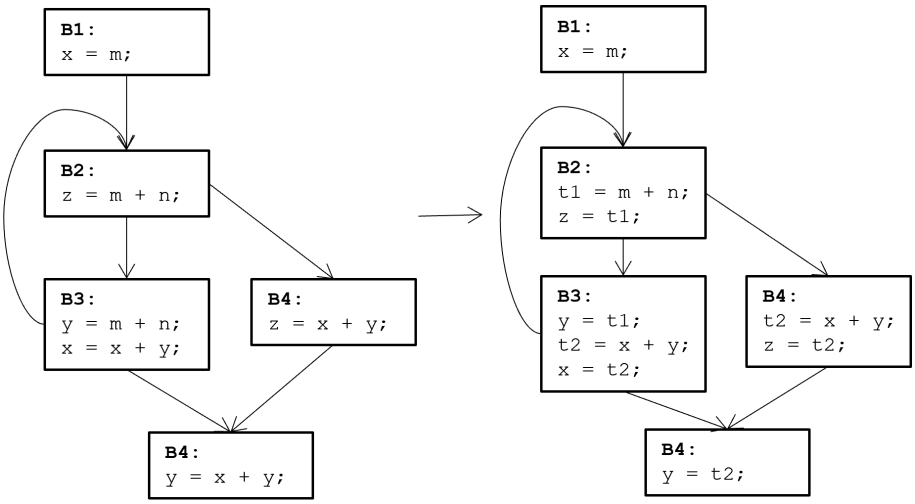


图 6.27 公共子表达式消除

6.1.4 可用表达式分析

可用表达式模式（Available Expression）关注程序内部的表达式是否在某些程序点可用。表达式可能由多个变量构成，表达式的计算过程包含了变量的 use，表达式处于赋值语句右侧，包含了变量的 define，与程序内部的变量的 define-use 关系相关，较上文提到的到达定值问题更为复杂一些。

在程序优化的过程中，可用表达式模式通常用于进行**公共子表达式消除（Common Subexpression Elimination）**。通过可用表达式模式对函数内部进行分析，消除重复计算的公共子表达式，如图 6.27 所示。表达式 m+n 与 x+y，在程序内部被多次计算，因此可以使用 t1 与 t2 代替其求值的结果，简化表达式的计算。

可用表达式模式的算法

与到达定值模式相同，首先我们确定应当使用 May 分析还是 Must 分析进行程序状态的分析与计算。

可用表达式问题，用于分析在某一程序点 p 上的表达式 e，对于所有经过程序点 p 的路径，表达式是否均已被计算过，且构成表达式的变量在之后未被重新定义，即，我们需判断所有经过 p 的路径，在程序点 p 上该表达式均存活。为此，我们使用 Must 分析的方式进行程序状态的分析。

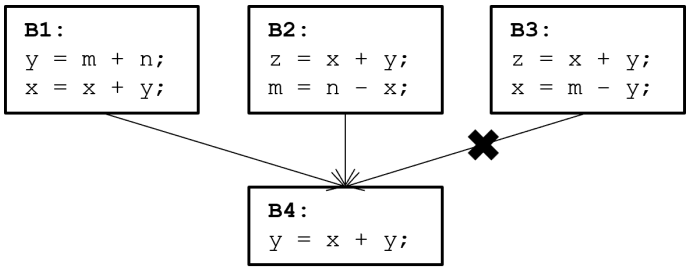


图 6.28 可用表达式

例如，对于图 6.28 中的这段代码，在基本块 B3 中对 x+y 进行了计算，但是之后对变量 x 的赋值使得表达式的计算结果失效，因此认为在 B3 出口处，该表达式不存活。

然后，我们要构造针对可用表达式问题的传递函数与控制流约束函数。

对于可用表达式问题，我们需要关注其中的表达式与构成表达式的变量的赋值语句。对于语句 s，其状态转换的传递函数定义为：

$f_s=e\_gen_s \cup (IN[s]-e\_kill_s)$

我们将该传递函数运用到图 6.29 的程序中，描述程序状态的变化。

在执行语句 x=n+y 之前，程序内部的可用子表达式共有两个：m+n 与 m+x，语句 x=n+y 生成了子表达式 n+y，同时对 x 重新进行赋值。因表达式 m+x 中存在对 x 的使用，对 x 的重新赋值使得 m+x 的求值失效。于是，我们认为，语句 x=n+y，“生成”了可用子表达式 n+y，“杀死”了可用子表达式 m+x。

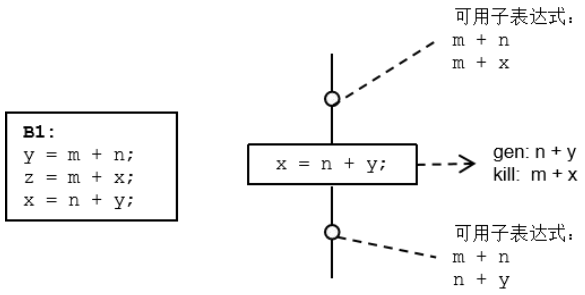
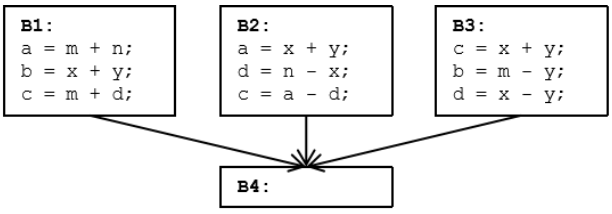


图 6.29 程序状态变化




```
B4_IN = B1_OUT ∩ B2_OUT ∩ B3_OUT
      = {m + n, x + y, m + d} ∩ {x + y, n - x, a - d} ∩
        {x + y, n - x, a - d}
      = {x + y}
```

图 6.30 状态合并

那么传递函数中 e_gen_s，IN[s]，e_kill_s分别为：

e_gen_s：语句 x=n+y 中，生成了可用子表达式 n＋y。

IN[s]：执行语句之前所有可用子表达式。

e_kill_s：执行语句之后，杀死的可用子表达式。

则传递函数 $f_s=e_gen_s \cup (IN[s]-e_kill_s)$的语义为，对于待分析的赋值语句 s，执行语句之前存在的子表达式情况记为 IN[s]，若其对某变量 x 进行赋值，则先使得包含 x 的所有子表达式失效$(IN[s]-e_kill_s)$，然后生成当前语句对应的子表达式 e_gen_s，最后将 e_gen_s加入到程序状态中。

对于控制流约束函数而言，可用子表达式模式描述的是在某一程序点上是否在任何情况下某一子表达式 e 均生效。因其为 Must 分析，我们通常将控制流约束函数构造为如下的形式：

$IN[B]=\cap_{P\text{是}B\text{的一个前驱}}OUT[P]$

我们将该控制流约束函数运用到图 6.30 的例子中，描述程序状态变化。

B4 的前驱基本块共有三个：基本块 B1、B2 与 B3，那么，要分析在基本块 B4 入口处生效的子表达式，需要合并前驱基本块在出口处的程序状态。要保证子表达式在基本块 B4 入口仍生效，需确保所有的前驱基本块的出口处该子表达式生效。因此，可用表达式的控制流约束函数，使用交运算进行程序状态的合并。

构造了传递函数与控制流约束函数，接下来我们应当基于函数构造算法，计算我们想要的分析结果。

```
OUT[ENTRY]=Ø;
for(除 ENTRY 之外的每个基本块 B) OUT[B]=U;
while(某个 OUT 值发生了改变){
    for(除 ENTRY 之外的每个基本块 B){
        IN[B]=∩P 是B的一个前驱 OUT[P];
        OUT[B]=e_genB∪(IN[B]-e_killB);
    }
}
```

我们用一个简单的程序举例，对其进行可用表达式分析。

因使用 Must 分析方式，只要有一个前驱节点不包含子表达式，在基本块的入口该表达式即处于失效状态，因此我们在进行分析之前，将程序内基本块（除 Entry 外）出口处子表达式生效情况均初始化为 1，如图 6.31 所示。

因可用表达式的有效关系与程序执行顺序一致，因此为前向数据流分析。

那么，第一次迭代后记录的程序状态如图 6.32 所示。第一次迭代后，基本块的 OUT 值发生了变化，因此进行第二次迭代。第二次迭代结果如图 6.33 所示。

在第三次迭代后，每个基本块的 OUT 值均没有改变，因此迭代算法就此终止。

在实践中，我们可以选用迭代算法，也可以使用 6.1.3 节中所示的工作流算法，实现优化算法。

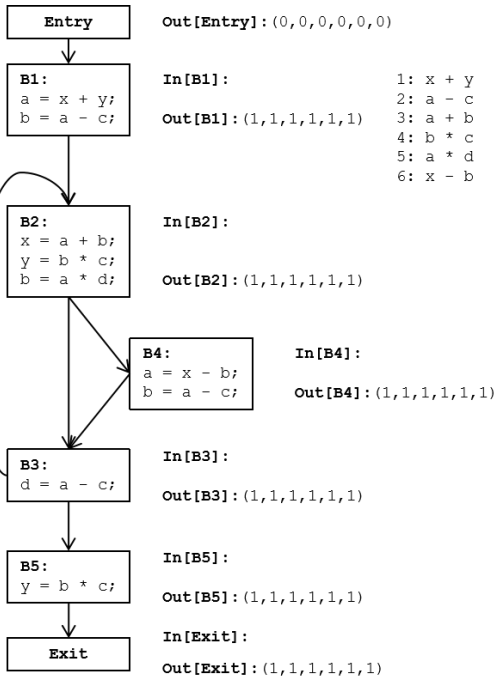


图 6.31 可用表达式模式初始化

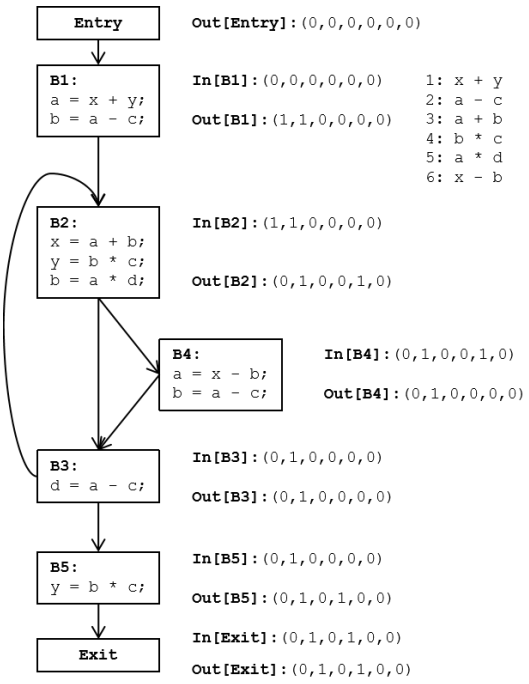


图 6.32 第一次迭代结果

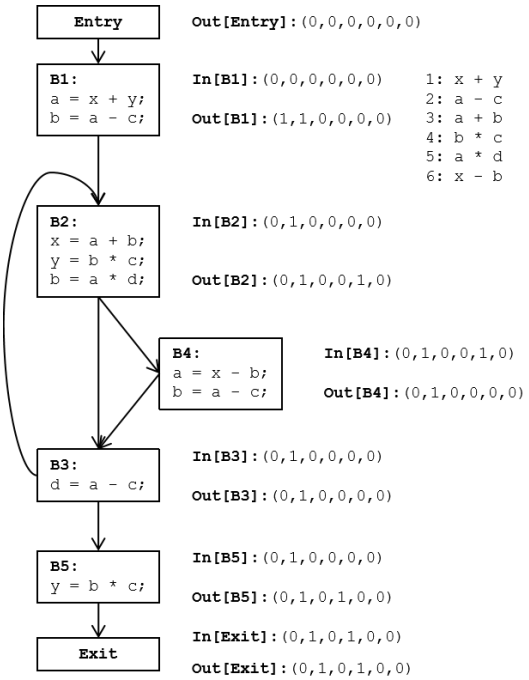


图 6.33 第二次迭代结果

6.1.5 活跃变量分析

活跃变量分析（Live-Variable Analysis）关注程序内部程序点 p 上对于某变量的定义，是否会在某条由 p 出发的路径上被使用。活跃变量分析与程序内部的变量的 define-use 关系相关，当变量被 define 后未被 use，该 define 语句被认为是无用的。

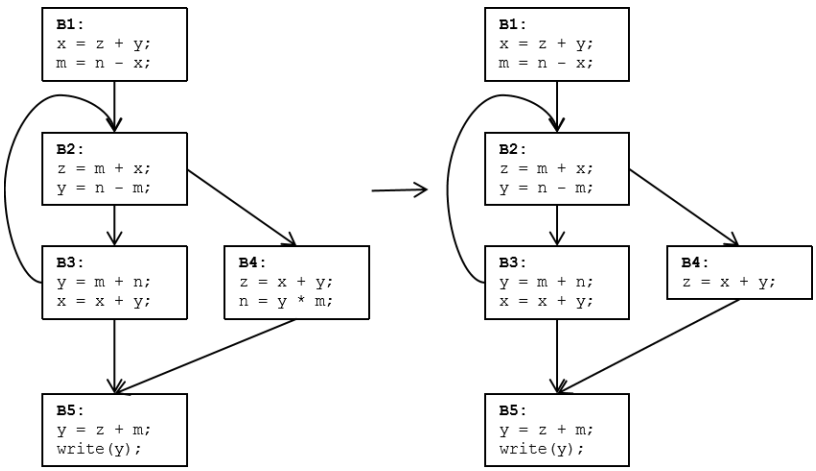


图 6.34 无用代码消除

在程序优化的过程中，活跃变量模式通常用于进行**无用代码消除（Dead Code Elimination）**。通过活跃变量模式对函数内部进行分析，消除未被使用的赋值与未执行的程序代码，如图 6.34 的例子所示。

基本块 B4 中的语句 n=y*m，由于变量 n 的 define 在之后的程序中未被 use，这个 define 语句是无用的，可消除此处对于变量 n 的赋值，优化程序代码。

活跃变量模式的算法

首先我们确定应当使用 May 分析还是 Must 分析进行程序状态的分析与计算。

活跃变量问题，用于分析在某一程序点 p1 上对变量 v 的赋值，是否在之后的程序点 p2 上被使用，且在 p1 到 p2 的路径上没有对该变量 v 的重定义，即，我们需判断所有经过 p1 的路径，是否存在一条路径使得变量 v 在之后被使用。为此，我们使用 May 分析的方式进行程序状态的分析。

然后，我们要构造针对活跃变量问题的传递函数与控制流约束函数。

活跃变量的遍历方式与之前的到达定值与可用表达式的遍历有所不同：到达定值问题与可用表达式问题按程序执行的顺序进行遍历（前序遍历），而活跃变量问题按程序执行的逆向对程序进行遍历（后序遍历）。

到达定值模式分析某一程序点 p 上对某一变量 x 的所有可能赋值情况，对应多条经过 p 的路径上前驱基本块中对变量 x 的赋值语句，如图 6.35 所示。

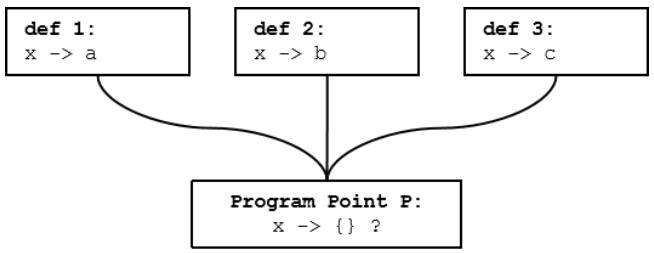


图 6.35 到达定值模式

与到达定值模式相同的是，可用子表达式模式分析在某一程序点 p 上是否存在可用的子表达式，子表达式在多条经过程序点 p 的前驱基本块内部被计算，算法分析在程序点 p 处被计算的子表达式是否还生效，如图 6.36 所示。

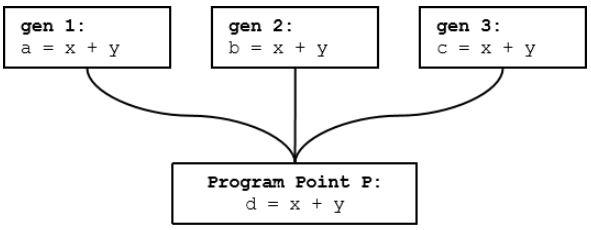


图 6.36 可用表达式模式

然而，对于活跃变量分析，虽然也是与变量的 define-use 相关的分析，但其分析针对的是某次 define 是否在之后的某个程序点上被使用，如图 6.37 所示。

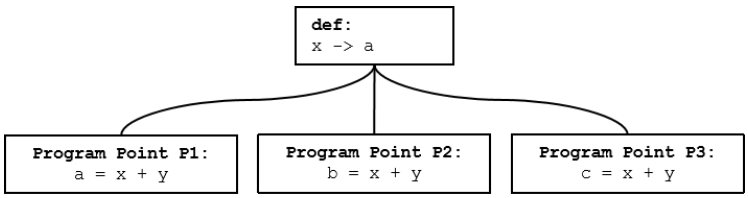


图 6.37 活跃变量模式

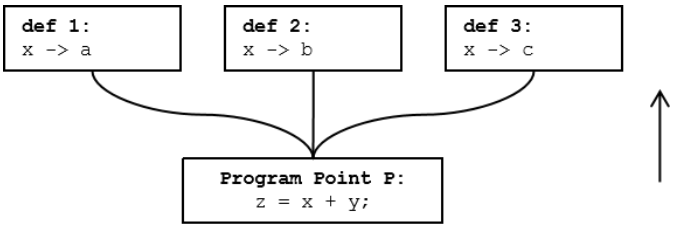


图 6.38 后向数据流分析

对变量 x 的一次定义 def: x→a，可能在后继基本块中的多个程序点 p1、p2、p3 等被使用，而多个定义可能在同一个程序点上被使用，因此使用前序遍历较为复杂。所以，我们使用后序遍历的方式进行数据流分析，将问题转化为如图 6.38 的情况。

程序点 p 上的语句 z=x+y，等号右侧的表达式使用到了变量 x（use），这使得先前的某一次定义（def）有了用武之地，我们分析这次 use 使得哪些前驱基本块内部对于变量 x 的 def 不是无用代码。我们使用后序遍历的方式，因此我们构造的传递函数与控制流约束函数较之前两种分析模式也有一些不同。

对于活跃变量问题，我们需要关注变量的定义（def）与使用（use）语句。对于程序内部的语句 s，其状态转换的传递函数定义为：

$f_s=use_s \cup (OUT[s]-def_s)$

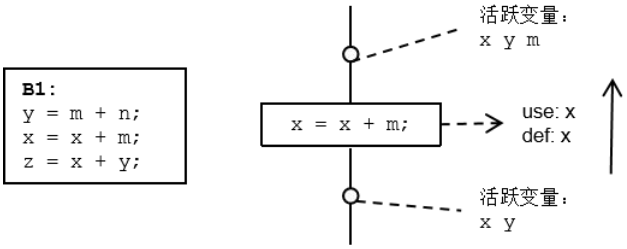


图 6.39 程序状态变化

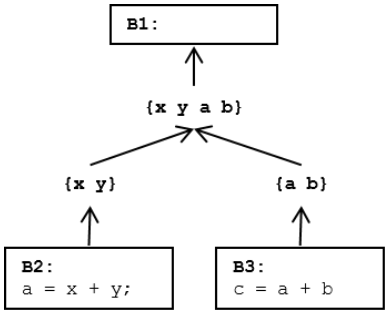


图 6.40 状态合并

我们将该传递函数运用到图 6.39 的程序中，描述程序状态的变化。

语句 x=x+m 中对 x 进行了定义，并在语句 z=x+y 中，该定义被使用。因此，x=x+m 首先将变量 x 从活跃变量列表中移除，然后，对于 x 的赋值使用到了 x 和 m 两个变量，因此这两个变量将在上文被定义，所以将变量 x 与 m 加入到活跃变量列表中。于是，我们认为，语句 x=x+m，从活跃列表中移除了变量 x，添加了变量 x 与 m。

那么传递函数中 use_s，OUT[s]，def_s 分别为：

use_s：语句 x=x+m 中，使用了变量 x 与 m。

OUT[s]：执行语句后的程序状态。

def_s：语句 x=x+m 中，定义的变量 x。

则传递函数 $f_s=use_s \cup (OUT[s]-def_s)$的语义为，对于待分析的赋值语句 s，执行语句之后存在的赋

值情况记为 OUT[s]，若其对某变量 x 进行赋值，则先从赋值语句中去除该变量 x（OUT[s]-def_s），然后将当前语句使用的变量 use_s 加入到程序状态中。

对于控制流约束函数而言，活跃变量模式描述的是在某一程序点上，其后继的基本块中是否可能存在对某变量的使用。因其为 May 分析，我们通常将控制流约束函数构造为如下的形式：

OUT[B]=$\cup_{S\text{是}B\text{的一个后继}}$IN[S]。

我们将该控制流约束函数运用到图 6.40 的例子中，描述程序状态变化。

B1 的后继基本块有两个：基本块 B2 和 B3，要分析在基本块 B1 出口处活跃的变量，需要合并后继基本块在入口处的程序状态。基本块 B2 和 B3 分别提供了活跃变量 x、y 和 a、b，因变量在后继基本块 B2 与 B3 中被使用，所以变量均可能在 B1 中存在定义语句。因此，活跃变量模式的控制流约束函数，使用并运算进行程序状态的合并。

构造了传递函数与控制流约束函数，接下来我们应当基于函数构造算法，计算我们想要的分析结果。我们用一个简单的程序举例，对其进行活跃变量分析。

因活跃变量分析使用后序分析的方式，因此构造的迭代算法与之前的有所不同：

```
IN[EXIT]=Ø;
for(除 EXIT 之外的每个基本块 B) IN[B]=Ø;
while(某个 IN 值发生了改变)
    for(除 EXIT 之外的每个基本块 B){
        OUT[B]=∪S是B的一个后继IN[S];
        IN[B]=useB∪(OUT[B]-defB);
    }
```

因使用 May 分析方式，我们在进行分析之前，将程序内的变量活跃情况初始化为 0（如图 6.41 所示）。 那么，第一次遍历后记录的程序状态如图 6.42 所示。

在第二次遍历后，每个基本块的 IN 值均没有改变，因此迭代算法就此终止。

在中间代码优化模块的实现中，我们通常需要使用上述的三个数据流分析模式，实现不同的优化策略；一般情况下，我们可以选择实现其中的一个或多个，完成优化管道的构建。

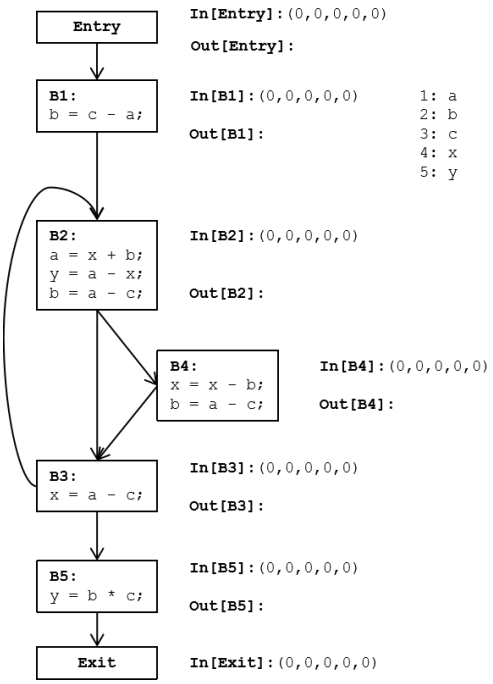


图 6.41 活跃变量分析模式初始化

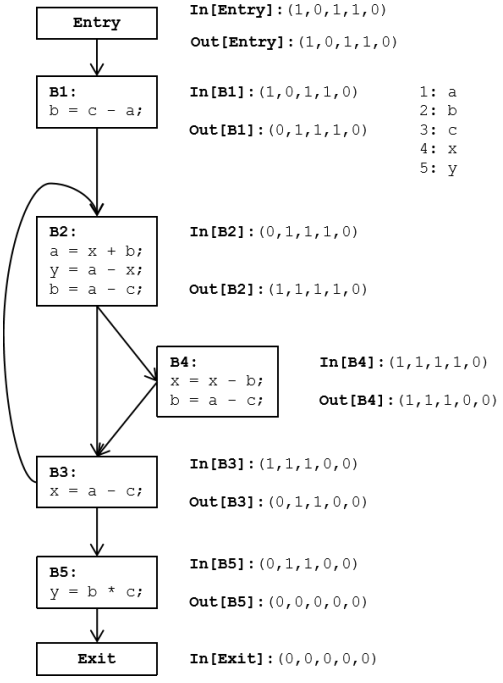


图 6.42 第一次迭代结果

6.2 中间代码优化的实践技术

6.2.1 局部优化

局部优化是对基本块内部进行优化。大量局部优化技术需要将基本块内部的指令转化为一个**有向无环图（Directed Acyclic Graph，DAG）**。通过生成的有向无环图，我们能够对中间代码进行语义不变的转换，从而提升目标代码的质量。

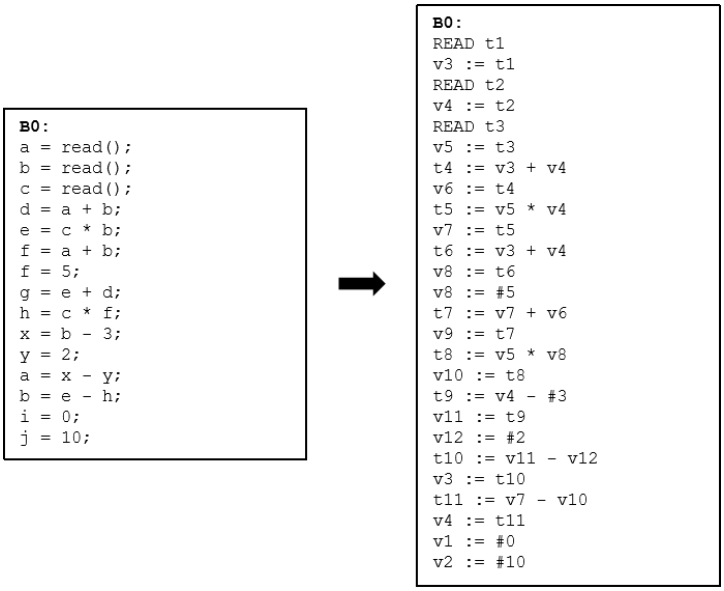


图 6.43 从源代码翻译为中间代码示例

我们以生成的程序控制流图中的基本块 B0 为例，首先将源代码翻译为图 6.43 中的中间代码，然后生成如图 6.44 所示的有向无环图。

有向无环图中的节点分为三类：顶部节点、底部节点和其他节点。底部节点表示常量或是定值，顶部节点表示当前基本块内部赋值但未使用的变量，其他节点标号中的运算符及子节点构成了赋值。

于是，对于图 6.43 中的每一条中间代码，我们以如下的方式完成有向无环图的构造：

1. 基本块中的每一个变量均有一个 DAG 节点对应其值。

2. 基本块中每一个语句 s 均有一个 DAG 节点 N，N 的子节点是 s 中用于定值的运算分量。

3. 节点 N 的标号是 s 中的运算符，若不存在则为变量本身（如 v5=t3）。

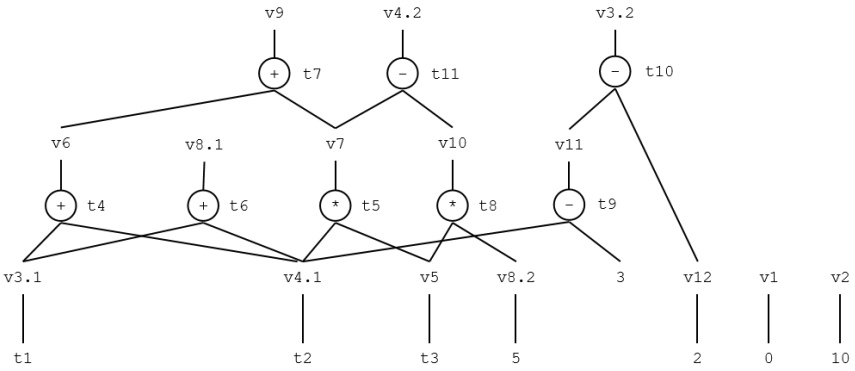


图 6.44 从图 6.43 中间代码生成的有向无环图

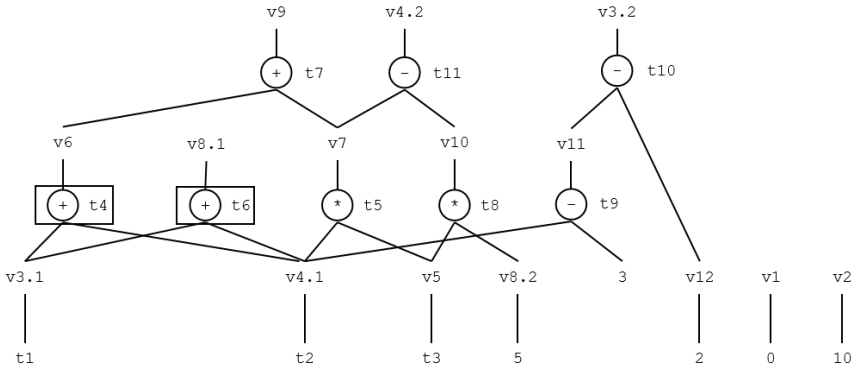


图 6.45 公共子表达式示例

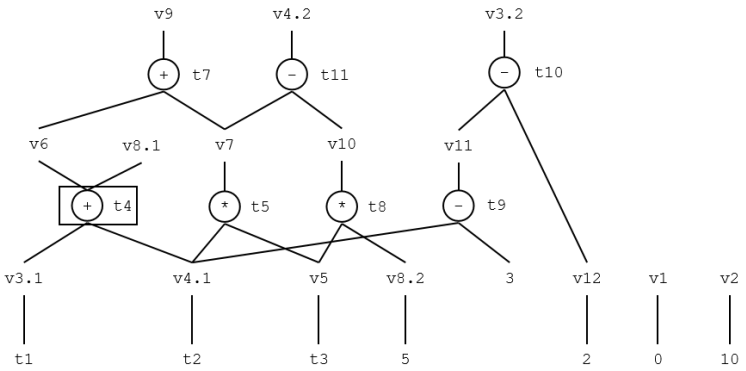


图 6.46 公共子表达式消除

基于构造的有向无环图，我们可以进行如下的优化：

（1）**公共子表达式消除（Common Subexperssion Elimination）：**如果表达式 E 在某次出现之前已经被计算过，并且表达式中的操作数在计算后一直没被修改过，那么在操作数被下次修改之前，E 被称为是公共子表达式。通过观察图 6.45 我们可以发现，有向无环图中标记为 t4 与 t6 的两个节点，运算符与所有子节点均相同，因此，对于 t4 的求值结果可以直接用于对 t6 的求值中。因此可以使用 t4 代替 t6，完成对子表达式的消除。在进行子表达式消除后，有向无环图如图 6.46 所示。

（2）**无用代码消除（Dead Code Elimination）：**我们可以看到，在基本块 B0 内部，存在对变量进行两次定义，并且之间没有使用该变量的情况。对于永远不会被使用的定义，为之开辟内存、进行计算是没有必要的。

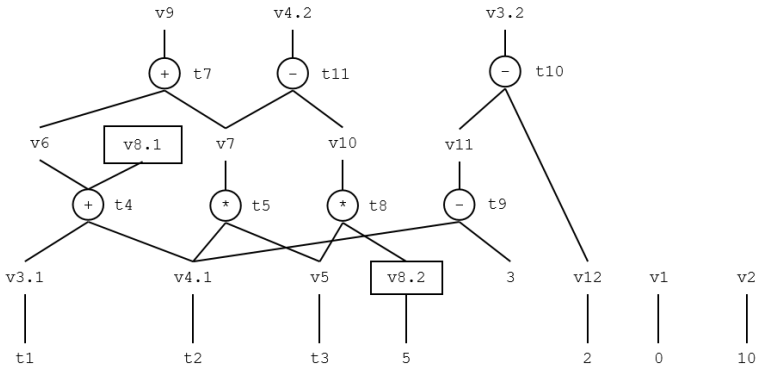


图 6.47 无用代码示例

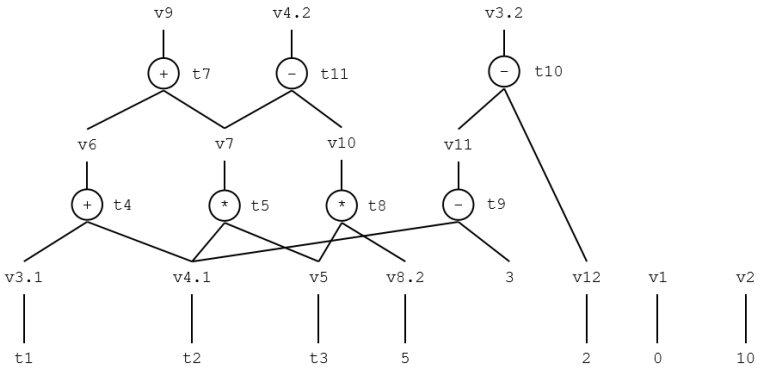


图 6.48 无用代码消除

如图 6.47 所示，有向无环图中的顶部节点表示在定义后未被使用，或两次定义之间未被使用的变量，图中共有 v1、v2、v3.2、v4.2、v8.1、v9 等，局部无用代码消除仅考虑消除基本块内部的冗余定义，因此可消除 v8.1 及其对应的赋值语句，转化后的有向无环图如图 6.48 所示。

（3）**常量折叠（Constant Folding）：**我们可以发现，对于程序内部如 f=5，y=2，i=0，j=10 等将变量定义为一个常量的赋值语句，在该变量被再次定义之前，可将变量替换为一个常量，从而简化计算的过程。

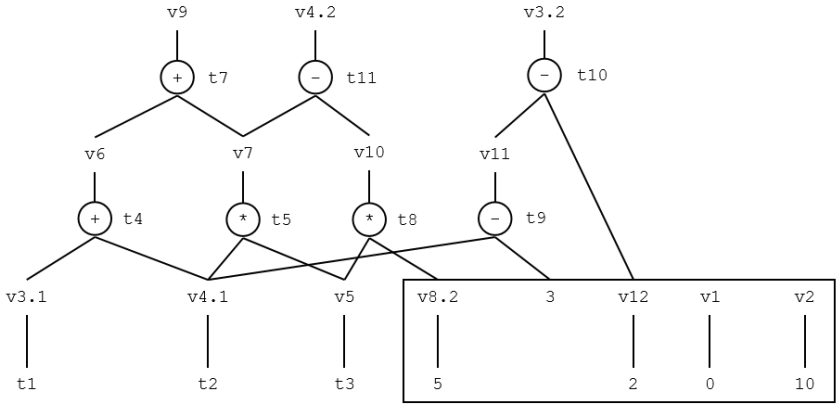


图 6.49 被赋值为常量的变量示例

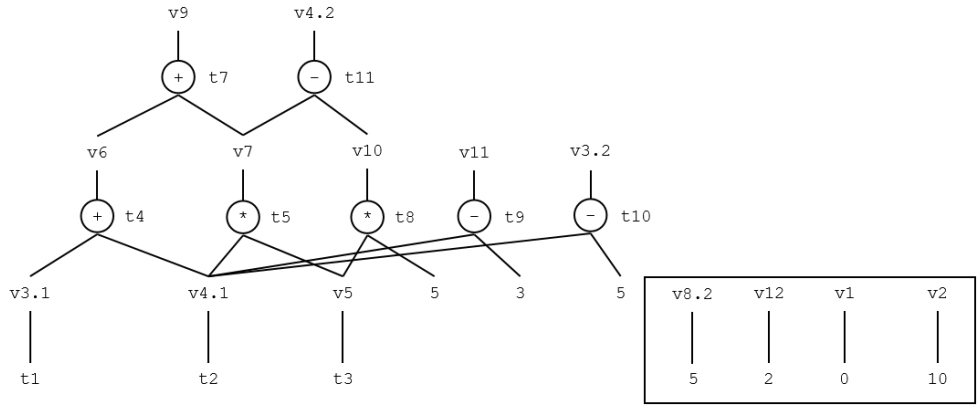


图 6.50 常量折叠

从图 6.49 中的叶子节点，我们可看出，被定义为常量的变量有 v1、v2、v8.2、v12。在使用的过程中我们能够使用常量值代替其参与运算，在进行常量折叠后，结果如图 6.50 所示。并且，我们可以基于代数恒等式计算，尽可能地简化与合并常量值。

有向无环图进行常量折叠之后，有向无环图的结构发生改变，可能导致无用代码的暴露。同时，对于生成的有向无环图进行代数恒等式替换，能够发掘代码内部的公共子表达式，如图 6.51 所示。在图 6.51 中，发现代数恒等式 t11=t2*t3-t3*5=t3*(t2-5)，转化后能够发掘出 v4.1-5 这一公共子表达式，并简化生成的中间代码。

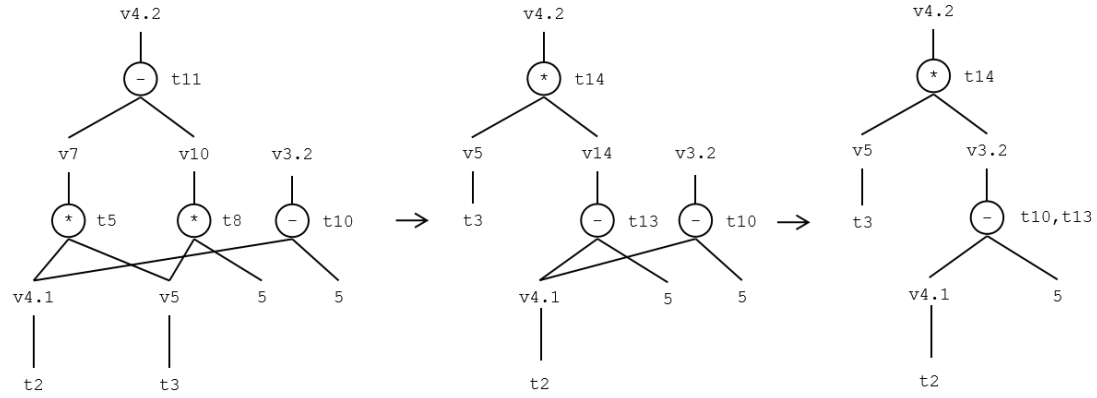


图 6.51 代数恒等式替换

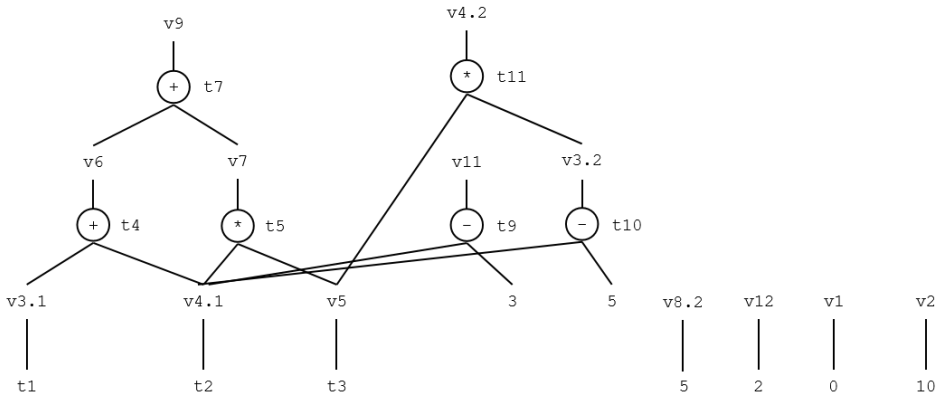


图 6.52 优化后的有向无环图

经过以上的优化后，我们需要将有向无环图还原为中间代码形式。同时，以上的方法还可用于机器相关的优化，通过调整代码顺序，进行寄存器优化等，提升目标代码的质量。最终生成的有向无环图与中间代码如图 6.52 所示，优化后的中间代码如图 6.53 所示。

完成了单个基本块的优化工作，我们将视野放大到整个函数，进行多个基本块的优化。

```
B0:
READ t1
v3 := t1
READ t2
v4 := t2
READ t3
v5 := t3
t4 := v3 + v4
v6 := t4
t5 := v5 * v4
v7 := t5
t6 := v3 + v4
v8 := t6
v8 := #5
t7 := v7 + v6
v9 := t7
t8 := v5 * v8
v10 := t8
t9 := v4 - #3
v11 := t9
v12 := #2
t10 := v11 - v12
v3 := t10
t11 := v7 - v10
v4 := t11
v1 := #0
v2 := #10
```

➡

```
B0:
READ t1
v3 := t1
READ t2
v4 := t2
READ t3
v5 := t3
t4 := v3 + v4
v6 := t4
t5 := v5 * v4
v7 := t5
v8 := #5
t6 := v7 + v6
v9 := t6
t7 := v5 * #5
v10 := t7
t8 := v4 - #3
v11 := t8
v12 := #2
t9 := v4 - #5
v3 := t9
t10 := v5 * v3
v4 := t10
v1 := #0
v2 := #10
```

图 6.53 优化后的中间代码

6.2.2 全局优化

前文提到，良好的优化顺序能够降低编译过程中的时空开销，针对于生成的中间代码，进行了局部优化之后，我们构造全局优化管道：常量传播-公共子表达式消除-复制传播-常量折叠-控制

流优化-无用代码消除-循环不变代码外提-归纳变量强度削减-控制流优化，我们使用前文所示的程序与生成的控制流图（图 6.54）进行全局优化。

全局优化 1 常量传播

我们之前在进行局部优化时，将恒为常量的变量转化为常量。基本块内部语句都按照从入口到出口的顺序依次执行，因此进行常量折叠并不困难。那么，对于在多个基本块之间乃至复杂的控制流图中进行常量传播的计算应当采取什么样的方法呢？

常量传播框架与到达定值模式较为接近，所不同的是，到达定值中对于变量的赋值情况只存在两种状态：生效与失效，而对于常量传播框架而言，常量值的集合是无限的。

常量传播框架中的变量的状态分为三种：

1.所有符合该变量类型的常量值；

2.NAC（Not-A-Constant）表示当前变量不是一个常量值。这代表该变量在到达程序点 p 的不同的路径上的值不同，或是被赋予了一个输入变量的值；

3.UNDEF，表示未定义的值。在到达程序点 p 的不同路径上存在至少一条路径未对变量的值进行定义。

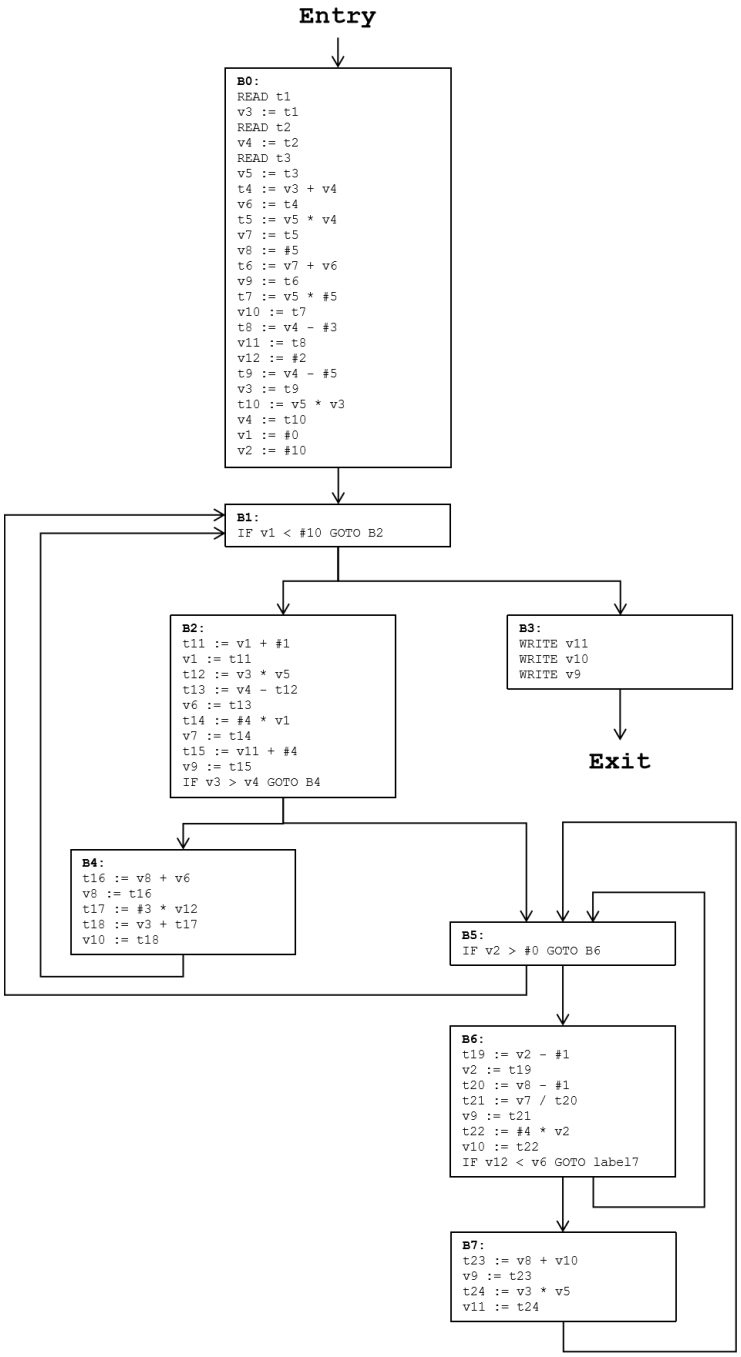


图 6.54 控制流图

常量传播函数：

三种状态分别对应三种情况：假设程序点上有变量 v，第一种状态表示该变量 v 唯一地对应

一个定值 a，即在程序点 p 上 v 的可能取值集合为{a}；第二种状态 NAC 表示，在经过程序点 p 的多条路径上分别对变量 v 进行赋值，且不同路径到达 p 时 v 的可能取值至少有 2 个且数量有限，即 v 的可能取值集合为{a,b,c,...,n}；第三种状态 UNDEF 表示在程序点 p 上存在至少一条从入口到程序点 p 的路径，在这条路径上，没有对变量 v 的赋值或已被赋空。

常量的传播主要通过变量的赋值语句进行，如 z＝x＋y 等表达式。设 S_z，S_x，S_y 分别为执行该语句后变量 z，x，y 的状态，则 S_z 的求值存在 9 种情况，如表 6.1 所示：

表 6.1 常量传播状态

	UNDEF	c1	NAC
UNDEF			
c2			
NAC			

因其存在对称性，故我们只要讨论其中 4 种情况即可。

（1）x 与 y 中有至少一个变量状态为 UNDEF，这表示 x 与 y 中至少有一个值，存在一条从入口到程序点 p 的路径，使得该变量未被定义，故 S_z 为 UNDEF，如表 6.2 所示。

表 6.2 x 与 y 至少有一个 UNDEF

	UNDEF	c1	NAC
UNDEF	UNDEF	UNDEF	UNDEF
c2	UNDEF		
NAC	UNDEF		

（2）x 为一定值 c1，y 为一定值 c2，那么 z＝x＋y 可转化为 z＝c1＋c2，S_z 为 c3=c1+c2，如表 6.3 所示。

表 6.3 x 与 y 均为定值

	UNDEF	c1	NAC

UNDEF	UNDEF	UNDEF	UNDEF
c2	UNDEF	c3	
NAC	UNDEF		

（3）x 为一定值 c1，y 为 NAC，即 y 的取值集合中存在多个值，y1 y2...，则 x+y 可转化为 c1+y1, c1+y2...，z 的取值集合也有多个值，因此 S_z 的状态为 NAC，如表 6.4 所示。

表 6.4 x 与 y 中有一个是 NAC

	UNDEF	c1	NAC
UNDEF	UNDEF	UNDEF	UNDEF
c2	UNDEF	c3	NAC
NAC	UNDEF	NAC	

（4）x，y 的状态均为 NAC，则由（3）可知其状态为 NAC，如表 6.5 所示。

表 6.5 x 与 y 均为 NAC

	UNDEF	c1	NAC
UNDEF	UNDEF	UNDEF	UNDEF
c2	UNDEF	c3	NAC
NAC	UNDEF	NAC	NAC

根据上述的四种情况可以看出，三种状态的转变具有单调性，状态转变的顺序只可能由 UNDEF 转化为常量再转化为 NAC，或直接由 UNDEF 转化为 NAC，而不可能反向转化。通过常量传播框架，可构造相应的传递与控制流约束函数，进行程序状态的分析。

我们关注四种常量传播算法：

1) **简单常量传播（Simple Constant Propagation）**

简单常量传播框架由 Kildall 设计，基于数据流和传递函数进行常量传播优化。算法的设计与前文所介绍的数据流分析模式有些类似：对程序状态进行初始化，构造传递函数与控制流约束函数，使用迭代算法或工作列表算法进行数据流值的传递与计算，输出结果。

传递函数：常量传播框架的传递函数较为复杂，因其存在三种状态与九种情况。根据上文的分析，我们现在能够将九种情况归纳如下：

对于一条针对变量 v 的赋值语句 s：

1. s 的等号右侧为一常量 c，那么 x 的取值集合为{c}，状态为常量 c；

2. s 的等号右侧为二元运算表达式（例如 x+y 等），则根据上一节所述的四种情况进行计算；

3. s 的等号右侧包含函数调用或其他语言特性，那么 x 的状态为 NAC。

控制流约束函数：

控制流约束函数与传递函数略有不同：

UNDEF∪c=c　　NAC∪c=NAC

c∪c=c　　c1∪c2=NAC

我们假设输入的中间代码不包含对未定义的变量的使用。程序内部的所有变量状态初始化为 UNDEF，因此控制流约束函数与传递函数的一个不同即在于，UNDEF∪c=c，其原因如下图：

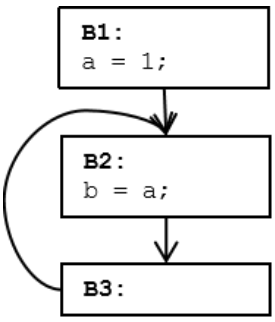


图 6.55 初始化示例

我们将程序内部的状态初始化为 UNDEF，那么，若 UNDEF∪c＝UNDEF，B3 中无对变量 a 的赋值语句，则在 B2 的入口，a 的状态恒为 UNDEF。

简单常量传播基于到达定值模式，我们可以基于前面章节介绍的到达定值模式和上文介绍的常量传播框架，基于数据流、传递函数、控制流约束函数与迭代算法，进行常量传播的计算。图 6.54 控制流图的简单常量传播计算结果如图 6.56 所示。

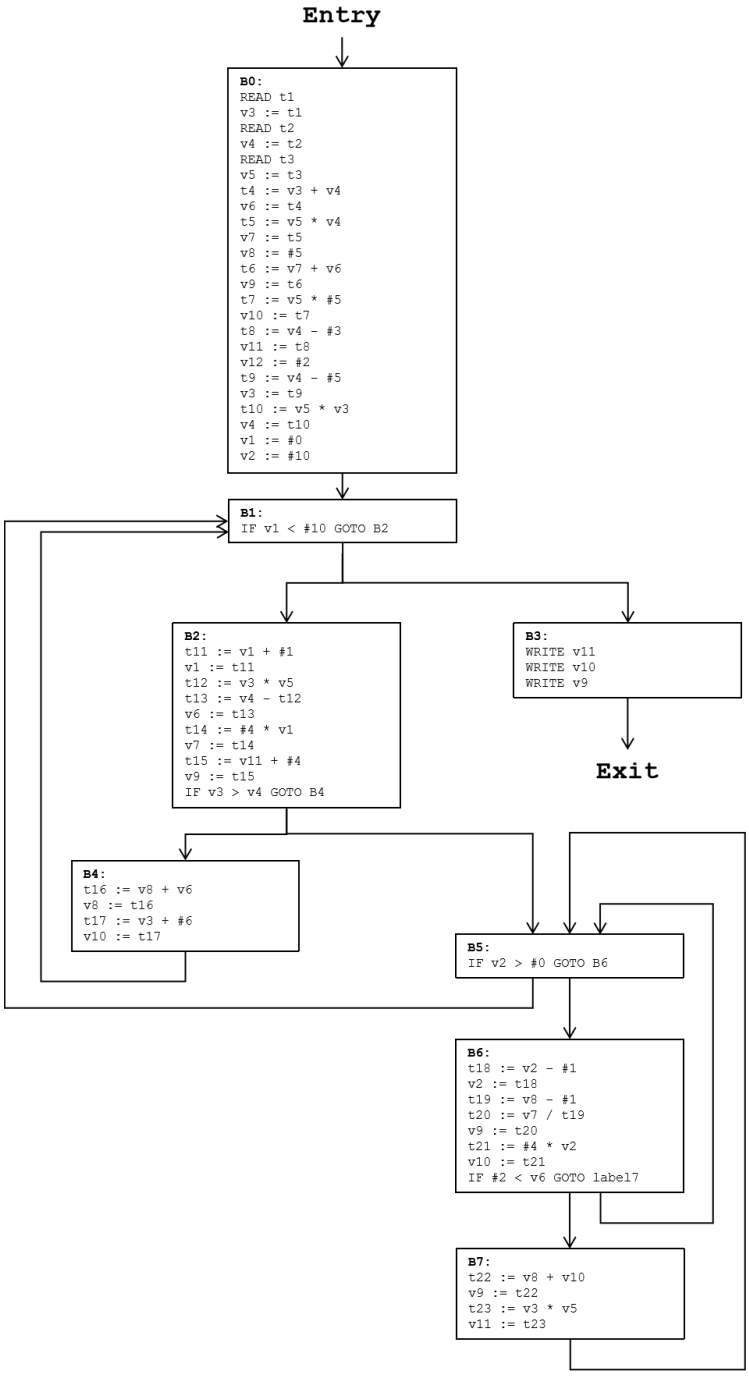


图 6.56 简单常量传播后的控制流图

使用以上的常量传播框架进行计算之后，我们的算法应当能够发现，变量 v12 在循环内部的值为一常量 6，因此可以使用常量去替换。如果框架足够优秀，或许能够发现，基本块 B2 中，定

值 t12 的值为零，且执行语句 v6 := t12 后变量 v6 的值也恒为零，可进行大量常量替换，简化控制流图等，这需要我们进行一系列的优化，在后文进行叙述。

本书只要求掌握基于简单常量传播，使用数据流分析，设计传递函数与控制流约束函数构造常量传播优化模块，完成常量传播优化。

在简单常量传播的基础上，我们介绍另外三种常量传播框架。

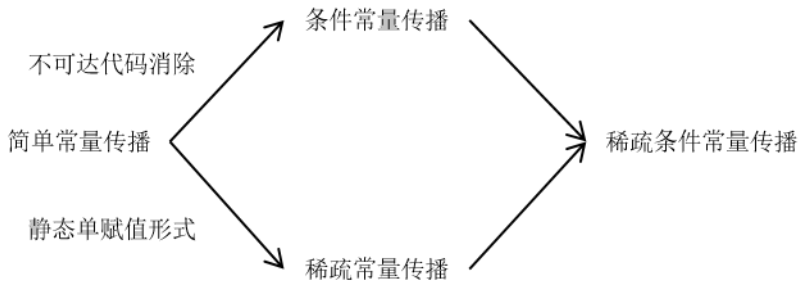


图 6.57 四种常量传播框架

2) 条件常量传播（Conditional Constant Propagation）

在某些情况下，数据流图上的部分基本块实际上是不可达的。这些不可达的基本块中，有一部分可以在编译阶段进行代码削减，它们属于无用代码中的一部分，这种优化被称为**不可达代码消除（Unreachable Code Elimination）**。不可达基本块中对变量的赋值关系，不应当加入到常量传播的计算之中。

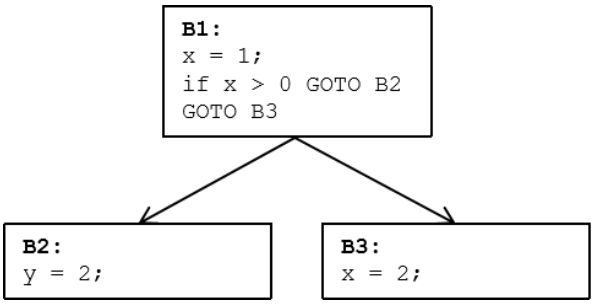


图 6.58 不可达代码示例

例如，对于图 6.58 中的控制流图，对它使用简单常量传播会分析 B1、B2、B3 三个基本块。

然而，我们使用工作表算法进行计算时，默认除了 Entry 之外的基本块都处于“未被执行”状态，对于每次基本块跳转，先进行常量传播分析，将可跳转的基本块加入到工作表中，消除未被执行的基本块。在图 6.58 中，基本 B1 的跳转语句中的条件判断 x > 0，通过常量传播分析可以知道 x 为常量 1，条件判断表达式恒为真，因此在分析时只会将基本块 B2 加入到工作表中，消除未被执行的基本块 B3。

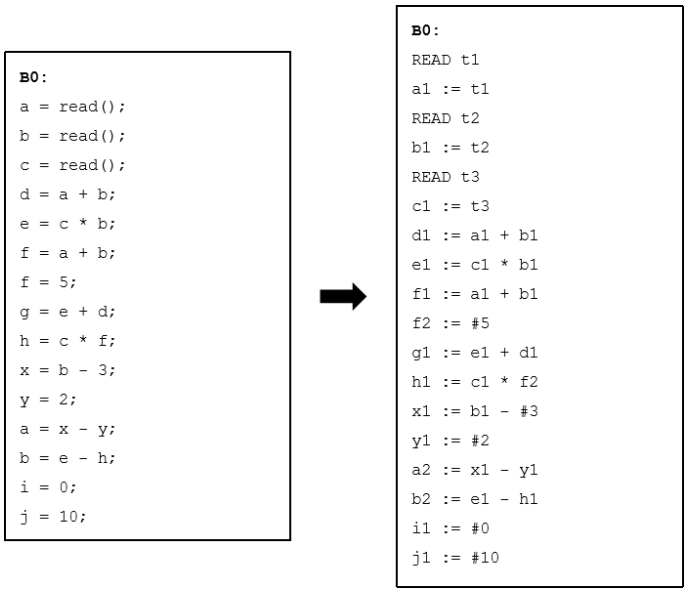


图 6.59 SSA form 中间代码

3) 稀疏常量传播（Sparse Constant Propagation）

要介绍稀疏常量传播，我们先需要介绍一系列概念。

稀疏常量传播中的**“稀疏”（Sparse）**二字，指的是以**静态单赋值形式（Static Single-Assignment Form, SSA form）**的中间表示形式为基础，分析变量的**使用-定义**关系。静态单赋值形式的中间代码，其每个变量仅被赋值一次，且每次使用前必定已被赋值，**变量的使用（Use）**是作为表达式的一个运算分量或赋值语句等号右侧的值，**变量的定义（Define）**指该变量被赋值。静态单一赋值的中间代码，因每个变量仅被赋值一次，对于每次使用均可轻易找到其对应的赋值关系，被称为使用-定义关系；通常情况下，为了便于叙述方便，我们将使用-定义关系简称为 **use-def 关系**。

通过SSA形式的中间代码，我们可以轻易地通过单次遍历完成基本块内部的局部优化：公共子表达式消除、常量折叠、无用代码消除的实现。

以基本块B0为例，例如，对于变量a，其在基本块内部被定义了两次：a1与a2。

我们将a1与a2看作不同的变量，对于每一个变量，均只被赋值了一次，那么，对于变量的赋值与使用关系，我们可以构造use-def表如下：

```
1  a1 := t1                a1 def: 1  use: 4 6
2  b1 := t2                b1 def: 2  use: 4 5 6
3  c1 := t3                c1 def: 3  use: 5 9
4  d1 := a1 + b1           d1 def: 4  use: 8
5  e1 := c1 * b1           e1 def: 5  use: 8 13
6  f1 := a1 + b1           f1 def: 6  use:
7  f2 := #5                f2 def: 7  use: 9
8  g1 := e1 + d1           g1 def: 8  use:
9  h1 := c1 * f2           h1 def: 9  use: 13
10 x1 := b1 - #3           x1 def: 10 use: 12
11 y1 := #2                y1 def: 11 use: 12
12 a2 := x1 - y1           a2 def: 12
13 b2 := e1 - h1           b2 def: 13
14 i1 := #0                i1 def: 14
15 j1 := #10               j1 def: 15
```

图 6.60 def-use 关系

常量折叠：因每一变量仅被定义一次，因此若对变量的赋值为一常量，可直接使用常量替换该定值，如变量f2等。

公共子表达式消除：对于公共子表达式a1+b1，变量a1与b1在第4行与第6行同时被use，因此可以进行公共子表达式消除。同时，记录变量的值，然后使用局部变量优化中所提到的代数恒等式变换，能够暴露b1-5这一公共子表达式，并予以消除。

无用代码消除：在基本块出口不活跃的变量，若在基本块的内部仅被def而未有use，可认为其是无用代码并予以消除，如f1 := a1 + b1等。

SSA形式的中间代码，给优化带来了极大的便利，但是，构造SSA形式的中间代码的算法较为复杂，我们介绍一种基于**支配树（Dominator Tree）**的SSA形式的中间代码生成算法。

支配节点：对于某节点n，如果从入口节点到n的每一条路径都必须经过节点m，则节点m支配节点n，记为m dom n，每一个节点都支配自己。

支配关系满足自反性、反对称性和传递性：

自反性：a dom a；

反对称性： a dom b, b dom a => a=b;

传递性：a dom b, b dom c => a dom c。

例如，对于图 6.61 中的控制流图，包含这些支配关系：B1 支配 B1、B2、B3、B4、B5，B2 支配 B2、B3、B4、B5，B3 支配 B3 和 B5 等。构造的支配树如图 6.62 所示。

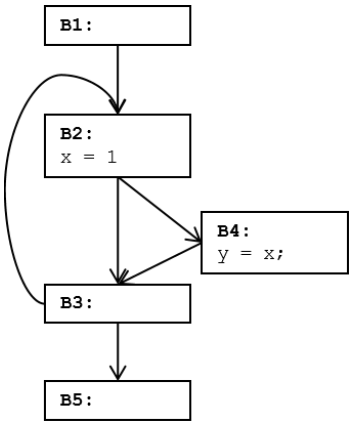


图 6.61 控制流图示例

利用支配树与支配关系，能够进行循环相关的优化，也可以帮助生成 SSA 形式的中间代码。为了生成 SSA 形式的中间代码，我们需基于支配树计算基本块的**支配边界（Dominance Frontier）**，即 A 支配 B 的一个前驱节点但不严格支配 B。在此我们引入一个虚拟函数 **phi 函数**，进行程序状态的合并。phi 函数是在支配边界上用于合并不同路径上程序状态的函数，例如，图 6.63 中使用 phi 函数合并基本块 B3 的两个前驱节点 B1、B2 中对 x 的赋值关系，并且使用 x3=phi(x1,x2)替代 x1 和 x2 在接下来的程序片段中被使用。

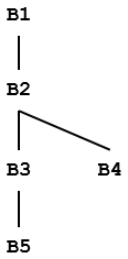


图 6.62 支配树

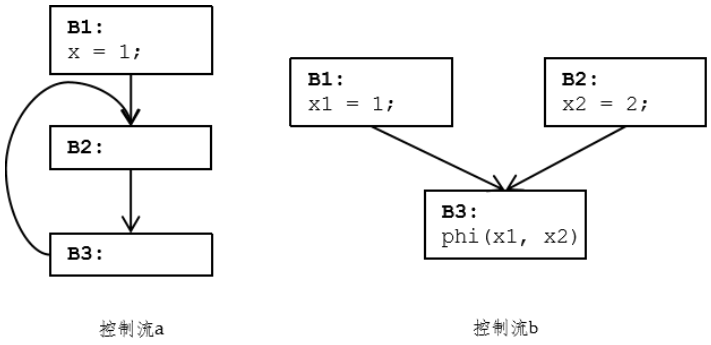


图 6.63 放置 phi 函数的示例

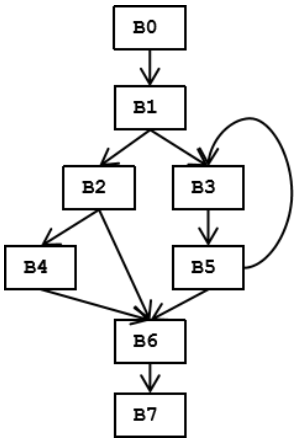


图 6.64 用于计算支配关系的控制流图

对于图 6.63 中的控制流 a，对变量 x 的赋值只存在于 B1 中，B1 支配 B2 与 B3，因此在 B2 中不需要添加 phi 函数。在控制流 b 中，对变量 x 的赋值存在于 B1 与 B2 中，B1 与 B2 均不支配 B3，因此需添加 phi 函数。

计算控制流图支配边界的算法较为复杂，我们将其分为以下的步骤：

1）计算支配关系。

基于迭代算法，我们可以计算基本块之间的支配关系。在上文中，我们能够得知，一个基本块 m 支配另一个基本块 n，这表示对于所有由入口到 n 的路径均经过 m，因此，对于基本块 n，若 m 支配其的所有前驱基本块，则 m 支配 n。因此，不同路径上的程序状态合并，与数据流分析模式中采用的 Must 分析类似。

block	D(n)
B0	{B0}
B1	{B0,B1,B2,B3,B4,B5,B6,B7}
B2	{B0,B1,B2,B3,B4,B5,B6,B7}
B3	{B0,B1,B2,B3,B4,B5,B6,B7}
B4	{B0,B1,B2,B3,B4,B5,B6,B7}
B5	{B0,B1,B2,B3,B4,B5,B6,B7}
B6	{B0,B1,B2,B3,B4,B5,B6,B7}
B7	{B0,B1,B2,B3,B4,B5,B6,B7}

block	D(n)
B0	{B0}
B1	{B0,B1}
B2	{B0,B1,B2}
B3	{B0,B1,B3}
B4	{B0,B1,B2,B4}
B5	{B0,B1,B3,B5}
B6	{B0,B1,B6}
B7	{B0,B1,B6,B7}

图 6.65 支配关系表

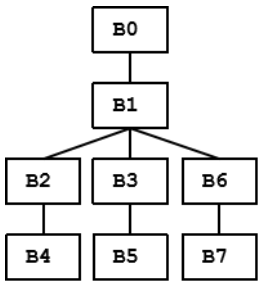


图 6.66 支配树

于是，我们构造支配情况更新函数如下：

D(n)=$\cap_{\text{p 是 n 的一个前驱}}$D(p)$\cup$ {n}，D(n)为支配 n 的基本块

根据支配的定义，支配节点 n 的基本块满足两个条件之一：

1.支配其所有前驱基本块 $\cap_{\text{p 是 n 的一个前驱}}$D(p)；

2.是 n 本身 {n}。

我们将基本块的支配情况初始化为包含所有基本块的集合，然后使用数据流分析模式中的 Must 分析类似方法进行支配情况的迭代更新，如图 6.65 所示。

2）基于支配关系，我们构造支配树如图 6.66。

3）构造严格支配表。

我们引入**严格支配（strictly dominated）**的概念，记作 m sdom n，其中 n≠m，基于支配树，我们可获得严格支配关系如下，其中 sdom(n)代表基本块 n 严格支配的基本块。

block	dom(n)
B0	{B0,B1,B2,B3,B4,B5,B6,B7}
B1	{B1,B2,B3,B4,B5,B6,B7}
B2	{B2,B4}
B3	{B3,B5}
B4	{B4}
B5	{B5}
B6	{B6,B7}
B7	{B7}

block	sdom(n)
B0	{B1,B2,B3,B4,B5,B6,B7}
B1	{B2,B3,B4,B5,B6,B7}
B2	{B4}
B3	{B5}
B4	{}
B5	{}
B6	{B7}
B7	{}

图 6.67 严格支配关系

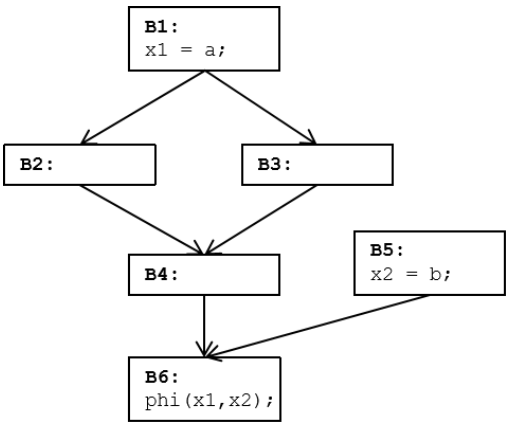


图 6.68 支配边界示例

4）寻找支配边界。

基本块 n 的支配边界可以用图 6.68 的例子来解释。

程序内部对变量 x 的赋值共有两处：基本块 B1 与 B5。基本块 B1 支配基本块 B2、B3 和 B4，而到达 B6 不必经过基本块 B1，因此 B6 为 B1 的支配边界，同理 B6 也是 B5 的支配边界，因此在基本块 B6 内添加 phi 函数。

要寻找支配边界，即寻找基本块 n 的后继中，第一个不被基本块 n 支配的基本块，要寻找具备这种性质的基本块，可以从被支配的基本块的后继基本块中寻找。此处对于图 6.68 中的控制流图及其中的基本块，B1 支配 B1、B2、B3、B4，而 B1 的后继 B2、B3，B2 和 B3 的后继 B4，B4 的后继 B6 组成的后继集合{B2、B3、B4、B6}中，只有 B6 不被基本块 B1 支配，因此 B6 为 B1 的

支配边界。

block	sdom(n)
B0	{B1,B2,B3,B4,B5,B6,B7}
B1	{B2,B3,B4,B5,B6,B7}
B2	{B4}
B3	{B5}
B4	{}
B5	{}
B6	{B7}
B7	{}

block	succ(dom(n))
B0	{B1,B2,B3,B4,B5,B6,B7}
B1	{B2,B3,B4,B5,B6,B7}
B2	{B4,B6}
B3	{B3,B5,B6}
B4	{B6}
B5	{B3,B6}
B6	{B7}
B7	{}

图 6.69 succ(dom(n))

block	
B0	{}
B1	{}
B2	{B6}
B3	{B3,B6}
B4	{B6}
B5	{B3,B6}
B6	{}
B7	{}

图 6.70 支配边界

于是，我们用 succ(dom(n))表示基本块 n 支配的基本块的后继，如图 6.69 所示。对于这些后继集合，我们只要从中去除基本块 n 支配的基本块，即可计算出 n 的支配边界，如图 6.70 所示。

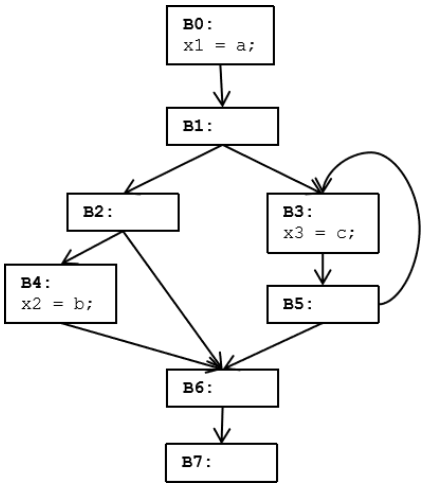


图 6.71 程序示例

计算出支配边界之后，我们就可基于支配边界进行 phi 函数的添加。

假设程序代码如图 6.71 所示。对变量 x 的赋值共有三处：B0、B3、B4，其中 B0 的支配边界集合为{}，B3 的支配边界集合为{B3,B6}，B4 的支配边界集合为{B6}。

首先，我们使用到达定值模式，计算在每个基本块入口处，x 的可能取值情况，如表 6.6 所示。

表 6. 6 x 的赋值情况

B0	B1	B2	B3	B4	B5	B6	B7
{x1}	{x1}	{x1}	{x1, x3}	{x1}	{x1, x3}	{x1, x2, x3}	{x1, x2, x3}

我们使用工作表算法，进行 phi 语句的植入。

工作表初始化为 w : {B0、B3、B4}

1. 从 w 中取得基本块 B0，B0 的支配边界集合为{}，不添加 phi 语句

w : {B3、B4}

2. 从 w 中取得基本块 B3，B3 的支配边界集合为{B3、B6}，在基本块 B3 和 B6 中添加 phi 语句，根据表 6.6 可知，B3 中添加的 phi 语句合并 x1 和 x2，B6 中添加的 phi 语句合并 x1、x2 和 x3，然后将 B3 和 B6 加入工作表中（因基本块 B3 已添加 phi 语句，因此只将 B6 加入到工作表 w 中）。

w : {B4，B6}

3. 从 w 中取得基本块 B4，B4 的支配边界集合为{B6}，因基本块 B6 已添加 phi 语句，因此不做操作。

w : {B6}

4. 从 w 中取得基本块 B6，B6 的支配边界集合为{}，因此不做操作。

因此，算法需要在基本块 B3 和 B6 中添加 phi 语句，并且将 x1、x2、x3 添加到 phi 语句中，获得的程序控制流图如图 6.72 所示。

添加了 phi 语句之后，便可构造 SSA 形式的中间代码。走了许久，不要忘了为什么出发，我们回看稀疏常量传播。稀疏常量传播中，变量有 use 和 def 的关系。以图 6.72 中的代码为例，变量 x1 的 def 在 B0 中，而存在 phi(x1, x3)、phi(x1, x2, x3)两处 use。相对于普通常量传播中在每个基本块的入口与出口进行变量状态的计算，在稀疏常量传播中仅需要在 def 和 use 处计算变量的状态，简化了大量的计算，这便是稀疏常量传播中“稀疏”二字的含义。

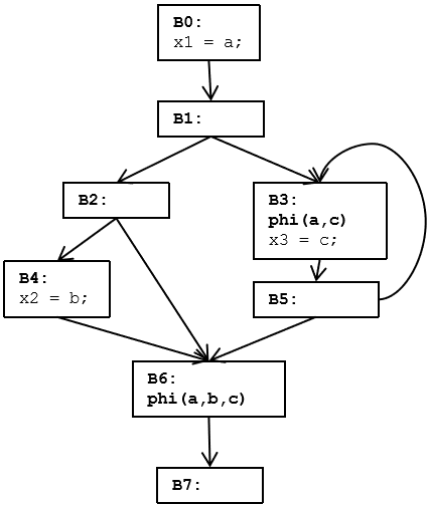


图 6.72 添加 phi 函数后的控制流图

例如，假设 x4＝phi(x1, x2,x3)，则 x4 的取值集合为{a, b, c}，状态为常量 a ∪ 常量 b ∪ 常量 c＝NAC，而不需要在基本块入口和出口反复进行计算。

4）稀疏条件常量传播（Sparse Conditional Constant Propagation）

将稀疏常量传播与条件常量传播相结合，就是当前主流的常量传播算法，稀疏条件常量传播

算法，通常简写为 SCCP，而过程间稀疏条件常量传播则简写为 IPSCCP。

在优化模块的实现中，本书重点关注基于简单常量传播算法进行常量传播，当然我们也可以构造其他的常量传播算法，或将中间代码转化为 SSA 形式，并进行下一步的优化工作。

全局优化 2 公共子表达式消除

在实践中，只要求你能够使用前文所述的可用表达式模式进行公共子表达式消除，因此，我们基于简单常量传播后的优化结果，运用可用表达式模式进行公共子表达式的消除。

可用表达式模式已在前文叙述，优化后的程序应如图 6.73 所示。

在经过子表达式消除之后，我们可以发现，对于基本块 B2 中的语句 t12 := v4 - t9 和 v6 := t12，经过对后续语句的扫描（或使用复制传播），变量 v4 和 v6 的值未被改变，而 v4 - t9 的值为一定值 5，此处可以使用常量 5 替换变量的使用。

我们可以注意到，在进行子表达式消除的时候，使用的子表达式消除方法可能较为呆板，且只有所有的前驱基本块中均存在该子表达式，子表达式才可被消除。事实上，编译器在子表达式消除优化上通常采取另一种策略：**部分冗余消除（Partial Redundancy Elimination, PRE）**，用于公共子表达式消除及循环不变代码外提。部分冗余消除使得表达式的求值满足以下性质：

1. 不改变程序语义；

2. 尽可能减少子表达式的求值；

3. 尽可能延后子表达式求值。（便于目标代码生成时，寄存器的相关优化）

部分冗余消除及相关的懒惰代码移动，此处囿于篇幅及重点不予展开，请学有余力的读者自行查阅相关资料并进行算法的实现。

进行常量传播和子表达式消除之后，部分变量被常量替代，冗余表达式被消除或移动位置，因此需要重新进行常量折叠与控制流优化，去除不可达的基本块及代码（分支条件不可满足），优化后结果如图 6.74。

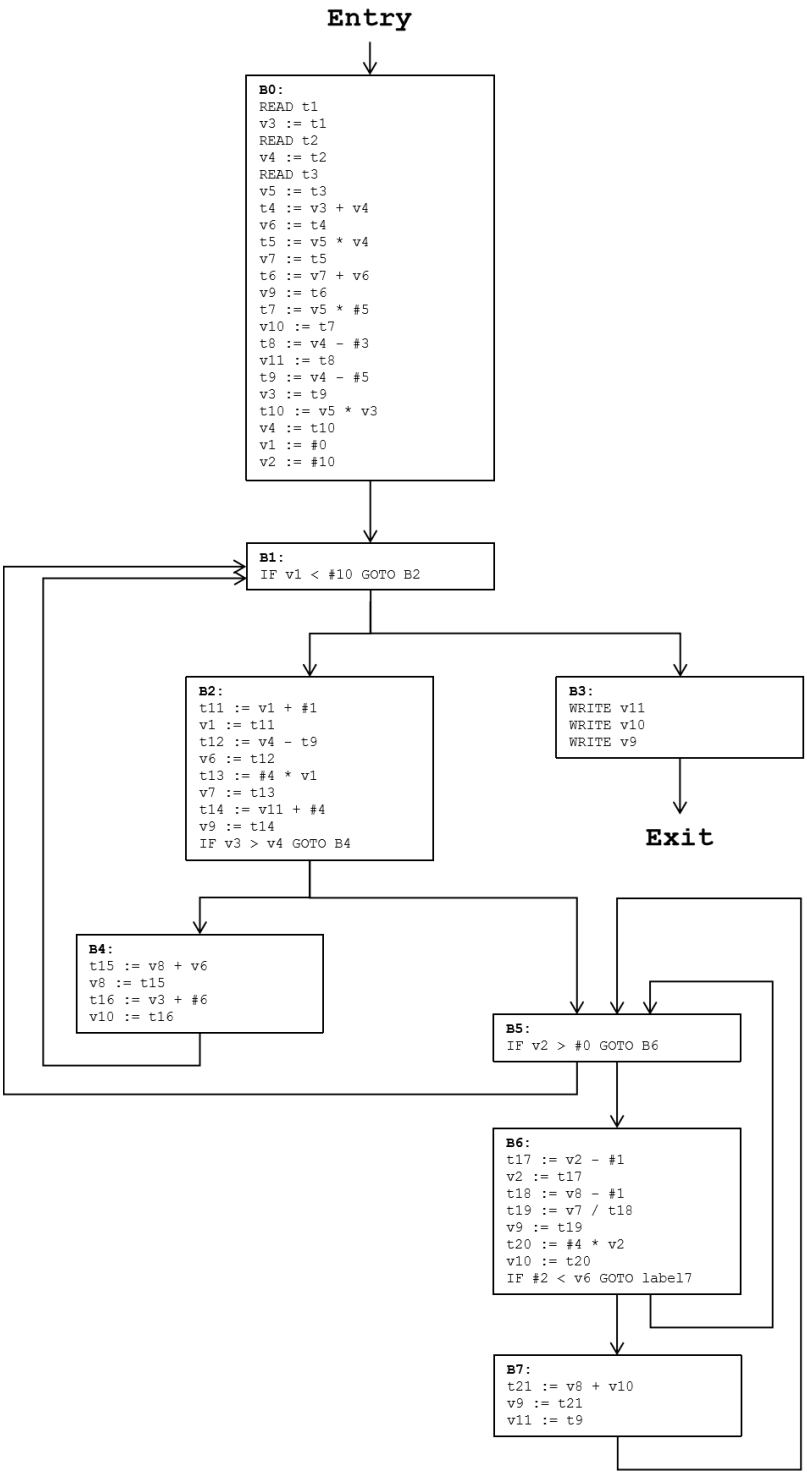


图 6.73 公共子表达式消除后的控制流图

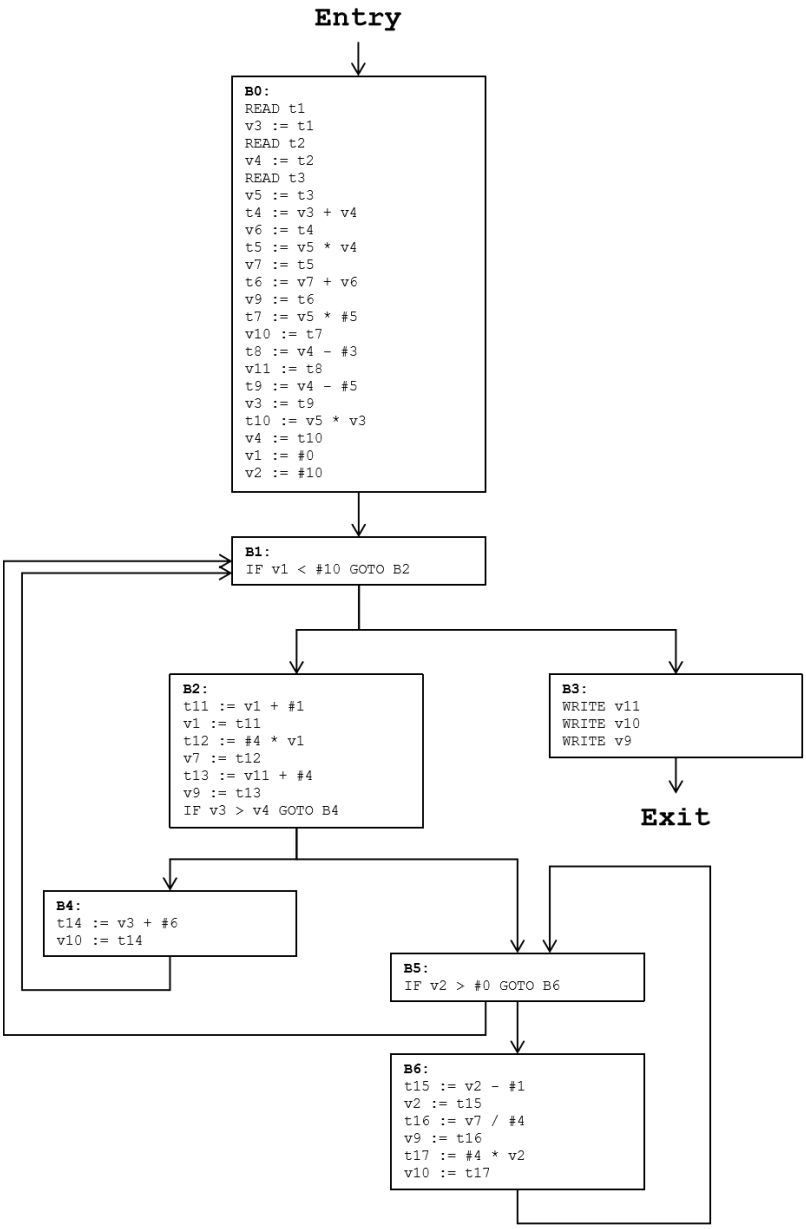


图 6.74 控制流优化后的控制流图

全局优化 3 无用代码消除

全局无用代码消除关注被赋值但未被使用的变量，若对于变量的一次定义，在后续的程序中从未被使用，该定义可以被认为是无用的而被消除。通常，我们需要迭代式地删除程序内部的无用代码。在本书的实践中，要求能够使用前文所述的活跃变量模式进行无用代码消除，当然，我

们也可基于生成的 SSA 形式的中间代码与 use-def 链，进行无用代码的消除。

活跃变量分析模式已在前文叙述，此处不再赘述，优化后的程序应如下所示：

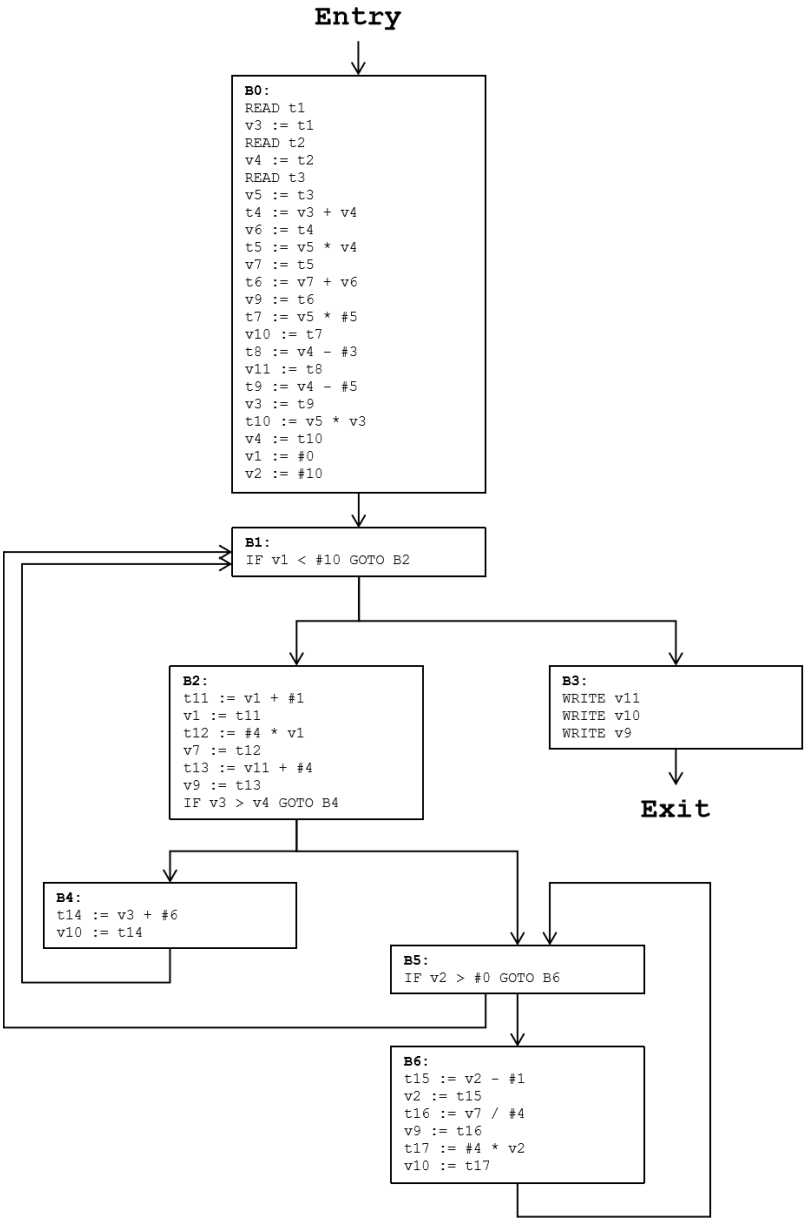


图 6.75 无用代码消除后的控制流图

全局优化 4 循环不变代码外提

在中间代码优化中，针对循环的优化是一个重要的课题。循环内部的语句通常比其他语句执行的次数更多，如果我们仅把不必出现在循环内部的语句移动到外部，也能起到较好的优化效果。**循环不变代码外提（Loop Invariant Code Motion，LICM）**关注那些每一次执行循环，得到的结果都不变的语句，对于这种语句，我们或许可以将他们移动到循环外部，从而完成优化过程。

本书讨论的循环为**自然循环（Natural Loops）**，自然循环是满足以下性质的循环：

1. 自然循环有唯一的入口结点，称为**首结点（Header）**，首结点支配循环内部的所有节点。

2. 循环内部至少存在一条指向首结点的**回边（Back Edges）**，若存在两节点 m 和 n，m dom n，存在有向边 e：n→m，则有向边 e 被称为回边。若不存在回边，则不构成循环。

在设计优化之前，我们首先要关注，有哪些语句被认为是可被移动的。循环不变的代码外提是一件较为简单的事情：对于一个赋值语句等号右边的表达式中，构成该表达式的每一个变量，使用到达定值模式进行分析，如果这些变量都在循环外部被定义，那么这条语句就是可被移动的。

在通常情况下，且不论在算法中是如何定义“循环”的，可移动代码有很多需要关注的性质：

1. 被移动前是不可达代码

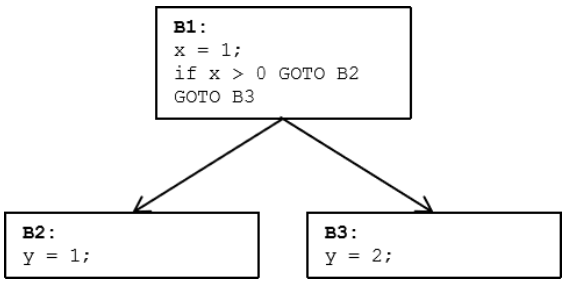


图 6.76 可达代码示例

例如，对以上的这段代码，基本块 B3 中的代码均为不可达代码，如果在不可达代码消除之前将语句移动到其他位置，反而增加了冗余代码，更可能导致程序语义的改变。因此，在我们的实践技术中设计的优化管道，在循环不变代码外提和无用代码消除之前，设计了常量折叠和控制流优化，进行不可达代码的消除。如果我们选择只完成这一部分的优化，也可设计一些简单的全局扫描与常量折叠算法，进行不可达代码的判断与消除。

2. 移动前，循环内部对该变量的赋值是唯一的

对于图 6.77 中的代码，循环内部存在两条对 x 的赋值语句，无论将哪一条外提都会对语义产生影响。

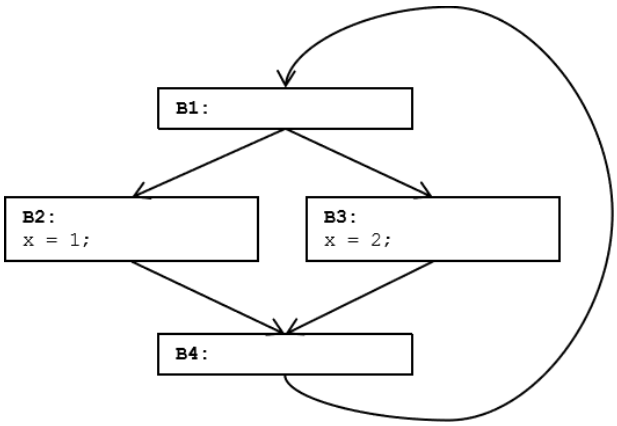


图 6.77 循环内部仅存在一条赋值语句示例

3. 移动前，对变量进行 def 之前，循环内部存在 use

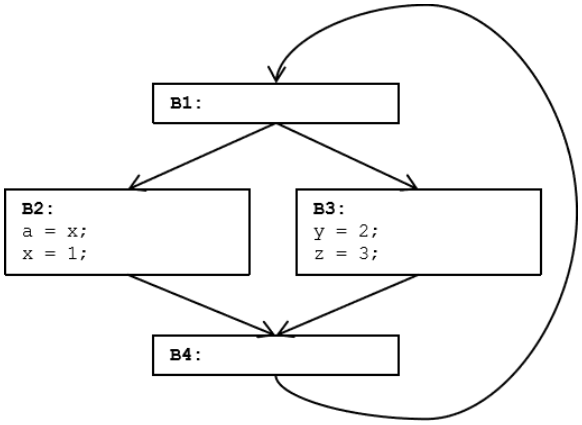


图 6.78 循环内部存在 use 的示例

如基本块中对 x 的赋值，在赋值之前存在对 x 的使用，若将其简单地外提到循环的外部，则会改变 x 的取值，从而改变语义。

4. 变量定义的基本块能够支配所有的循环出口

例如，对于图 6.78 中的代码，循环存在两个出口基本块 B2 与 B4，虽然变量 y 与 z 均在循环的外部被定义，但是该语句并不能被移动到 B1 或更前的基本块（但该语句或许可被移动到基本

块 B6）。在实践中，本书只要求将代码移动到基本块入口、入口的前驱基本块或在入口之前新构造一个基本块中，而如果算法足够优秀，也可将其移动到后继的基本块中。

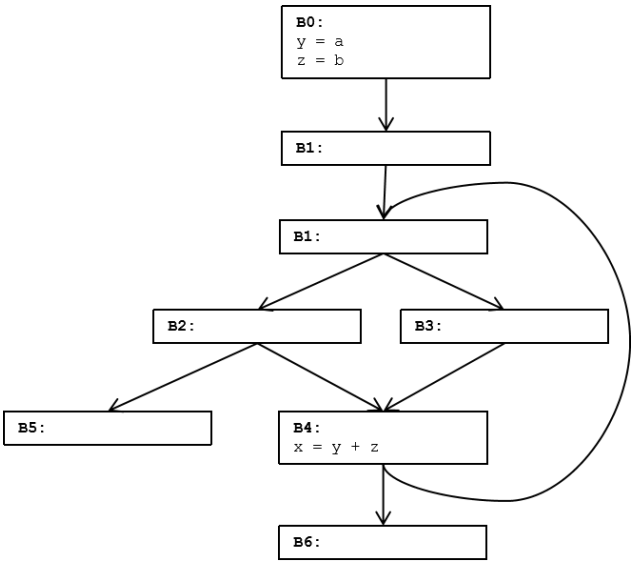


图 6.79 循环内部定义支配所有 use 的示例

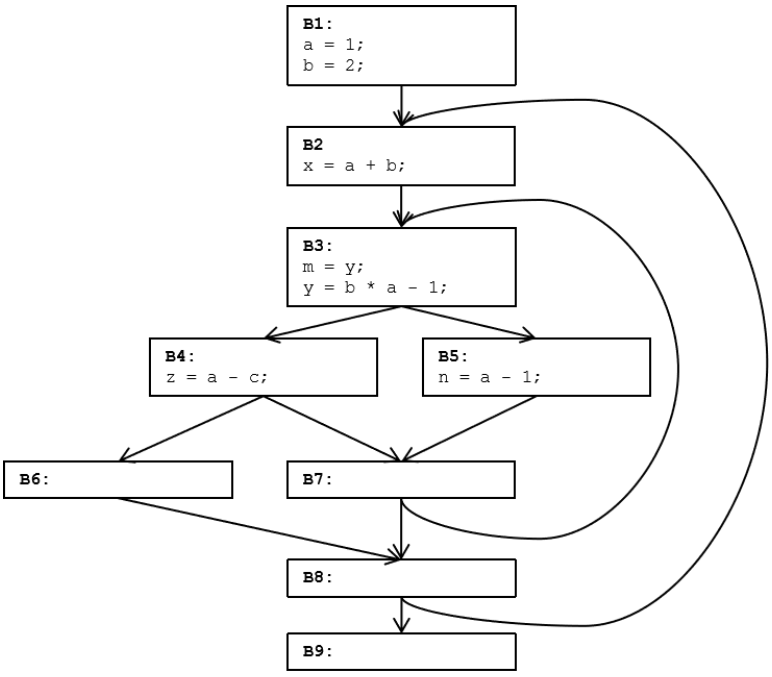


图 6.80 循环不变代码外提示例

因此，可移动的代码需要满足以下的特性：

1. 循环不变

2. 所处的基本块能够支配所有的出口基本块

3. 循环内部不存在其他对该变量的赋值

4. 所处的基本块能够支配所有存在该变量使用语句的基本块

对于支配的相关概念，请翻阅本书常量传播中有关稀疏常量传播的部分。

同时，循环不变代码外提中，我们通常都是从最内层的循环进行代码外提，以下是一个由里到外进行循环不变代码外提的例子如图 6.80。

我们首先构造循环代码的支配树如图 6.81 所示。

图 6.79 中的这段代码，包含了两个循环：{B3,B4,B5,B7}和{B2,B3,B4,B5,B6,B7,B8}，我们将循环{B3,B4,B5,B7}标记为 L1，{B2,B3,B4,B5,B6,B7,B8}标记为 L2，循环 L1 存在两个“出口”B4 与 B7，循环 L2 仅存在一个“出口”B8。

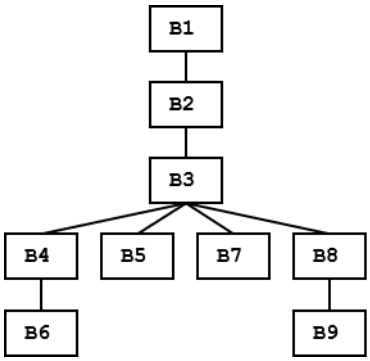


图 6.81 支配树

我们基于从里到外的顺序，先处理内层循环 L1：

1.循环 L1 内部能够支配所有出口的基本块仅有 B3，其中存在的语句：

m＝y;

y＝b＊a - 1;

基本块 B4、B5 中的语句不为可被外提的语句。

2.循环 L1 内部不存在其他对变量的赋值：

变量 y，m 在循环 L1 中均不存在其他的赋值。

3.循环 L1 内部赋值语句产生的变量的 def关系能够支配所有对该变量的 use：

对变量 y 的赋值因不能支配所有的 use，故不能作为可被外提的语句。

4.对赋值语句等号右边的表达式中存在的变量使用到达定值模式进行分析：

m＝y 中变量 y 的值非定值，因此也不可以作为循环不变的代码外提。

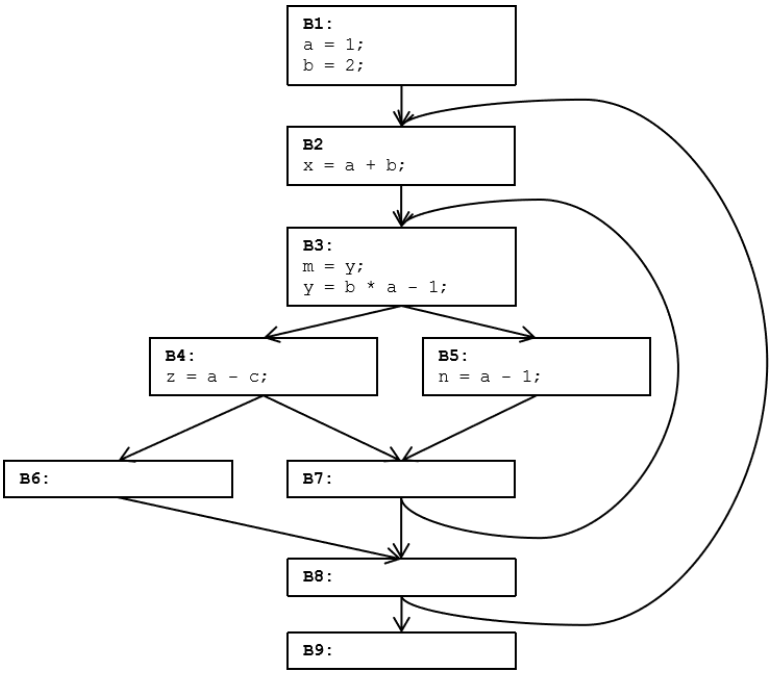


图 6.82 循环不变代码外提结果

经过 4 个步骤，循环 L1 处理完毕，然后我们处理循环 L2：

1.循环 L2 内部能够支配所有出口的基本块有 B2 和 B3，其中存在的语句：

x＝a＋b;

m＝y;

y＝b * a - 1;

2.循环 L2 内部不存在其他对变量的赋值：

变量 x，y，m 在循环 L2 中均不存在其他的赋值。

3.循环 L2 内部赋值语句产生的变量的 def关系能够支配所有对该变量的 use：

基本块 B3 中的变量已被分析过，而变量 x 在循环中不存在 use，因此满足条件。

4.对赋值语句的等号右边的表达式中存在的变量使用到达定值模式进行分析：

赋值语句 x=a+b 等号右边的表达式 a+b 中存在的变量 a 与 b，使用到达定值模式分析后，其值均为定值，因此可作为循环不变的代码外提。

对循环的分析结束后，构成的控制流图如图 6.82 所示。

我们将语句 x=a+b 从基本块 B2 移除，将其移动到循环“入口”的前驱基本块 B1 中。有时，循环的入口存在多个前驱基本块，此时我们可额外增加一个基本块，并将语句置于其中。

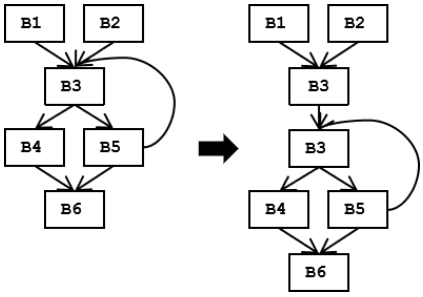


图 6.83 添加一个基本块

在实践中，我们需要基于到达定值模式及上述可移动代码的相关性质，迭代式地进行代码移动，生成的代码可以如图 6.84 所示。

相较于前三种优化，与循环相关的优化工作量较大，所需实现的算法也较多，存在一定难度。

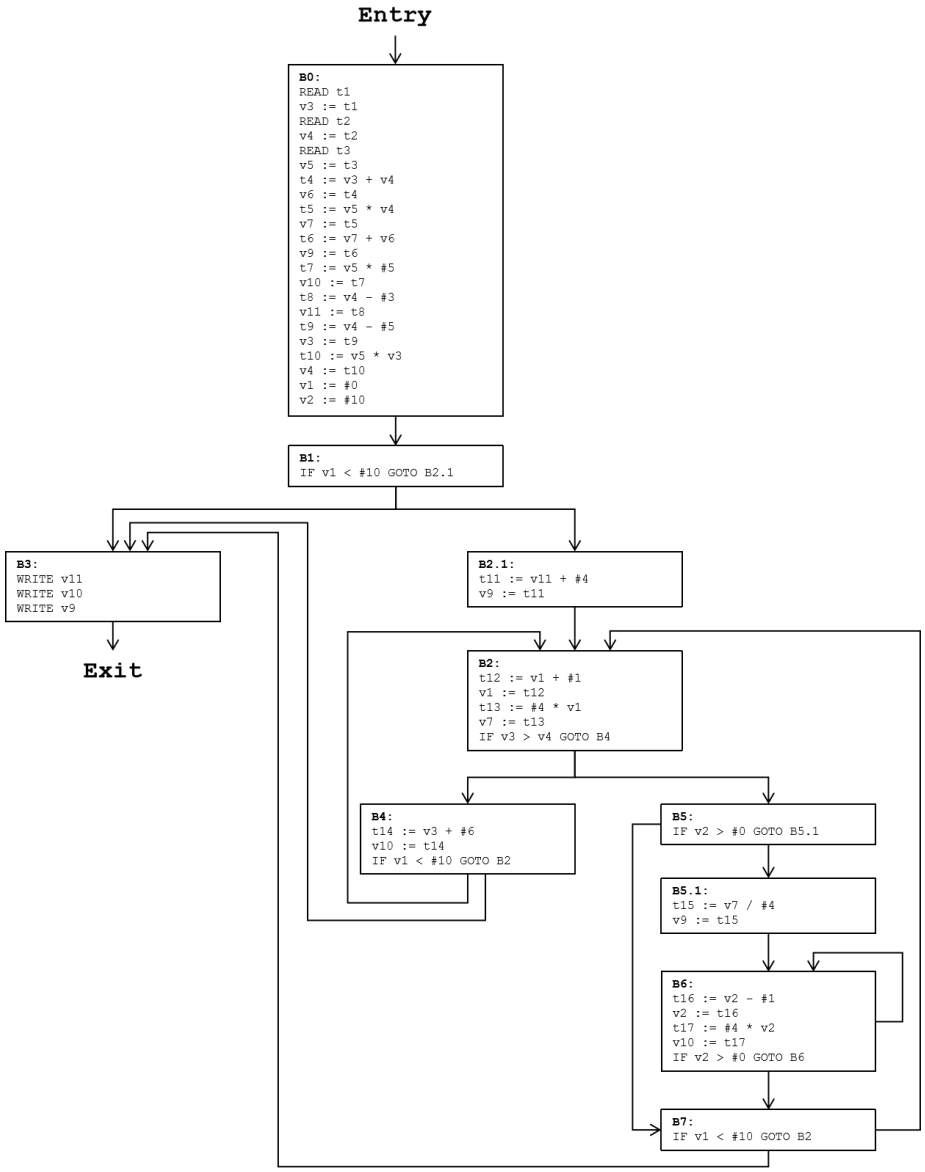


图 6.84 循环不变代码外提后的控制流图

全局优化 5 归纳变量强度削减：

本书关注的中另一个循环相关的优化是**归纳变量强度削减（Induction Variables Strength Reduction）**。在优化管道的设计中，归纳变量强度削减通常跟随在循环不变代码外提之后。

基础归纳变量（Basic Induction Variable）是在循环内部每次赋值都加或减一个常数 c，并且该变量不能被替换成常数，且不是循环不变的变量，如：

`x=x+c` 或 `x=x-c`，c 为一常量

则，**归纳变量（Induction Variable）**可能是：

1. 一个基础归纳变量 x；

2. 在循环中仅有一条针对该变量的赋值语句，且赋值语句等号右边的表达式为一个基础归纳变量的线性方程组，形如 `y=x*c1+c2`，其中 c1 与 c2 为常量，x 为一基础归纳变量。

构成基础归纳变量 a 的家族被定义为：一个变量构成的集合 A，对于 A 中的任意变量 b，在循环内部对 b 的赋值语句均为基础归纳变量 a 的线性方程组。

我们首先要关注，有哪些变量可被看作能够进行强度削减的变量。

对于基础归纳变量 a 的家族集合中的一个变量 b，b 能够被表示为：

`b=a`**`op`**`c1+c2`，其中 **op** 表示任何二元运算

当程序内部存在以下的语句：

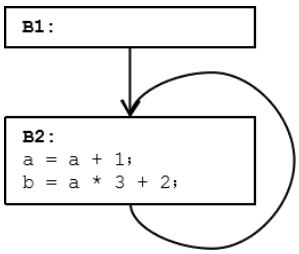


图 6.85 代码示例

我们可以使用 b’代替 b，使用代数恒等式变化将代码转变为：

$$b' = b' + 3 * 1;$$

则代码将转化为：

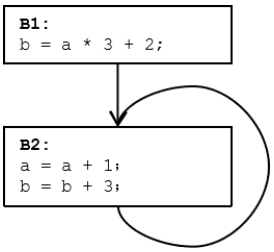


图 6.86 强度削减

对于可进行强度削减的变量，同样存在一些特性：

1.对变量的赋值能够支配所有的使用（与循环不变代码外提相同）；

2.对于基础归纳变量 a 与其家族的成员 b，不存在循环外部的 a 对 b 的赋值产生影响；

3.循环内部不存在形如 a＝a＋c（c 为常量）之外对 a 的赋值语句，且对基础归纳变量的赋值语句能够支配所有对家族内部变量的赋值语句。

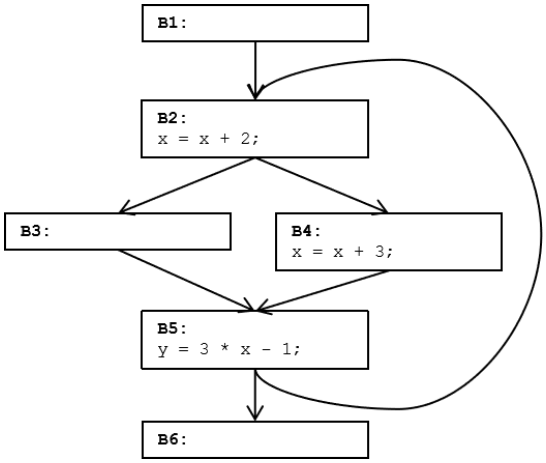


图 6.87 不能进行强度削减的示例

例如，对于图 6.87 的这段代码，B4 中对基础归纳变量 x 的赋值语句不支配 B5 中对 y 的赋值，因而不能进行归纳变量强度削减。

以下是一个归纳变量强度削减的例子：

1.对于图 6.88 中的代码，构造支配树（见稀疏常量传播中支配树相关）如图 6.89 所示；

2.基于支配树，寻找基础归纳变量，寻找到变量 x；

3.迭代地构造基础归纳变量 x 的家族；

4.使用代数恒等式进行替换，实现强度削减。

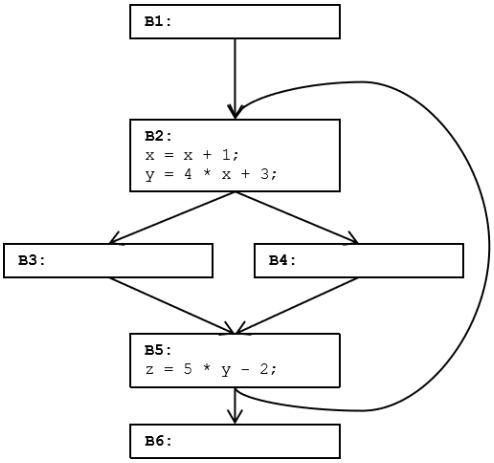


图 6.88 归纳变量强度削减示例

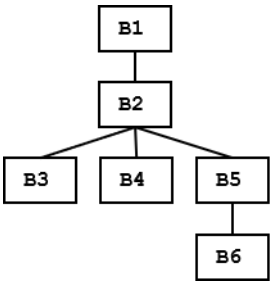


图 6.89 支配树

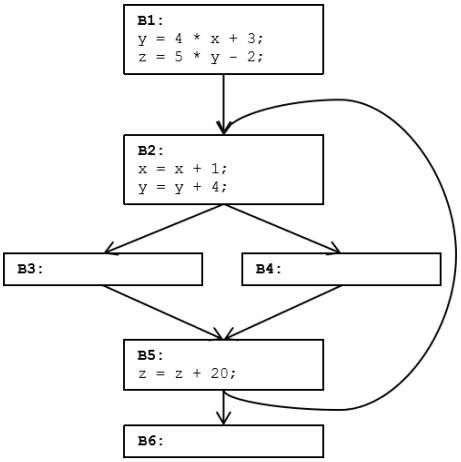


图 6.90 强度削减结果

在实践中，我们需要基于到达定值算法及上述归纳变量的相关性质，迭代式地进行强度削减，生成的代码可以如图 6.91 所示。

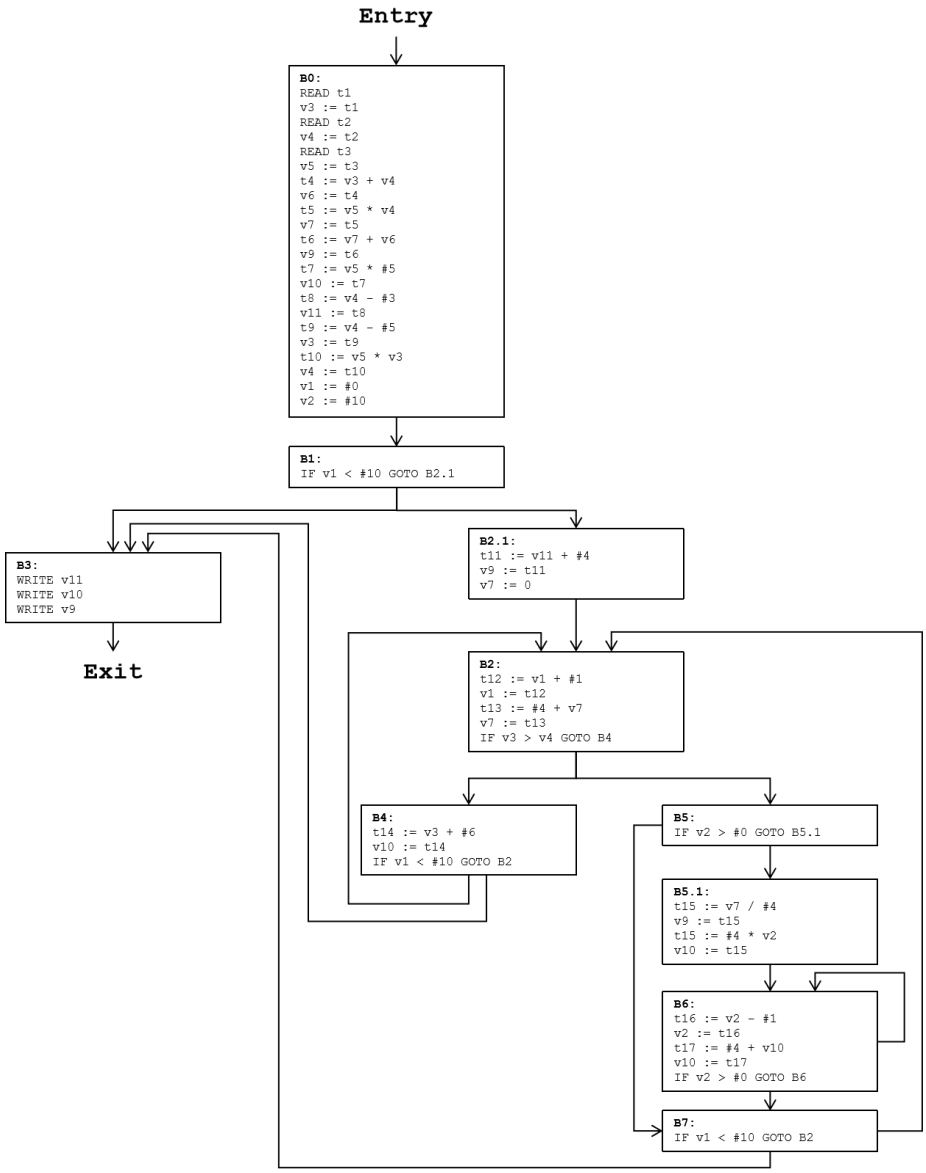


图 6.91 强度削减后的控制流图

我们可基于以上五个全局优化，尝试构造优化管道并实现优化算法。我们需要使得优化后的中间代码在执行时经过尽可能少且开销较少的语句，并且确保生成的中间代码的语义不发生改变。

6.2.3 过程间优化

至今为止，我们所提到的优化都为**过程内优化（Intra-Procedural Optimization）**。过程内优化只关注单个函数内部的程序优化，而相较于前文提到的局部优化（基本块内部）与全局优化（函数内部），**过程间优化（Inter-Procedural Optimization）**将优化范围扩大到多个函数，关注数据在调用者与被调用者之间的流动。过程间优化作为拓展阅读，不包含在本章实践中。

部分优化，如指针相关的优化，必须基于过程间优化进行。出于安全性的考虑，我们认为一次函数调用可能改变任何可被访问的指针变量所指向的内容，如果仅使用过程内分析进行程序优化，对指针相关的优化将极其保守，且十分低效。

```
int func (int x)
{
  void *p = (void*) x;
  int r = (int)p;
  return r;
}
```

C 语言中，int 类型的值可轻易与指针类型互转，若使用过程内分析的方式分析函数调用后的程序状态，会导致指针分析结果的全面失效。

调用图（Call Graph）：

进行过程间优化，我们首先需要了解，程序内部在哪些地方进行函数调用、调用者和被调用者分别是哪个函数及哪些数据在调用之间流动。为此，我们使用一个与过程内控制流图类似的图来表示函数调用之间的关系：调用图。

```
f1(){
  1:f2();
  2:f3();
  3:f3();
}

f2(){
  4:f3();
  5:f4();
}

f3(){
  6:f1();
  7:f4():
}

f4(){

}
```

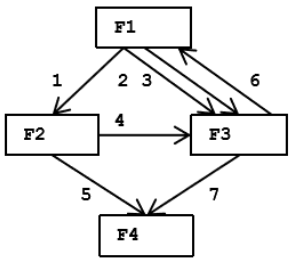


图 6.92 示例代码及其对应的调用图

例如，对于以上这段代码，共有七个函数调用，构造的函数调用图如右所示。有了调用图，我们就可以开始从事过程间的分析与优化。

函数内联（Inlining）：

进行过程间的分析，首先能被想到的，自然是将函数的调用与返回加入到一个大的控制流图中，以“过程内”的方式进行过程间的分析。

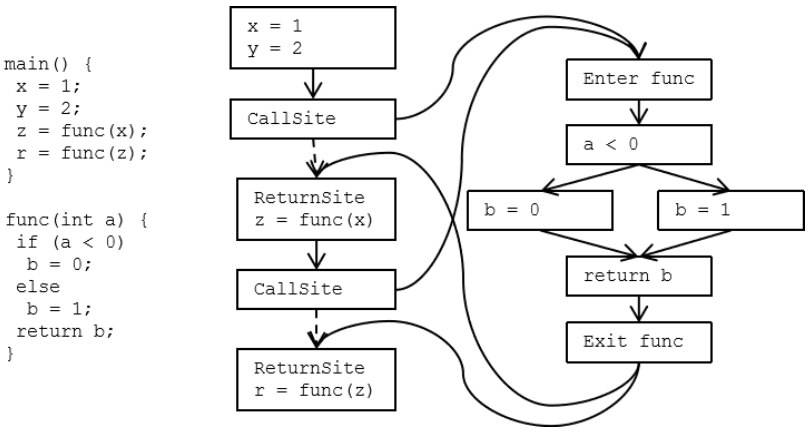


图 6.93 函数调用示例

在进行分析时，我们将被调用的函数内联进来：

```
main() {
    x = 1;
    y = 2;
    if (x < 0)
        b1 = 0;
    else
        b1 = 1;
    z = b1;
    if (z < 0)
        b2 = 0;
    else
        b2 = 1;
    r = b2;
}
```

这样的分析方式简单，易于理解，也是进行过程间分析的主要方式之一。然而，这种方法存在着开销昂贵的问题，且对于部分场景难以使用。因为，函数中存在着大量循环与函数的嵌套调用，使用内联进行分析时会使得代码规模指数性增长，进行分析与优化时开销极大，如下面这段代码所示。

```
main() {
     while(a > 0){
          func1();
     }

     func1(){
          func2();
     }
     ... ...
}

func1(x){
     ... ...
     func1(x-1);
}
```

函数的递归调用，难以使用函数内联的方式进行分析。

函数摘要（Summary）：

基于摘要的过程间分析是另一种进行分析的方式。在进行过程内的优化时，我们可以同步收集函数的各个输入参数从函数入口到出口之间的状态改变。对于不同的优化需求，我们构造相对应的抽象与内存模型，然后基于数据流分析或值流分析的方式，构造函数入口的程序状态与函数出口的程序状态之间的状态映射。

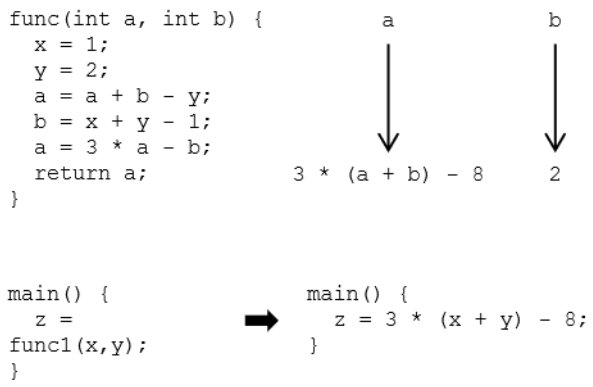


图 6.94 函数摘要示例

在进行优化时，采用由内到外的顺序，依次基于函数入口和出口的程序状态之间的映射关系，进行程序状态的更新与计算，进而实现程序的优化工作。

过程间优化广泛用于指针相关优化、Java 等面向对象的语言对虚方法的调用优化和并行代码的优化等，现有的过程间优化基于可达性分析、值流分析等理论，虽然在本章实践中没有要求，但是学有余力的同学可自行查阅资料进行进一步的学习与理解。

6.3 中间代码优化的实践内容

本章为实践内容五，任务是在词法分析、语法分析、语义分析和中间代码生成程序的基础上，使用数据流分析算法等消除效率低下和无法被轻易转化为机器代码的中间代码，从而将 C--源代码翻译成的中间代码转化为语义等价但是更加简洁高效的版本。本实践内容是实践内容三（中间代码生成）的延续，将针对生成代码的语义一致性及代码的简洁性和高效性进行检查。

需要注意的是，由于本次实践内容的代码会与之前实践内容中已经写好的代码进行对接，因此保持一个良好的代码风格、系统地设计代码结构和各模块之间的接口对于整个实践来讲相当重要。

6.3.1 实践要求

在本次实践中，对于给定的程序，在实践内容三生成的 IR 基础上，对 IR 进行中间代码优化。实践假设同实践三中间代码生成部分，输入的源程序不包含词法语法语义分析错误，可以正常生成 IR。对于给定的 IR，程序要能够进行如下的中间代码优化：公共子表达式消除、无用代码消除、常量传播、循环不变表达式外提、强度削减、控制流图优化等。

1) **要求 6.1**：完成基于数据流分析模式的优化：

a) **公共子表达式消除**：避免重复计算已经在先前计算过的表达式。

b) **无用代码消除**：删除执行不到的基本块和指令、仅存储但不使用的变量等。

c) **常量传播**：对于值恒为常量的表达式进行常量替换。

基于 6.1.3-6.1.5 节所述数据流分析模式和 6.2.2 节所述全局优化的相关理论，完成三种优化模块设计和实现。程序需要输出与输入中间代码语义相同且更精炼的中间代码，并且优化的代码应只包含上述三种优化能够覆盖的中间代码。

2) **要求 6.2**：完成循环不变代码外提优化，将每次执行结果都不变的表达式，移动到循环外部。我们的程序应能够基于 6.2.2 节中所述循环不变代码外提的相关理论，进行循环不变代码外提优化。我们的程序需要输出与输入中间代码语义相同且更精炼的中间代码，并且优化的代码应只包含循环不变代码外提优化能够覆盖的中间代码。

3) **要求 6.3：**完成归纳变量强度削减优化，将循环内部与归纳变量相关的高代价运算转换为低代价的运算。我们的程序应能够基于 6.2.2 节中所述归纳变量强度削减的相关理论，进行归纳变量强度削减优化。我们的程序需要输出与输入中间代码语义相同且更精炼的中间代码，并且优化的代码应只包含归纳变量强度削减优化能够覆盖的中间代码。

优化模块还需包含控制流图简化，去除空基本块，优化执行顺序。

实践内容五主要考察程序输出的中间代码的执行效率，因此需要考虑如何优化中间代码的生成，基于后文所述的中间代码优化方法进行代码优化。虚拟机将根据优化之后的代码行数与操作数量评估优化效果。

6.3.2 输入格式

我们的程序的输入是一个实践内容三输出的中间表达文本文件。输入文件每行一条中间代码，如果包含多个函数定义，则通过 FUNCTION 语句将这些函数隔开。程序需要能够接收一个输入文件名和一个输出文件名作为参数。例如，假设程序名为 cc、输入文件名为 input.ir、输出文件名为 output.ir、程序和输入文件都位于当前目录下，那么在 Linux 命令行下运行./cc input.ir output.ir 即可获得以 input.ir 作为输入文件的输出结果。

6.3.3 输出格式

实践内容五在实践内容三的基础上进行中间代码优化，输出格式与实践内容三相同。我们将使用虚拟机小程序统计优化后的中间代码所执行过的各种操作的次数，以此来估计程序生成的中间代码的效率。

6.3.4 验证环境

程序将在如下环境中被编译并运行（同实践内容一）：

1) GNU Linux Release: Ubuntu 20.04, kernel version 5.4.0-28-generic；

2) GCC version 7.5.0；

3) GNU Flex version 2.6.4；

4) GNU Bison version 3.0.4。

5) 虚拟机（基于 Python3.8 实现）。

一般而言，只要避免使用过于冷门的特性，使用其它版本的 Linux 或者 GCC 等，也基本上不会出现兼容性方面的问题。注意，实践内容五的检查过程中不会去安装或尝试引用各类方便编程的函数库（如 glib 等），因此请不要在你的程序中使用它们。在实验报告中，请标注所使用的版本。

6.3.5 提交要求

实践内容五要求提交如下内容（同实践内容一）：

1) Flex、Bison 以及 C 语言的可被正确编译运行的源程序。

2) 一份 PDF 格式的实验报告，内容包括：

a) 程序实现了哪些功能？简要说明如何实现这些功能。清晰的说明有助于助教对程序所实现的功能进行合理的测试。

b) 程序应该如何被编译？可以使用脚本、makefile 或逐条输入命令进行编译，请详细说明应该如何编译程序。无法顺利编译将导致助教无法对程序所实现的功能进行任何测试，从而丢失相应的分数。

c) 实验报告的长度不得超过三页。所以实验报告中需要重点描述的是程序中的亮点、最个性化、最具独创性的内容，而相对简单的、任何人都可以做的内容则可不提或简单地提一下，尤其要避免大段地向报告里贴代码。实验报告中所出现的最小字号不得小于五号字（或英文 11 号字）。

6.3.6 样例 （必做内容）

实践内容五的样例包括**必做内容样例**与**选做要求样例**两部分，分别对应于实践内容要求中的必做内容和选做要求。请仔细阅读样例，以加深对实践内容要求以及输出格式要求的理解。这节

列举必做内容样例。

样例 1（局部公共子表达式消除）：

输入：

```
1   FUNCTION main :
2   READ t1
3   v1 := t1
4   READ t2
5   v2 := t2
6   t3 := v1 + #2
7   v3 := t3
8   t4 := v3 + #1
9   v4 := t4
10  t5 := v1 + #3
11  v5 := t5
12  t6 := v1 + v2
13  v6 := t6
14  t7 := v1 + #2
15  v7 := t7
16  t8 := v1 + #2
17  v2 := t8
18  t9 := #0
19  RETURN t9
```

输出：

对于这段只包含单个基本块的中间代码，对基本块中表达式构造有向无环图，并且运用代数恒等式转换。我们检查有向无环图中一个节点 N，是否与另一个节点 M 具有相同的运算符与子节点，若相同则消除公共子表达式，生成的中间代码可以是这样的：

```
1   FUNCTION main :
2   READ t1
3   v1 := t1
4   READ t2
5   v2 := t2
6   t3 := v1 + #2
7   v3 := t3
8   t4 := v3 + #1
9   v4 := t4
10  t5 := v1 + #3
11  v5 := t5
12  t6 := v1 + v2
13  v6 := t6
14  v7 := t3
15  v2 := t3
16  t9 := #0
17  RETURN t9
```

如果你的方法足够聪明，你会发现 t4 与 t5 也是相同的，通过设计合适的框架，能够使得公共子表达式的消除效果更好。

样例 2（局部无用代码消除）：

输入：

```
1   FUNCTION main :
2   READ t1
3   v1 := t1
4   READ t2
5   v2 := t2
6   v3 := #2
7   v4 := #3
8   t3 := v2 + v3
9   v1 := t3
10  t4 := v1 - v4
11  v2 := t4
12  t5 := v2 + v3
13  v3 := t5
14  t6 := v1 - v4
15  v4 := t6
16  t7 := v1 + v3
17  v5 := t7
18  t8 := v1 - v4
19  v6 := t8
20  t9 := v4 + v6
21  v7 := t9
22  t10 := v2 + v1
23  v1 := t10
24  t11 := v7 + v6
25  v8 := t11
26  WRITE v8
27  t12 := v1 + v4
28  v9 := t12
29  WRITE v9
30  t13 := #0
31  RETURN t13
```

输出：

对于 v1-v7 七个变量，构造有向无环图。我们基于有向无环图，迭代地去除没有父节点且未被使用的根节点，消除无用代码，因此生成的中间代码可以是这样的：

```
1   FUNCTION main :
2   READ t1
3   READ t2
4   v2 := t2
5   v3 := #2
6   v4 := #3
7   t3 := v2 + v3
8   v1 := t3
9   t4 := v1 - v4
10  v2 := t4
11  t5:= v2 + v3
12  v3 := t5
13  t6 := v1 - v4
14  v4 := t6
15  t7 := v1 - v4
16  v6 := t7
17  t8 := v4 + v6
```

```
18  v7 := t8
19  t9 := v2 + v1
20  v1 := t9
21  t10 := v7 + v6
22  v8 := t10
23  WRITE v8
24  t11 := v1 + v4
25  v9 := t11
26  WRITE v9
27  t12 := #0
28  RETURN t12
```

样例 3（常量折叠）：

输入：

```
1   FUNCTION main :
2   DEC v6 40
3   v1 := #30
4   t1 := v1 / #5
5   v2 := t1
6   t2 := #9 - v2
7   v2 := t2
8   t3 := v2 * #4
9   v3 := t3
10  t4 := #2 * v3
11  t5 := v1 - t4
12  v4 := t5
13  t6 := v4 * #4
14  t7 := &v6 + t6
15  t8 := *t7
16  v5 := t8
17  RETURN #0
```

输出：

在编译时，我们使用常量折叠技术，识别并计算常量表达式，以避免在运行时计算他们的值。通常，我们迭代式地遍历中间代码，基于复制传播及常量传播技术，用常量代替一切识别为常量的变量或是表达式。对于以上的中间代码，优化后中间代码可以是这样的：

```
1   FUNCTION main :
2   DEC v6 40
3   v1 := #30
4   t1 := #6
5   v2 := #6
6   t2 := #3
7   v2 := #3
8   t3 := #12
9   v3 := #12
10  t4 := #24
11  t5 := #6
12  v4 := #6
13  t6 := #24
14  t7 := &v6 + #24
15  t8 := *t7
```

```
16  v5 := t8
17  RETURN #0
```

6.3.7 样例（选做要求）

这节列举选做要求样例。

样例 1（全局公共子表达式消除）：

输入：

```
1   FUNCTION main :
2   READ t1
3   v1 := t1
4   READ t2
5   v2 := t2
6   t3 := v1 + #1
7   v3 := t3
8   t4 := v2 + #2
9   v4 := t4
10  t5 := v1 + v2
11  v5 := t5
12  t6 := v5 - v3
13  v6 := t6
14  t7 := v1 + v3
15  v7 := t7
16  t8 := v1 + v6
17  v4 := t8
18  v8 := #0
19  LABEL label1 :
20  IF v8 < v1 GOTO label2
21  GOTO label3
22  LABEL label2 :
23  t9 := v8 + #1
24  v8 := t9
25  IF v1 < v2 GOTO label4
26  GOTO label5
27  LABEL label4 :
28  t10 := v1 + v2
29  v8 := t10
30  t11 := v1 + #1
31  v6 := t11
32  GOTO label6
33  LABEL label5 :
34  t12 := v2 + v3
35  v1 := t12
36  t13 := v1 + v2
37  v5 := t13
38  LABEL label6 :
39  GOTO label1
40  LABEL label3 :
41  RETURN #0
```

输出：

对于全局的公共子表达式的消除，通常我们采用可用表达式模式，分析到达某一程序点 p 的

所有路径上，是否存在可用的表达式。全局公共子表达式消除后，中间代码可以是这样的：

```
1   FUNCTION main :
2   READ t1
3   v1 := t1
4   READ t2
5   v2 := t2
6   t3 := v1 + #1
7   v3 := t3
8   t4 := v2 + #2
9   v4 := t4
10  t5 := v1 + v2
11  v5 := t5
12  t6 := v5 - v3
13  v6 := t6
14  t7 := v1 + v3
15  v7 := t7
16  t8 := v1 + v6
17  v4 := t8
18  v8 := #0
19  LABEL label1 :
20  IF v8 < v1 GOTO label2
21  GOTO label3
22  LABEL label2 :
23  t9 := v8 + #1
24  v8 := t9
25  IF v1 < v2 GOTO label4
26  GOTO label5
27  LABEL label4 :
28  v8 := v5
29  t10 := v1 + #1
30  v6 := t10
31  GOTO label6
32  LABEL label5 :
33  t11 := v2 + v3
34  v1 := t11
35  t12 := v1 + v2
36  v5 := t12
37  LABEL label6 :
38  GOTO label1
39  LABEL label3 :
40  RETURN #0
```

样例 2（常量传播）：

输入：

```
1   FUNCTION main :
2   READ t1
3   v1 := t1
4   READ t2
5   v2 := t2
6   t3 := v1 + #1
7   v3 := t3
8   v4 := #2
9   t4 := v3 * v4
10  v5 := t4
```

```
11  t5 := #2 + v3
12  v8 := t5
13  t6 := v3 * #2
14  v7 := t6
15  t7 := #2 * v3
16  t8 := t7 + #4
17  v9 := t8
18  IF v9 == v7 GOTO label2
19  LABEL label1 :
20  v3 := #3
21  t9 := v4 + #5
22  v5 := t9
23  t10 := #2 * v8
24  v1 := t10
25  t11 := v9 - v1
26  v2 := t11
27  GOTO label3
28  LABEL label2 :
29  v2 := v3
30  t12 := v5 + v2
31  v1 := t12
32  t13 := v7 * v3
33  v9 := t13
34  LABEL label3 :
35  t14 := v1 + v2
36  v8 := t14
37  v7 := v3
38  t15 := v5 - v2
39  v9 := t15
40  RETURN #0
```

输出：

对于全局常量传播的计算，在实践中，我们仅要求读者能够熟练使用简单常量传播算法。在常量替换后，中间代码可以是这样的：

```
1   FUNCTION main :
2   READ t1
3   v1 := t1
4   READ t2
5   v2 := t2
6   t3 := v1 + #1
7   v3 := t3
8   v4 := #2
9   t4 := v3 * #2
10  v5 := t4
11  t5 := #2 + v3
12  v8 := t5
13  t6 := v3 * #2
14  v7 := t6
15  t7 := #2 * v3
16  t8 := t7 + #4
17  v9 := t8
18  IF v9 == v7 GOTO label2
19  LABEL label1 :
20  v3 := #3
21  v5 := #7
22  t9 := #2 * v8
```

```
23  v1 := t9
24  t10 := v9 - v1
25  v2 := t10
26  GOTO label3
27  LABEL label2 :
28  v2 := v3
29  t11 := v5 + v2
30  v1 := t11
31  t12 := v7 * v3
32  v9 := t12
33  LABEL label3 :
34  t13 := v1 + v2
35  v8 := t13
36  v7 := v3
37  t14 := v5 - v2
38  v9 := t14
39  RETURN #0
```

如果我们使用本书上介绍的条件常量传播算法，并且采取更加智能的常量传播框架，可以发现，以上的中间代码中，v9 != v7 恒成立，程序只会执行 label1 分支的语句。因此你可以将中间代码优化为如下的形式。

包含变量的表达式存在不同的情况，在简化时需注意（例如 a+1 与 a-1，a+1>a-1 并不恒成立）。

```
1   FUNCTION main :
2   READ t1
3   v1 := t1
4   READ t2
5   v2 := t2
6   t3 := v1 + #1
7   v3 := t3
8   v4 := #2
9   t4 := v3 * #2
10  v5 := t4
11  t5 := #2 + v3
12  v8 := t5
13  t6 := v3 * #2
14  v7 := t6
15  t7 := #2 * v3
16  t8 := t7 + #4
17  v9 := t8
18  v3 := #3
19  v5 := #7
20  t9 := #2 * v8
21  v1 := t9
22  v2 := #0
23  v8 := v1
24  v7 := v3
25  v9 := #7
26  RETURN #0
```

样例 3（全局无用代码消除）：

输入：

```
1   FUNCTION main :
2   READ t1
3   v1 := t1
4   READ t2
5   v2 := t2
6   t3 := v1 + #1
7   v3 := t3
8   t4 := v2 + #2
9   v4 := t4
10  t5 := v1 + v2
11  v5 := t5
12  v6 := #0
13  LABEL label1 :
14  IF v6 < v1 GOTO label2
15  GOTO label3
16  LABEL label2 :
17  v1 := v5
18  t6 := v1 - #1
19  v4 := t6
20  IF v3 < v4 GOTO label4
21  GOTO label5
22  LABEL label4 :
23  v3 := #4
24  v2 := v4
25  t7 := v1 + v2
26  v6 := t7
27  GOTO label6
28  LABEL label5 :
29  t8 := v3 + #3
30  v3 := t8
31  v6 := v1
32  t9 := v1 + #1
33  v6 := t9
34  LABEL label6 :
35  t10 := v6 + #3
36  v6 := t10
37  GOTO label1
38  LABEL label3 :
39  t11 := #2 * v1
40  v5 := t11
41  t12 := v2 + v5
42  v6 := t12
43  RETURN #0
```

输出：

对于一个变量在某个程序点上定义的值，若在之后的程序执行过程中未被使用，该定义语句是无用代码，可被削减。同时，对于不被执行的条件分支中的语句，也被认为是无用代码。对于无用代码消除，我们通常采取活跃变量分析的模式。生成的中间代码可以是这样的：

```
1   FUNCTION main :
2   READ t1
```

```
3    v1 := t1
4    READ t2
5    v2 := t2
6    t3 := v1 + #1
7    v3 := t3
8    t4 := v1 + v2
9    v5 := t4
10   v6 := #0
11   LABEL label1 :
12   IF v6 < v1 GOTO label2
13   GOTO label3
14   LABEL label2 :
15   v1 := v5
16   t5 := v1 - #1
17   v4 := t5
18   IF v3 < v4 GOTO label4
19   GOTO label5
20   LABEL label4 :
21   v3 := #4
22   v2 := v4
23   GOTO label6
24   LABEL label5 :
25   t6 := v3 + #3
26   v3 := t6
27   t7 := v1 + #1
28   v6 := t7
29   LABEL label6 :
30   t8 := v6 + #3
31   v6 := t8
32   GOTO label1
33   LABEL label3 :
34   t9 := #2 * v1
35   v5 := t9
36   t10 := v2 + v5
37   v6 := t10
38   RETURN #0
```

样例 4（循环不变代码外提）：

输入：

```
1    FUNCTION main :
2    READ t1
3    v1 := t1
4    READ t2
5    v2 := t2
6    t3 := v1 + #1
7    v3 := t3
8    t4 := v2 * #2
9    v4 := t5
10   t5 := v3 * v4
11   v5 := t5
12   t6 := v5 * v4
13   t7 := v1 + t6
14   v6 := t7
```

```
15  t8 := v3 - v1
16  v7 := t8
17  LABEL label1 :
18  v1 := v7
19  t9 := v7 * v4
20  v2 := t9
21  IF v3 > v4 GOTO label2
22  GOTO label3
23  LABEL label2 :
24  v3 := #4
25  t10 := v1 * #5
26  v4 := t10
27  GOTO label4
28  LABEL label3 :
29  t11 := v1 - #3
30  v8 := t11
31  LABEL label4 :
32  IF v1 > v2 GOTO label1
33  GOTO label5
34  LABEL label5 :
35  t12 := #2 * v4
36  v9 := t12
37  RETURN #0
```

输出：

程序执行的过程中，大部分的执行时间都花在循环上，因此，研究人员针对循环设计了许多编译优化技术。对于循环内部不管执行了多少次，都能得到相同结果的表达式，我们可以将其移动到循环外部，以避免循环过程中反复执行该表达式。经过循环不变代码外提后，中间代码可以是这样的：

```
1   FUNCTION main :
2   READ t1
3   v1 := t1
4   READ t2
5   v2 := t2
6   t3 := v1 + #1
7   v3 := t3
8   t4 := v2 * #2
9   v4 := t4
10  t5 := v3 * v4
11  v5 := t5
12  t6 := v5 * v4
13  t7 := v1 + t6
14  v6 := t7
15  t8 := v3 - v1
16  v7 := t8
17  v1 := v7
18  LABEL label1 :
19  t9 := v7 * v4
20  v2 := t9
21  IF v3 > v4 GOTO label2
22  GOTO label3
```

```
23  LABEL label2 :
24  v3 := #4
25  t10 := v1 * #5
26  v4 := t10
27  GOTO label4
28  LABEL label3 :
29  t11 := v1 - #3
30  v8 := t11
31  LABEL label4 :
32  IF v1 > v2 GOTO label1
33  GOTO label5
34  LABEL label5 :
35  t12 := #2 * v4
36  v9 := t12
37  RETURN #0
```

样例 5（强度削减）：

输入：

```
1   FUNCTION main :
2   READ t1
3   v1 := t1
4   READ t2
5   v2 := t2
6   t3 := #4 * v1
7   v3 := t3
8   t4 := #4 * v2
9   v4 := t4
10  t5 := v1 - #1
11  v5 := t5
12  t6 := v2 * #2
13  v6 := t6
14  t7 := v1 + v2
15  v7 := t7
16  t8 := v3 * v5
17  v8 := t8
18  t9 := v7 - v4
19  v9 := t9
20  LABEL label1 :
21  IF v3 < v9 GOTO label2
22  GOTO label3
23  LABEL label2 :
24  t10 := v1 + #1
25  v1 := t10
26  t11 := #4 * v1
27  v3 := t11
28  t12 := v3 * v4
29  V5 := t12
30  LABEL label4 :
31  IF v4 > v5 GOTO label5
32  GOTO label6
33  LABEL label5 :
34  t13 := v2 - #1
35  v2 := t13
36  t14 := #4 * v2
37  v4 := t14
```

```
38  GOTO label4
39  LABEL label6 :
40  GOTO label1
41  LABEL label3 :
42  RETURN #0
```

输出：

对于循环内部的昂贵操作，我们可以对其做一些语义相同的转化，将“昂贵”的操作转化为相对便宜的操作。我们运用强度削减，对其中可以简化的乘法操作转化为加法或减法操作，生成的中间代码可以是这样的：

```
1   FUNCTION main :
2   READ t1
3   v1 := t1
4   READ t2
5   v2 := t2
6   t3 := #4 * v1
7   v3 := t3
8   t4 := #4 * v2
9   v4 := t4
10  t5 := v1 - #1
11  v5 := t5
12  t6 := v2 * #2
13  v6 := t6
14  t7 := v1 + v2
15  v7 := t7
16  t8 := v3 * v5
17  v8 := t8
18  t9 := v7 - v4
19  v9 := t9
20  LABEL label1 :
21  IF v3 < v9 GOTO label2
22  GOTO label3
23  LABEL label2 :
24  t10 := v1 + #1
25  v1 := t10
26  t11 := v3 + #4
27  v3 := t11
28  t12 := v3 * v4
29  V5 := t12
30  LABEL label4 :
31  IF v4 > v5 GOTO label5
32  GOTO label6
33  LABEL label5 :
34  t13 := v2 - #1
35  v2 := t13
36  t14 := v4 - #4
37  v4 := t14
38  GOTO label4
39  LABEL label6 :
40  GOTO label1
41  LABEL label3 :
42  RETURN #0
```

6.4 本章小结

本章介绍了编译器中间代码优化的关键原理与算法，中间代码优化是本书前面章节中间代码生成内容的后续。在该步骤中，主要是采用数据流分析技术对中间代码进行常量传播、公共子表达式消除、无用代码消除、循环不变代码外提等优化，以期生成性能更高的目标代码。在本章中，我们提供了中间代码优化的基础理论和实践指导，包括数据流分析理论与框架，到达定值分析，可用表达式分析，活跃变量分析，局部优化，全局优化，以及过程间优化等内容。通过本章的学习，读者可以了解到基本的程序分析技术，并掌握中间代码优化的完整流程和方法。

习题

6.1 指出下列偏序集是否是半格或格

（1）(A, |)，其中|表示正整数上的整除关系，A 是集合{1, 3, 5, 9, 15, 25, 45, 75}

（2）(E, ＞)，其中＞表示大于关系，E 表示所有偶数组成的集合

（3）哈斯图：

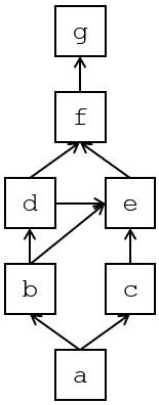


图 6.95 习题 6.1 的哈斯图

6.2 对于图 6.96 中的控制流图，结合到达定值分析算法（6.1.3），计算下列值：

（1）每个基本块的 gen 和 kill 集合。

（2）每个基本块的 IN 和 OUT 集合。

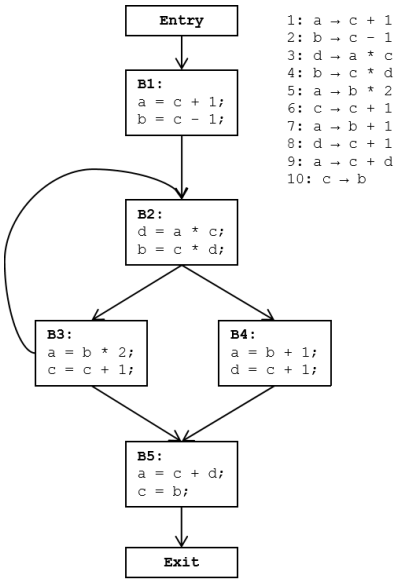


图 6.96 习题 6.2 的控制流图

6.3 对于图 6.97 中的控制流图，结合可用表达式分析算法（6.1.4），计算下列值：

（1）每个基本块的 e_gen 和 e_kill 集合。

（2）每个基本块的 IN 和 OUT 集合。

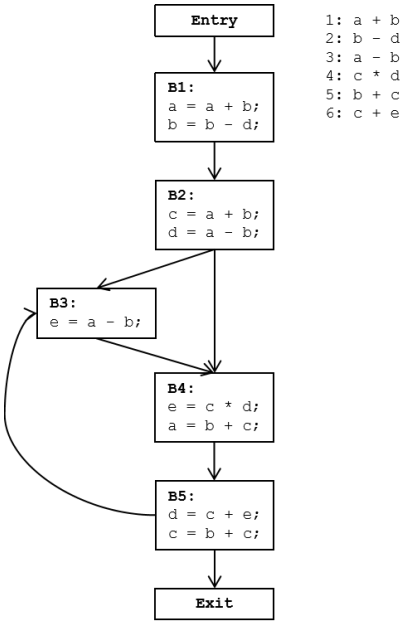


图 6.97 习题 6.3 的控制流图

6.4 对于图 6.98 中的控制流图，结合活跃变量分析算法（6.1.5），计算下列值：

（1）每个基本块的 def 和 use 集合。

（2）每个基本块的 IN 和 OUT 集合。

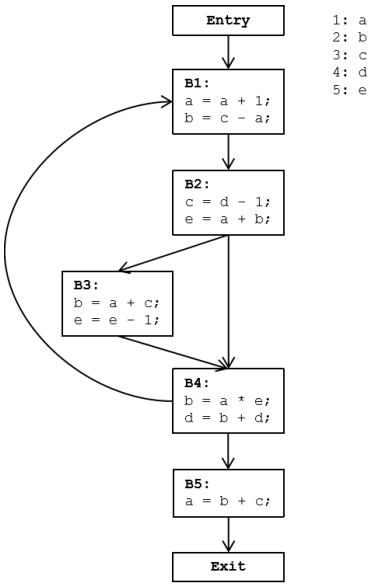


图 6.98 习题 6.4 的控制流图

6.5 为图 6.99 中的两个基本块构造 DAG

```
B1:
READ t1
v1 := t1
v2 := #1
t2 := v1 + v2
v3 := t2
v4 := #2
t3 := v3 + v4
v5 := t3
```

```
B2:
v1 := #5
v2 := #10
t1 := v1 + v2
v3 := t1
v4 := #15
t2 := v3 - v4
v5 := t2
v6 := #0
```

图 6.99 习题 6.5 的基本块

6.6 对图 6.100 中的基本块进行局部优化。

（1）构造基本块的有向无环图；

（2）对（1）中构造的有向无环图，进行局部公共子表达式消除优化；

（3）对（2）中构造的有向无环图，进行常量折叠优化。

```
B0:
v1 := #1
v2 := #2
READ t1
v3 := t1
t2 := v1 + v2
v4 := t2
t3 := v4 + v1
v5 := t3
t4 := v3 + v5
v3 := t4
t5 := v1 + v4
v2 := t5
t6 := v3 - v2
v6 := t6
```

图 6.100 习题 6.6 的基本块

6.7 对图 6.101 中的程序控制流图运行简单常量传播算法，并在每个基本块的出口处指出所有变量的状态（UNDEF，常量 c 或 NAC）。

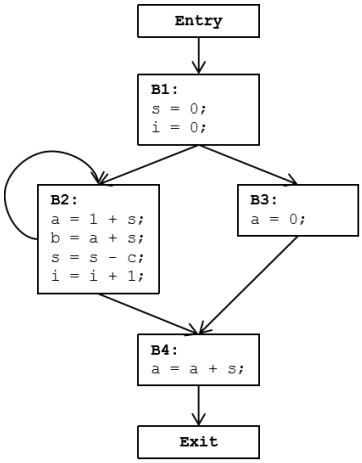


图 6.101 习题 6.7 的控制流图

6.8 对于图 6.102 中的控制流图进行全局优化。

（1）尽可能地对该控制流图中的消除全局公共子表达式；

（2）在前一小问的基础上，尽可能地对该控制流图进行复制传播优化；

（3）在前一小问的基础上，尽可能地消除该控制流图中的所有死代码。

6.9 图 6.103 中的中间代码是用来计算两个向量 A 和 B 的欧式距离的平方。按照以下全局优化管道对它尽可能地进行优化：消除公共子表达式，对归纳变量进行强度消减，消除归纳变量。

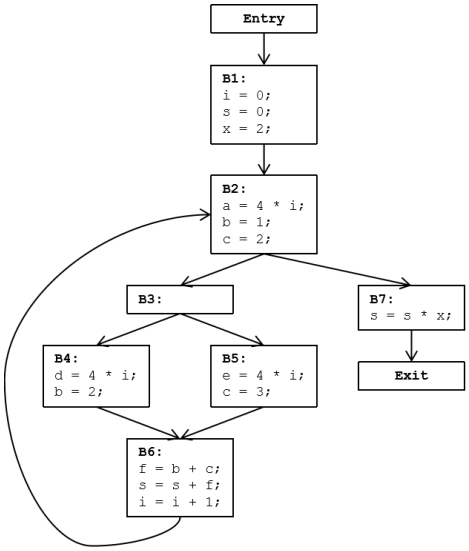


图 6.102 习题 6.8 的控制流图

```
(1)  sum = 0
(2)  i = 0
(3)  a = i * 8
(4)  b = A[a]
(5)  c = i * 8
(6)  d = B[c]
(7)  e = b - c
(8)  f = b - c
(9)  g = e * f
(10) sum = sum + g
(11) i = i + 1
(12) if i < n goto (3)
```

图 6.103 习题 6.9 的中间代码

6.10 对下面两段代码分别进行过程间优化。

（1）对下面的C代码进行函数内联优化；

```
int func1(int t2) {
  int c = -t2;
  return c;
}

int func2(int t1) {
  if (t1 > 0) {
    int a = t1 + 1;
    return a;
  } else {
    int b = func1(t1);
    return b;
  }
```

```
}

int main() {
  int x = 1;
  int y = func2(x);
  return 0;
}
```

（2）对下面的C代码进行函数摘要优化。

```
int hypot2(int x, int y) {
  int a = x * x;
  int b = y * y;
  int c = a + b;
  return a + b;
}
int main() {
  int x = ...;
  int y = ...;
  int z = hypot2(x, y);
  return 0;
}
```