第 5 章

静态单赋值形式

静态单赋值形式（Static Single-Assignment form，SSA）是现代编译器中广泛采用的一种重要中间表示，由于其单赋值的良好性质，它可以简化很多程序分析和优化，并为编译的其它阶段。本章讨论静态单赋值形式的基本概念、构建、消去、在编译优化中的应用、以及和函数式中间表示的关系。

5.1 引言

许多数据流分析需要寻找表达式中每个定值变量的使用点，或者每个使用变量的定值点。编译器为了高效表达这些信息，可以使用定值-使用链（def-use chain，DU 链）或使用-定值链（use-def chain，UD 链）的数据结构：即对控制流图中的每条语句，编译器构建并维护两个由指针组成的列表，其中一个指针指向在该语句中定值的变量的所有使用点，另一个指针指向该语句中使用的变量的所有定值点。通过这个方法，编译器能够快速地从使用跳到定值，或从定值跳到使用。

静态单赋值形式是对定值-使用链思想的一种改进的中间表示，它满足：在程序中，每个变量只有唯一一个定值。但由

于这个（静态的）定值可能位于一个可（动态）执行多次的循环中，因此我们把这种中间表示称为静态单赋值形式，而不是简单地称为单赋值形式（在单赋值形式中，变量不会被动态重新定值）。

和 UD 链相比，静态单赋值形式有以下优势。首先，静态单赋值形式可以简化优化，即如果每个变量只有一个静态定值时，则数据流分析和优化算法的实现更加简单。其次，静态单赋值形式可以节约空间，如果一个变量有 N 个使用和 M 个定值（占了程序中大约 $N + M$ 条指令），表示 UD 链所需要的空间（和时间）和 $N * M$ 成正比，即具有平方量级。但是对于几乎所有的实际程序，静态单赋值形式的大小和原始程序的大小成线性关系。再次，静态单赋值形式可以简化编译的其它阶段，在静态单赋值形式形式中，变量的使用和定值和控制流图的支配节点结构存在紧密联系，这简化了编译的其它阶段；例如，在静态单赋值形式上构建的干涉图都是弦图（Chordal graph），而不是一般的无向图。最后，静态单赋值形式可以移除伪相关，源程序中同一个变量的不相关的使用在静态单赋值形式中被命名为不同的变量，从而删除了它们之间不必要的关系。例如，对于程序：

```
int i;
for i=1 to N do
   A[i] = 0;
for i=1 to M do
   s = s + B[i];
```

即使这两个循环计数器的名字都是 `i`，静态单赋值形式也会将其命名为不同的变量，从而不需要使用同一个机器寄存器或中间代码临时变量来保存它们，有可能有利于降低寄存器压力。

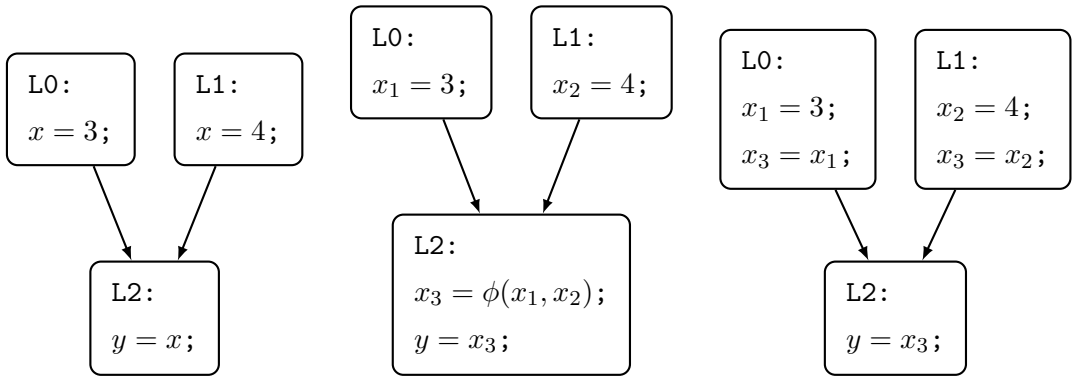
$a = x + y$	$a_1 = x + y$
$b = a - 1$	$b_1 = a_1 - 1$
$a = y + b$	$a_2 = y + b_1$
$b = x \times 4$	$b_2 = x \times 4$
$a = a + b$	$a_3 = a_2 + b_2$
（a）直线型程序	（b）静态单赋值形式的程序

图 5.1: 对原始直线型程序进行重命名后，可以得到静态单赋值形式的程序。

为了将程序转换成对应的静态单赋值形式，编译器可以引入变量版本号的概念，即让每个变量定值都引入一个新版本的变量，而让每个变量使用指向最新的变量定值。在不包含控制流结构的直线型代码中（例如，在一个基本块中），编译器可以让每条指令定值重新命名一个变量。图 1.1 中给出了一个示例程序被转换为静态单赋值形式的结果，每个变量的每个新定值都被修改为定值一个全新的变量名（例如，变量 a 先后被重新命名为 a_1、a_2、a_3 等），该变量的每个使用被修改为使用上一次定值的那个版本。

但是这个简单的重命名策略，无法处理一般的控制流图，特别的，当两条控制流边汇合到一起时，就无法确定变量的“最后”一个版本。例如，在图1.2a 中，基本块 L_0 和 L_1 中分别定值了变量 x 的一个新版本，那么在基本块 L_2 中，我们将无法确定该使用变量 x 的哪个版本，因为对基本块 L_2 来说，基本块 L_0 和 L_1 不存在距离 L_2“更近”的概念。

为解决这个问题，我们可以引入一个虚构的操作，称为 ϕ 函数。在图 1.2b 中，ϕ 函数 $y = \phi(x_1, x_2)$ 接受来自基本块 L_0 中定值的 x_1 和来自基本块 $L1$ 中定值的 x_2 作为参数。但是，和普通的数学函数不同，ϕ 函数沿着控制流的边选取参数值：



(a) 控制流图的边汇合，导致有多个版本的变量定值，同时到达变量的使用点。

(b) 通过引入 ϕ 函数，变量的不同版本可以汇聚成一个版本。

(c) 移除 ϕ 函数后的程序。注意到程序包含对变量 x_3 的重复定值，不再满足静态单赋值形式的要求。

图 5.2: 对有控制流交汇的程序（图1.2a），通过引入 ϕ 函数，可以将其转换为静态单赋值形式（图1.2b）；在进行 ϕ 函数消去后，可以将其重新转换为普通程序（图1.2c）。

即如果控制流沿边 $L_0 \to L_2$ 到达基本块 L_2，则 $\phi(x_1, x_2)$ 选取 x_1 的值并赋值给左侧的变量 x_3，而如果控制流沿边 $L_1 \to L_2$ 到达基本块 L_2，则 $\phi(x_1, x_2)$ 选取变量 x_2 的值并赋值给左侧的变量 x_3。最后，具有唯一定值的变量 x_3 被赋值给变量 y。不难验证，在这个程序里，每个变量都只被定值一次。

大部分实际的机器上，并不包括像 ϕ 函数这样的依赖于控制流的赋值指令，因此，静态单赋值形式的程序无法在这类机器上直接运行。编译器必须先通过消去 ϕ 函数，把静态单赋值形式的程序转换回不含 ϕ 函数的普通程序。简单讲，编译器可以通过将 ϕ 函数对应的赋值，放置到合适的控制流边上完成。对于图1.2b 中的例子，其消去 ϕ 函数后得到程序如图1.2c 所示。ϕ 函数被拆分成两条赋值语句 $x_3 = x_1$ 和 $x_3 = x_2$，被分别放置在基本块 L_0 和 L_1 的末尾。

如果有多个变量在程序的某个基本块汇聚，则可能需要多个 ϕ 函数。图 1.3 中，左侧原始程序中的变量 x、y 都能到

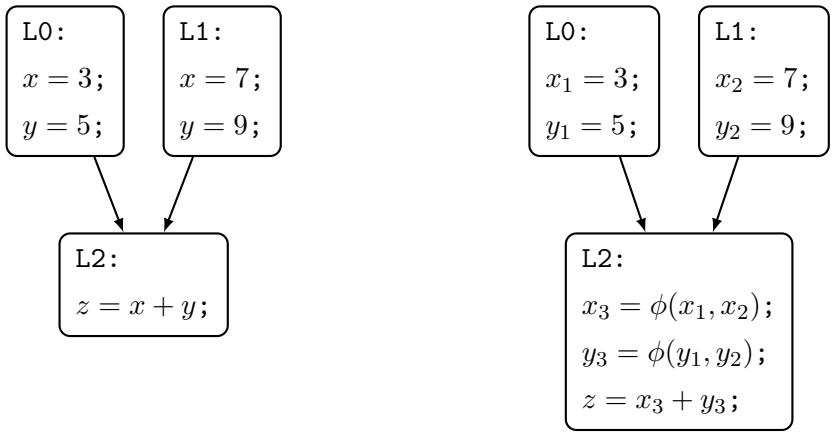


图 5.3: 需要多个 ϕ 函数的程序。

达基本块 L_2。因此，编译器需要对这两个变量都引入 ϕ 函数，从而得到右侧的静态单赋值形式。需要特别注意的是，静态单赋值形式的语义规定同一个基本块（如此处的 L_2）中的 ϕ 函数并行执行，即变量 x_3、y_3 并行同时赋值。对这个例子来说，顺序赋值和并行赋值的结果是相同的。但对于更加复杂的例子，两者可能产生完全不同的结果。假设控制流图某个基本块中包括下面的 ϕ 函数：

$y = \phi(x, x);$
$x = \phi(y, y);$

则上述两条语句将完成变量 x、y 值的交换；而顺序执行会产生错误的结果。

5.2 SSA 的平凡构建

在本小节，我们首先讨论静态单赋值形式的一个平凡构建算法，这个算法包括两个主要步骤：一是在控制流图的汇合点（即有多个前驱的基本块）中插入 ϕ 函数，二是重命名变量，以符合静态单赋值形式名称空间的规则。

首先，在每个汇合点 n 的起始位置，编译器为程序中出

算法 5.1 SSA 的平凡构造

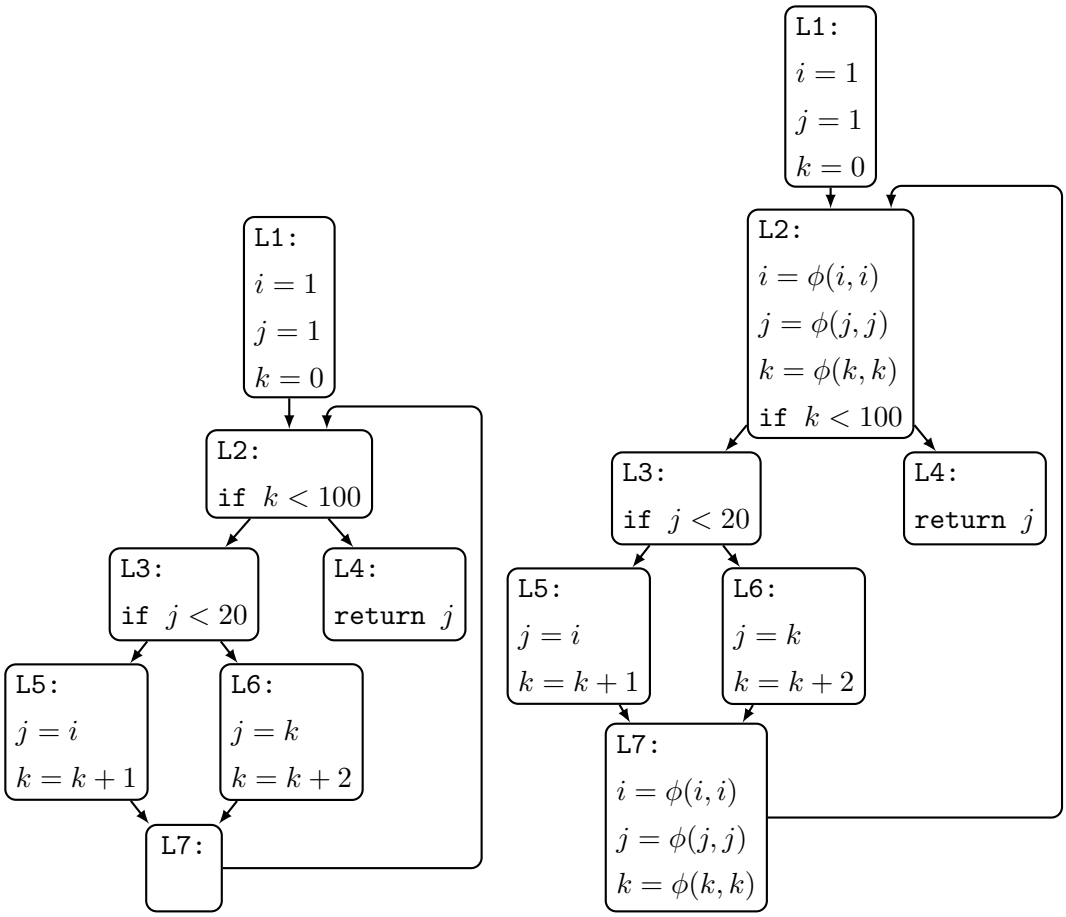
输入： 控制流图形式的程序 G，入口节点为 s_0

输出： 程序 G 的最大化 SSA 形式

```
1: procedure constructMaximalSSA(G)
2:     for each basic block n ∈ G that has multiple predecessors (i.e., joint) do
3:         for each variable x ∈ G do
4:             insert statement x = φ(x, ..., x) into n
5:     ⟨In, Out⟩ =reachDefinition(G)
6:     renameDef(G)
7:     for each basic block b ∈ G do
8:         renameUse(b, Out)
9: procedure renameDef(G)
10:    for each statement x = e ∈ G (including φ) do
11:        write to x' = e (x' is a fresh variable)
12: procedure renameUse(b, Out)
13:    p_1, ..., p_k = predecessors(b)
14:    for each statement x = φ(x, ..., x) ∈ b do
15:        rewrite to x = φ(Out[p_1](x), ..., Out[p_n](x))
16:    for each statement x = e ∈ b do
17:        rewrite to x = In(e)
```

现的每个变量 x 都插入一个平凡的 $x = \phi(x, \ldots, x)$，其中 ϕ 函数的参数数量等于前驱块的数量。其次，在插入 ϕ 函数之后，编译器会对每个变量的定值进行重命名，并将对变量的使用更新为重命名后新的变量名。由于这种形式会包含对所有变量的 ϕ 函数，因此被称为**最大化静态单赋值形式**（maximal SSA form）。

算法1.1描述了最大化静态单赋值形式的构建过程。算法中的函数 constructMaximalSSA 接受程序的控制流图 G 作为输入，将其转换为静态单赋值形式。算法首先在每个作为聚合点出现的基本块 b 上，对程序的所有变量都插入 ϕ 函数。然后，算法进行到达定值分析，计算得到定值集合 In、Out。接

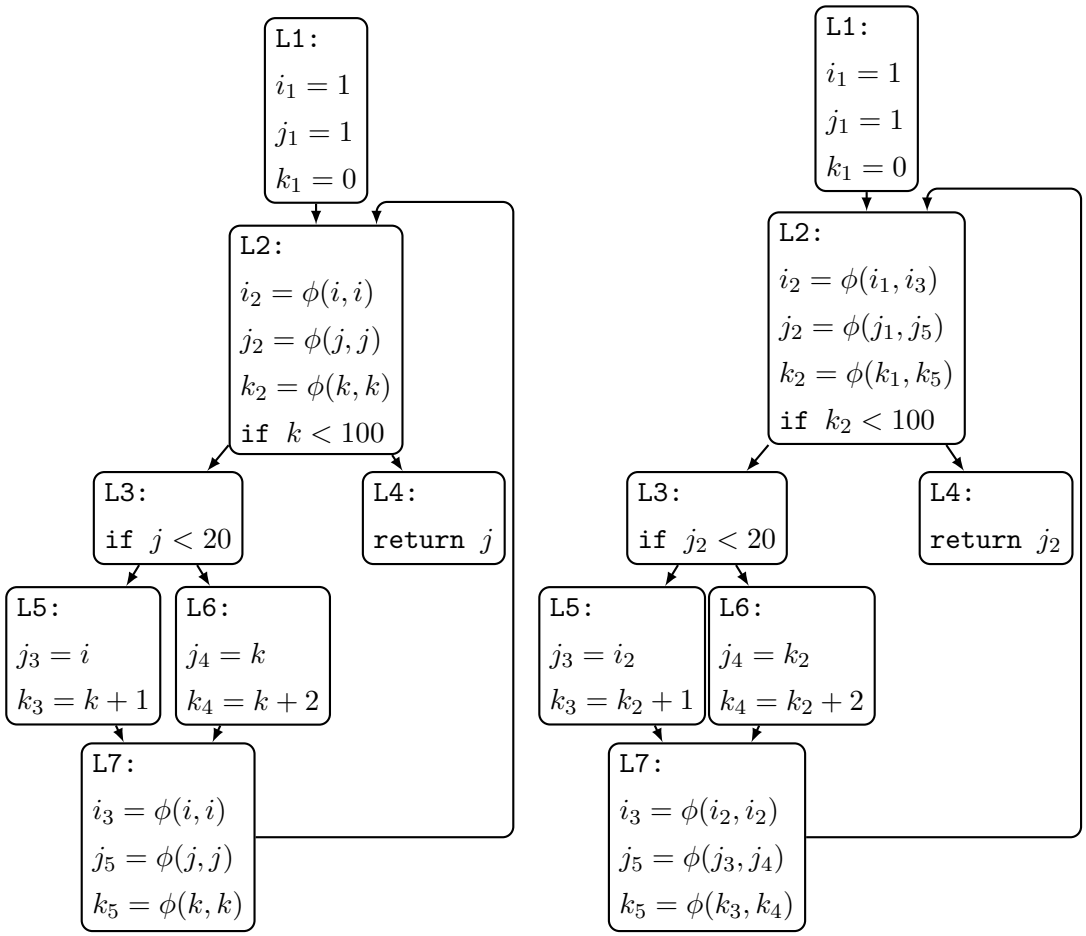


(a) 程序的初始控制流图。 (b) 插入平凡的 ϕ 函数后的控制流图。

图 5.4: 算法1.1在示例控制流图上的执行过程。

着，函数调用 renameDef 函数，对所有的变量定值进行重命名，需要注意的是，由于 ϕ 也涉及到变量定值，因此也需要对 ϕ 函数定值的变量进行重命名。最后，算法调用函数 renameUse，对每个基本块 b 中的变量使用进行系统的重命名。

对变量定值的重命名函数 renameDef 比较简单，我们只需给每个定值一个新的变量名即可。而对于变量使用的重命名 renameUse，我们需要分情况讨论：对于 ϕ 函数中的变量使用（即 ϕ 参数），我们需要从当前基本块 b 的第 i 个前驱 p_i，取得其对应的定值集合 $Out[p_i]$，并利用其重命名变量 x 为 $Out[p_i](x)$。而对于普通语句 $x = e$ 的变量使用，我们将其直接重名为当前语句的 In 所包括的到达定值。



(a) 对变量定值进行重命名。 (b) 对变量使用进行重命名。

图 5.5: 最大化静态单赋值形式的构建。

以图1.4a中给定的程序为例，首先，程序共有两个汇合点 L_2 和 L_7 上，算法在这两个汇合点上，分别为变量 i、j、k 插入 ϕ 函数，得到图1.4b中的控制流图。接着，算法利用到达定值分析，计算每个语句的到达定值。我们在前面章节已经讨论过到达定值的计算过程，具体过程此处从略。例如，对于基本块 L_7，能够到达它的定值集合为（注意，为清晰起见，我们对定值，按照其所在的基本块，进行了分组）

$$L_2 : i = \phi(i, i)$$

$$L_5 : j = i; k = k + 1$$

$$L_6 : j = k; k = k + 2$$

接着，算法对控制流图中变量的定值进行重命名，再对控制流图中语句使用的变量进行重命名，最后得到重命名后的程序如图 1.5a所示，其中每个变量都有唯一的版本。

最后，算法对控制流图中变量的使用进行重命名，把被使用的变量名，替换为能够到达该变量使用的变量定值（且是定值中）。图1.5b给出了对变量使用进行重命名后的程序。仍以基本块 L_7 为例，其中 ϕ 函数参数对变量的使用，都已经被重名为该变量定值中新的变量名。程序最终形式满足静态单赋值形式的要求。

尽管上述算法能够生成正确的静态单赋值形式，但是由于该算法采用激进的策略，在所有汇合点上对每个变量都插入 ϕ 函数，因此可能在程序中引入冗余的 ϕ 函数。例如，在图1.5b的基本块 L_7，ϕ 函数 $i_3 = \phi(i_2, i_2)$ 的两个参数相同，因此等价于赋值 $i_3 = i_2$。多余的 ϕ 函数不仅占用额外的内存，还增加了算法执行的时间。

为了解决这一问题，我们引入了**剪枝静态单赋值形式**（pruned SSA form），基本思想是减少在每个汇合点需要插入 ϕ 函数的变量数量。为此，我们可以对算法 1.1进行修改，在插入 ϕ 函数前，首先对程序执行活跃分析。然后，对于每个汇合点基本块 b，只有当某个变量 x 在基本块 b 的入口处是活跃的（即 $x \in liveIn[b]$），才为该变量 x 插入 ϕ 函数。插入了所需的 ϕ 函数后，接下来的步骤与算法1.1 类似。编译器需要为每个变量定义重新命名，并重新建立使用和定义之间的映射关系。

考虑图1.4a中的程序，利用活跃分析，不难得到变量 $x \notin liveIn[L_7]$，因此，算法不会在基本块 L_7 上放置关于变量 x 的

ϕ 函数。

5.3 SSA 的高效构建

为了高效构建静态单赋值形式，必须减少不必要的 ϕ 函数，那么关键在于搞清楚在哪些汇合点需要为变量插入 ϕ 函数，而这可以通过分析支配关系来实现。

考虑控制流图中某个节点 n，其中包括对变量 x 的一个定值。在 n 支配的节点范围内，x 的定值可以直接传递到这些节点，不需要 ϕ 函数。只有对节点 n“恰好支配不到”的节点，才需要引入 ϕ 函数，把不同的对 x 的定值重命名。我们使用**支配边界**（Dominance Frontier，DF）来刻画节点“恰好支配不到”的位置。通过计算支配边界，编译器就能精确地确定在哪些汇合点需要对哪些变量插入 ϕ 函数。

在接下来几个小节,我们首先讨论支配边界的计算(1.3.1节),再讨论如何在这些支配边界上正确放置 ϕ 函数（1.3.2节），以及如何通过变量重命名（1.3.3节），最终构建合法的静态单赋值形式。

5.3.1 支配边界

给定控制流图 G，我们可给出

定义 5.1 (严格支配节点) *如果节点 n 是 w 的支配节点，并且 $n \neq w$，则我们称 n 是 w 的严格支配节点* (strict dominator)。

例如，在图1.4给出的控制流图中，节点 L_3 是节点 L_7 的严格支配节点。而每个节点（如 L_4）都不是自身的严格支配节点。（回想一下，每个节点都是自身平凡的支配节点。）

为了严格刻画“恰好支配不到”的直觉，我们给出

定义 5.2 (支配边界) 节点 n 的支配边界（dominance frontier）$DF[n]$ 是所有符合下面条件的节点 w 的集合：n 是 w 的某个前驱节点 v 的支配节点，但不是 w 的严格支配节点。

例如，在图1.4给出的控制流图中，节点 L_3 支配节点 L_2 的前驱（即节点 L_3），但并不严格支配节点 L_2（实际上节点 L_3 并不支配 L_2），因此节点 L_2 属于节点 L_3 的支配边界，即 $L_2 \in DF[L_3]$。同理，节点 L_2 支配其自身的一个前驱 L_7，但因为 $L_2 = L_2$，因此 L_2 并不严格支配自身，所以节点 L_2 属于自身的支配边界，即 $L_2 \in DF[L_2]$。

为了计算控制流图中每个节点 n 的支配边界 $DF[n]$，我们定义两个辅助集合：

(1) $DF_{local}[n]$：不以 n 为严格支配节点的 n 的后继；

(2) $DF_{up}[n]$：属于 n 的支配边界，但是不以 n 的直接支配节点作为严格支配节点的节点。

根据 DF_{local} 和 DF_{up}，可以计算每个节点 n 的支配边界 $DF[n]$，即：

$$DF[n] = DF_{local}[n] \cup \bigcup_{c \in children[n]} DF_{up}[c]$$

其中，$children[n]$ 是直接支配节点为 n 的所有节点，亦即支配树上节点 n 的所有直接孩子节点。

为了更容易地计算 $DF_{local}[n]$（使用直接支配节点而不是支配节点），可以利用下面的结论：

$$DF_{local}[n] = \{a \mid a \in Succ[n] 且 Idom[a] \neq n\}。$$

算法 5.2 计算支配边界

输入： 控制流图的节点 n

输出： 每个节点 n 的支配边界 $DF[n]$

```
1: procedure computeDF(n)
2:     S = {}
3:     for each y ∈ Succ[n] do
4:         if Idom[y] ≠ n then
5:             S = S ∪ {y}
6:     for each c ∈ Succ[n] do
7:         computeDF(c)
8:         for each w ∈ DF[c] do
9:             if n ∉ Idom[y] or n == w then
10:                S = S ∪ {w}
11:    DF[n] = S
```

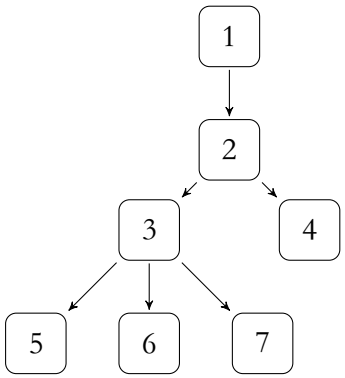


图 5.6: 图1.4中的控制流图对应的支配树。

算法1.2从支配树的根（通常是控制流图的起始节点 s_0）开始，遍历支配树来计算每个节点 n 的支配边界 $DF[n]$。它通过检查 n 的后继来计算 $DF_{local}[n]$，然后将 $DF_{local}[n]$ 与每个子节点 c 的 $DF_{up}[c]$ 进行合并，最终得到完整的支配边界 $DF[n]$。

对于图1.4中的控制流图，其支配树如图1.6所示。进一步，我们可根据算法 1.2，计算得到每个节点的支配边界，如表1.1所示。

此算法的时间复杂度主要取决于控制流图的边数和支配

表 5.1: 支配边界

n	1	2	3	4	5	6	7
$DF[n]$	{}	{2}	{2}	{}	{7}	{7}	{2}

算法 5.3 计算支配边界

输入: 控制流图 G、及其支配节点树 T

输出: 每个节点 n 的支配边界 $DF[n]$

```
1: procedure computeDF(G)
2:     for each node b that has more than one predecessors do
3:         for each predecessor p ∈ Pred[b] do
4:             runner = p
5:             while runner ≠ IDom[b] do
6:                 add b to runner's dominance frontier set
7:                 runner = IDom[runner]
```

边界的总大小。算法在每个节点上只需检查其后继，因而在支配边界较小的情况下，复杂度与图的边数呈线性关系。虽然存在少数极端图结构会导致支配边界较大，但在一般情况下，支配边界的总大小接近线性。因此，在实际中该算法的运行时间几乎总是线性的。

尽管上述算法是计算支配边界的经典算法，但不够简洁直观。我们下面将给出一个更简单的支配边界计算的算法。这个算法基于以下三个核心观察：首先，支配边界中的节点一定是图中的汇合点，而汇合点有多个前驱节点。其次，汇合点的每个前驱节点都会将该汇合点加入自己的支配边界集合，除非它直接支配了该汇合点。最后，前驱节点的支配者们也会在支配边界集合中包含该汇合点，除非它们也支配该汇合点。

基于这些观察，我们在算法1.3中给出了一个基于迭代的计算方法。首先，我们依次考察控制流图中的每个汇合点 b

(即有多于一个的前驱节点)。然后，我们依次检查 b 的每个前驱节点 p，并从 p 开始沿支配树向上遍历（即依次遍历直接支配节点)，直到到达节点 b 的直接支配节点 $IDom[b]$ 为止。在这个过程中，节点 b 被加入遍历经过的每个节点的支配边界集合中。最终，遍历完控制流图中所有度大于等于 2 的节点之后，即可得到所有节点 n 的支配边界 $DF[n]$。

以图1.4控制流图中的节点 L_2 为例，由于 L_2 是一个汇合点，我们逐个考察其前驱节点 L_1 和 L_7（注意，其支配树如图1.6所示)。由于前驱节点 L_1 直接支配节点 L_2，因此略过。而由于前驱节点 L_7 不是 L_2 的直接支配节点，因此将 L_2 加入 L_7 的支配边界集合，并继续考察 L_7 的直接支配节点 L_3。这个过程一直进行，最后也得到表1.1所示的每个节点的支配边界。

5.3.2 放置 ϕ 函数

支配边界给出了在汇合点放置 ϕ 函数的标准，即只要节点 n 包含某个变量 x 的一个定值，则节点 n 的支配边界中的任何节点 z 都需要一个 x 的 ϕ 函数。由于 ϕ 函数本身也是一种定值，因此我们必须迭代地应用支配边界标准，直到再没有节点需要 ϕ 函数为止。

基于这个思想，我们使用算法1.4对一个普通程序 P，通过放置适当的 ϕ 函数，将程序 P 转换为静态单赋值形式。

对于控制流图中的每个节点 n，我们记在节点 n 初始就存在定值的所有变量的集合为 $A_{\text{orig}}[n]$，记需要在节点 n 处插入 ϕ 函数的变量集合为 $A_\phi[n]$。

首先，算法对每个变量 x，构建包含其定值的基本块集合

算法 5.4 ϕ 函数插入算法

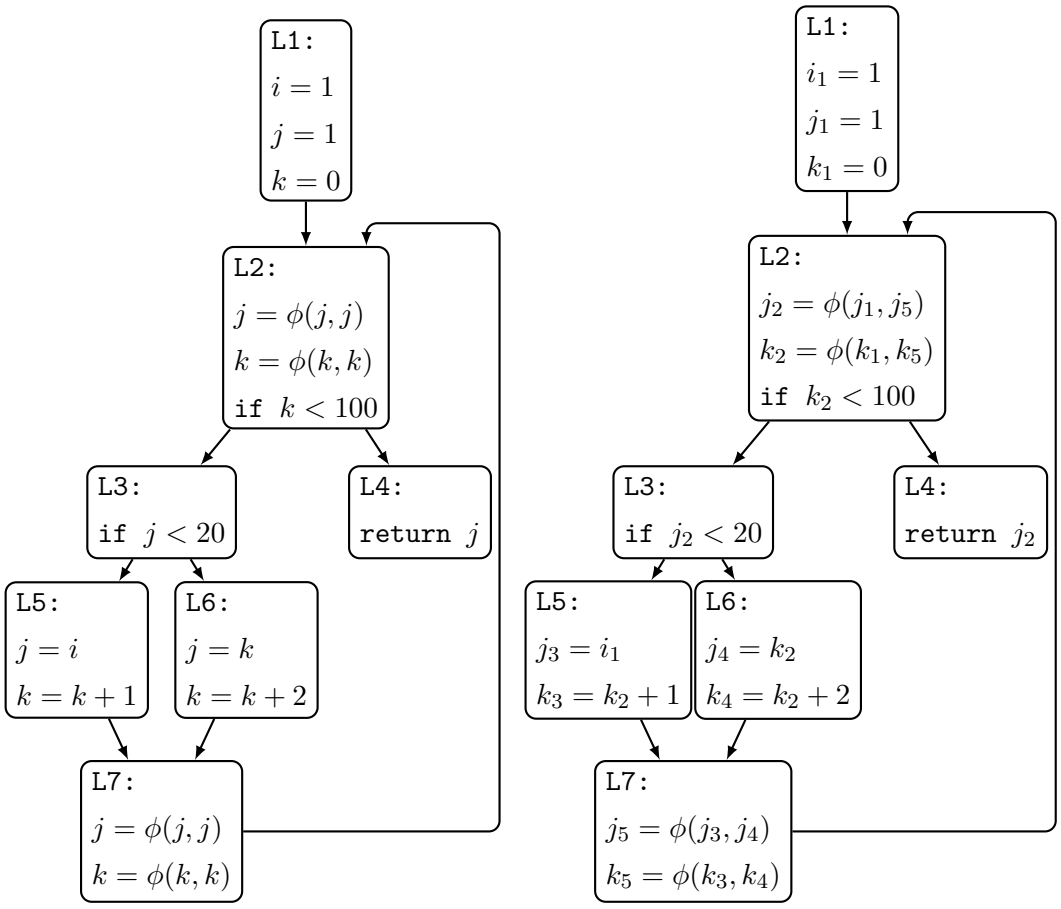
输入：控制流图形式的程序 G

输出：程序 G 的静态单赋值形式

```
1:  procedure Place-φ(G)
2:      for each basic block n ∈ G do
3:          for each variable x ∈ A_orig[n] do
4:              defsites[x] = defsites[x] ∪ {n}
5:      for each variable x do
6:          W = defsites[x]
7:          while isNotEmpty(W) do
8:              n = remove(W)
9:              for each node Y ∈ DF[n] do
10:                 if x ∉ A_φ[Y] then
11:                     insert statement x = φ(x, ..., x) into Y
12:                     A_φ[Y] = A_φ[Y] ∪ {x}
13:                     if x ∉ A_orig[n] then
14:                         W = W ∪ {Y}
```

$defsites[x]$（第 2 到 4 行）。接着，算法考察程序中的每一个变量 x，对包含 x 的定值的基本块集合 W 使用工作表算法，即对于任意一个定值基本块 n，算法遍历 n 的每一个支配边界节点 Y，如果还没有给变量 x 生成过 ϕ 函数的话，则给该基本块 Y 生成一个平凡的 ϕ 函数（即参数名都相同的 ϕ 函数），并把 x 加入到 $A_\phi[Y]$。最后，如果变量 x 是基本块 Y 中新增加的定值变量的话，则也需要将基本块 Y 加入工作表 W。

例如，对图1.4中的初始控制流图来说，变量 i 的初始定制点为 $\{L_1\}$，而变量 j 和 k 的 $defsites[]$ 集合相同，都为 $\{L_1, L_5, L_6\}$。我们以变量 j 为例，应用算法1.4，则其初始工作表 $W = \{L_1, L_5, L_6\}$。首先从工作表 W 中取出基本块 L_1，它的支配边界为 $\emptyset$，不需要处理。然后取出基本块 L_5，它的支配边界为 $\{L_7\}$，因此，需要在 L_7 中为变量 j 放置 ϕ 函数



(a) 放置了 ϕ 函数的控制流图。 (b) 完成变量重命名后的控制流图。

图 5.7: 对控制流图放置 ϕ 函数及变量重命名。

$j = \phi(j, j)$。考虑到基本块 L_7 中原本不包含对 j 的定值，因此要将 L_7 加入工作表 W，即此时 $W = \{L_6, L_7\}$。接着算法取出基本块 L_6，它的支配边界也是 $\{L_7\}$，由于基本块 L_7 上已有关于变量 j 的 ϕ 函数，不需要再处理。然后算法从工作表 W 中取出基本块 L_7，它的支配边界为 $\{L_2\}$，类似地，要在基本块 L_2 中放置变量 j 的 ϕ 函数 $j = \phi(j, j)$，并将 L_2 加入工作表 W，此时 $W = \{L_2\}$。最后，算法从工作表中取出基本块 L_2，它的支配边界为 $\{L_2\}$，由于 L_2 中已经包含关于变量 j 的 ϕ 函数，因此不需要再处理，此时工作表 $W = \emptyset$，关于变量 j 的循环终止。最终，当整个算法运行结束后，加入 ϕ 函数之后的程序的控制流图如图 1.7a所示。

假定初始程序中共有 N 条语句，则程序变量数、语句数以及基本块数都和 N 成正比，则在最坏情况下，算法需要在每个基本块上，对每个变量都插入一个 ϕ 函数，因此，算法插入的 ϕ 函数的数目近似 N^2，即算法的最坏时间复杂度为 $O(N^2)$。但研究表明，在通常情况下，插入的 ϕ 函数数量和 N 成正比。因此在实际中，该算法以近似线性的时间 $O(N)$ 运行。

5.3.3 变量重命名

放置好 ϕ 函数后，我们对变量的定值和使用进行重命名，以满足静态单赋值形式的语法要求。为此，我们遍历支配树，将变量 x 的不同定值（包括对 ϕ 函数的定值），重新命名为不同的版本。在任意基本块内部，程序呈现直线形式，我们可以依次重命名变量 x 的所有定值，然后将 x 的每次使用用距离 x 的最近的定值重新命名。而对于汇合点 n，我们用 n 的第 j 个前驱中变量 x 的定值，来重命名 n 的 ϕ 函数中，对变量 x 的第 j 个使用。

算法1.5给出了对插入 ϕ 函数后的程序进行重命名的步骤。该算法接受流图的基本块 n 作为输入，并重命名其中所有的变量。在算法初始执行时，以程序控制流图的入口节点 s_0（即支配树的根节点）来进行调用。

算法对支配树进行前序遍历，在遍历过程中，算法为每个变量 x 维护两个数据结构：

(1) 变量编号 $Count[x]$：变量 x 下一个要使用的版本号，该值总是单调递增；

算法 5.5 变量重命名算法

输入： 控制流图中的某个基本块 n

输出： 基本块 n 中的变量完成重命名

```
1:  procedure Rename(n)
2:      for each statement S ∈ n do
3:          if S is not a φ function then
4:              for each variable x used in S do
5:                  i =top(Stack[x])
6:                  replace each use x by x_i, in S
7:          for each definition of variable x, in S (including φ) do
8:              Count[x] = Count[x] + 1
9:              i = Count[x]
10:             push i on Stack[x]
11:             replace the definition of x by x_i, in S
12:     for each successor Y of n, in the control flow graph do
13:         suppose n is the j-th predecessor of Y
14:         for each φ function in Y do
15:             suppose the j-th argument of φ function is x
16:             i =top(Stack[x])
17:             replace the j-th argument x by x_i
18:     for each child X of n, in the dominator tree do
19:         Rename(X)
20:     for each statement S in n do
21:         for each definition x in original S do
22:             pop(Stack[x])
```

⑵ 变量“版本”栈 $Stack[x]$：以“记住”变量 x 的最近定值版本。

在算法运行前，编译器给每个变量 x 初始化这两个数据结构：$Count[x] = 0$ 且 $Stack[x] = []$。

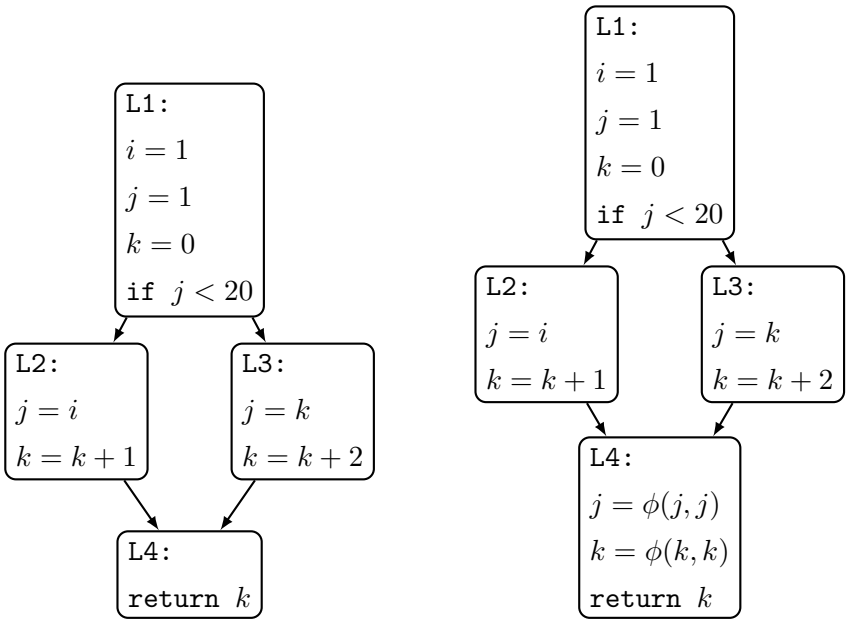
算法首先遍历当前基本块 n 当中的每条语句 S（第 2 到 11 行），如果 S 不是 ϕ 函数的话，则对于语句 S 中使用的每个变量 x，都把该使用替换成 x 的最近的版本号 x_i（注意到 i 是 x 的版本栈中的栈顶元素）；再考察语句 S 定值的每个变量 x（第 7 到 11 行），给这个变量 x 自增一个新的版本号 x_i，并将语句 S 定义的变量 x 改写成 x_i。

接着，算法对节点 n 在控制流图中的每个后继节点 Y 进行遍历（并且假设 Y 是节点 n 的第 j 个后继），对节点 Y 中的每个 ϕ 函数进行遍历，将 ϕ 函数的第 j 个参数 x 替换成 x 的栈顶元素 x_i（第 12 到 17 行）。

算法完成对当前节点 n 的处理后，继续对支配树上其每个孩子节点 X 进行递归遍历（第 18 到 19 行）；最后，算法在结束执行前，需要把当前基本块 n 中所有被定值的变量的版本都清除（第 20 到 22 行）。

算法需要的运行时间和插入 ϕ 函数后的程序的大小成正比，因此在实际中，其运行时间应该和原始程序的大小成近似线性的关系。

图1.7b给出了对变量重命名之后的控制流图。和采用最大化静态单赋值构建算法生成的程序相比，该算法构建的程序一般会包含更少量的 ϕ 函数，更加紧致。例如，和图1.5 中的程序相比，这里图1.7b中没有关于变量 i 的 ϕ 函数。正因如此，基于支配边界的静态单赋值构建算法构建的静态单赋值形式，



(a) 初始控制流图。 (b) 放置 ϕ 函数后的控制流图。

图 5.8: 对控制流图放置 ϕ 函数。

也被称为**最小化静态单赋值形式**（Minimal SSA）。

需要特别注意的是，尽管最小化静态单赋值形式减少了 ϕ 函数的数量，但它未必包含最少个数的 ϕ 函数，即未必生成了最优静态单赋值形式。导致这个问题的根本原因是在最小化静态单赋值形式的构建中，算法没有考虑变量的活跃性，因此可能放置无用的 ϕ 函数。以图1.8a中的控制流图为例，算法为其放置 ϕ 函数后，将得到图1.8b 所示的控制流图。不难发现，变量 j 在基本块 L_4 的入口处并不活跃（即 $j \notin liveIn[L_7]$），因此，在基本块 L_4 上放置对变量 j 的 ϕ 函数实际上是多余的。

为了进一步减少需要放置的 ϕ 函数的数量，我们同样可以采用剪枝的静态单赋值形式中的技术（见1.2小节），即先进行活跃分析，以便确定在每个基本块 b 入口活跃的变量集合 $liveIn[b]$，并仅为 $liveIn[b]$ 中的每个变量 x 放置 ϕ 函数 $x = \phi(x, \ldots, x)$。但是，活跃分析的运行时间复杂度较高，可能给即时编译等对编译性能敏感的场景下带来不必要的开销。

算法 5.6 计算具有全局定值的变量集合

输入： 控制流图 G

输出： 具有全局定值的变量集合

```
1: procedure calculateGlobalDef(G)
2:     GlobalDefs = {}
3:     for each block b ∈ G do
4:         LocalDefs = {}
5:         for each statement x = y ⊕ z ∈ b, topdown do
6:             if y ∉ LocalDefs then
7:                 GlobalDefs = GlobalDefs ∪ y
8:             if z ∉ LocalDefs then
9:                 GlobalDefs = GlobalDefs ∪ z
10:            LocalDefs = LocalDefscupx
```

为了在减少冗余 ϕ 函数的同时避免复杂的活跃分析带来的开销,我们可以采用一个**半剪枝静态单赋值形式**(Semipruned SSA）的折中方案。半剪枝静态单赋值形式的核心思想是确定“具有全局定值的变量”集合，所谓全局定值指的是变量 x 的使用在一个基本块 b 中，但其定值不在基本块 b 内，而是在另一个基本块 c 中。从基本块 b 的视角，变量 x 肯定是入口活跃的，即 $x \in liveIn[b]$。

我们可以记程序具有全局定义的变量集合为 $GlobalDefs$，则算法 1.6给出了计算该集合的步骤。算法扫描给定的控制流图 G 中的每个基本块 b，对基本块 b 中的每个语句 s，如果其变量使用没有在出现在基本块 b 内，即将该变量加入全局集合 $GlobalDefs$。

由于算法需要扫描程序的每条语句，因此其运行时间复杂度与程序的语句数量 N 成正比，即最坏时间复杂度为 $O(N)$。

以图1.8a中的控制流图为例,算法计算得到 $GlobalDefs =$

$\{i, k\}$。可以看到，$GlobalDefs$ 集合实际是每个基本块 b 入口活跃集合的保守近似，即有 $liveIn[b] \subseteq GlobalDefs$。

利用集合 $GlobalDefs$，我们可以改进 ϕ 函数的放置算法，仅当变量 $x \in GlobalDefs$ 时，才放置关于变量 x 的 ϕ 函数（算法1.4第 5 行）。

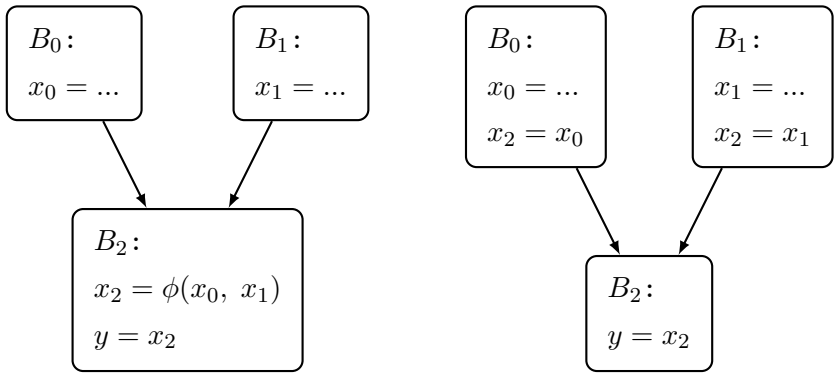
半剪枝静态单赋值形式利用全局定义的变量来指导 ϕ 函数的插入，尽管它不如剪枝静态单赋值形式使用的活跃分析那么精确，但它往往比最小静态单赋值形式生成的 ϕ 函数更少。这使得半剪枝静态单赋值形式在执行效率和分析效果之间取得了有效的平衡。

5.4 SSA 的消去

在现代处理器中，一般没有与 ϕ 函数对应的指令，所以静态单赋值形式的代码并不能直接在处理器上执行。为执行代码，编译器需要在对静态单赋值形式的代码执行完分析和优化后，将其转变成不含 ϕ 函数的形式，这个过程称为**静态单赋值形式的消去**。

编译器可以通过插入一组拷贝指令来消去 ϕ 函数，每一个前驱基本块对应一条拷贝指令。如图1.9所示，为了消去基本块 B_2 中的 ϕ 函数 $x_2 = \phi(x_0, x_1)$，编译器在基本块 B_2 的前驱基本块块 B_0 和 B_1 的结尾分别插入一个拷贝指令 $x_2 = x_0$ 和 $x_2 = x_1$。

但是对一般的静态单赋值形式进行消去，由于图中关键边的存在、以及 ϕ 函数的并行赋值语义，可能分别导致**拷贝丢失问题**（Lost-Copy Problem）或者**交换问题**（Swap Problem）。



(a) SSA 消去前　　(b) SSA 消去后

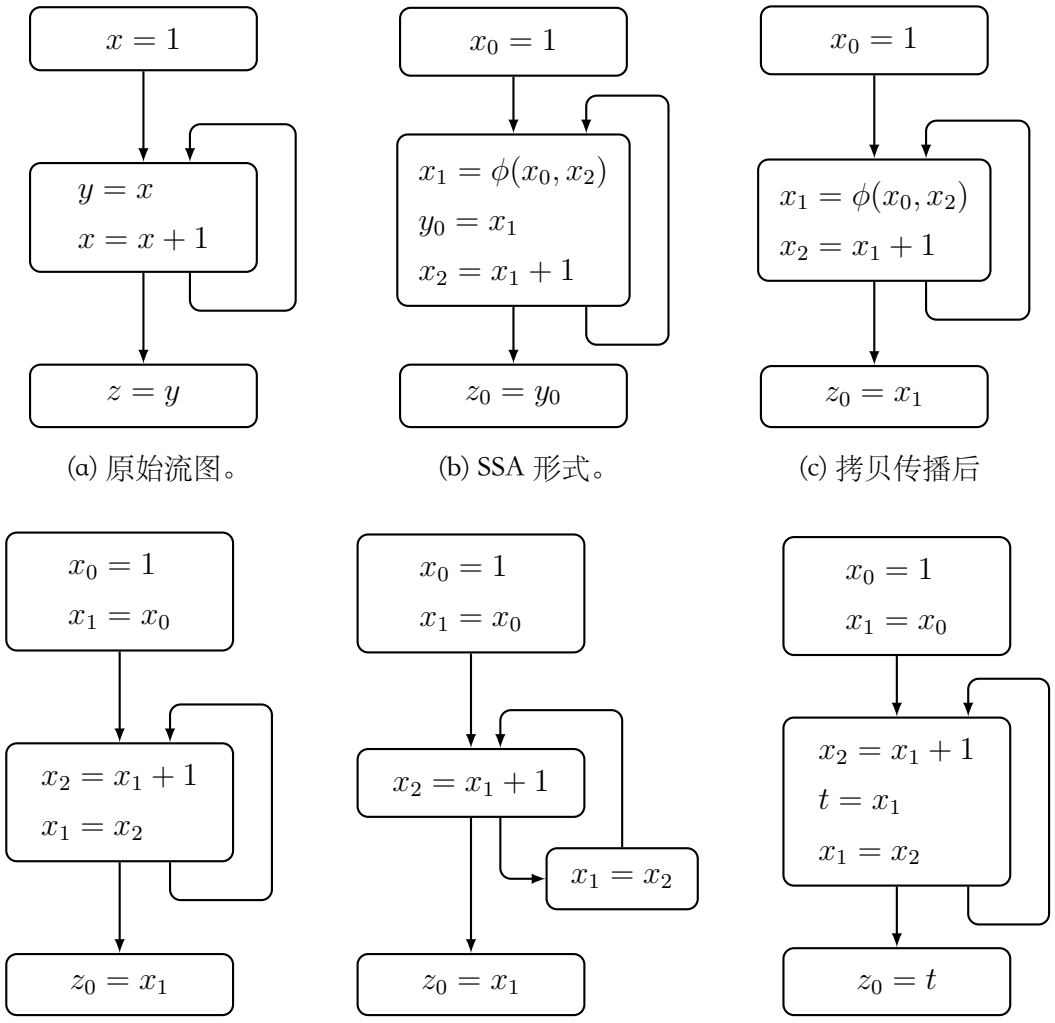
图 5.9: 通过拷贝消去 ϕ 函数。

5.4.1 拷贝丢失问题

但是当流图中存在关键边时（回顾一下，我们曾在??节中讨论过关键边切分问题），编译器不能直接在关键边的前驱基本块中插入指令，否则可能破坏程序的正确性，在静态单赋值形式的消去中这可能导致**拷贝丢失问题**（Lost-Copy Problem）。

图1.10通过具体实例，分析了导致拷贝丢失问题的根因、及其解决措施。图1.10a为原始的控制流图，循环体的回边是一条关键边。图1.10b为流图的静态单赋值形式。在该静态单赋值形式上，对变量 y_0 执行拷贝传播，可将 $z_0 = y_0$ 替换为 $z_0 = x_1$，并且之后 $y_0 = x_1$ 可以作为死代码删除，得到的结果如图1.10c所示。图1.10d给出了消去 ϕ 函数后的结果，将拷贝指令直接插入到前驱块后，会发现对 z_0 的赋值结果比原本的程序多了 1，产生了错误结果。若要避免拷贝丢失问题，可以在插入拷贝指令前对关键边进行切分，得到的正确结果如图1.10e所示。

本质上来讲，拷贝丢失源于变量的生命周期冲突。在图1.10c中，拷贝传播将 x_1 的生命周期由原本的结束位置 $x_2 = x_1 + 1$ 延长到了 $z_0 = x_1$。当对图1.10c 执行 ϕ 消去时，拷贝语



(a) 原始流图。 (b) SSA 形式。 (c) 拷贝传播后

(d) ϕ 消去（错误）。 (e) 切分关键边。 (f) 引入临时变量。

图 5.10: 拷贝丢失问题。

句 $x_1 = x_2$ 的插入并没有顾及 x_1 在插入位置是否活跃，导致了对 x_1 的重新定值，造成了错误的结果。

解决拷贝丢失问题，除了提前进行关键边切分，同样可以在插入拷贝指令时，通过对 ϕ 函数的目标变量的生命周期进行检查来避免拷贝丢失：如果发现目标变量在插入位置活跃，则必须引入一个临时变量来保存目标变量的当前值，并使用该临时变量重写后续对目标变量的使用。例如在图1.10d中插入 $x_1 = x_2$ 时，此时变量 x_1 仍在待插入的位置活跃，所以不能直接插入 $x_1 = x_2$。可以通过引入一个临时变量 t 并将

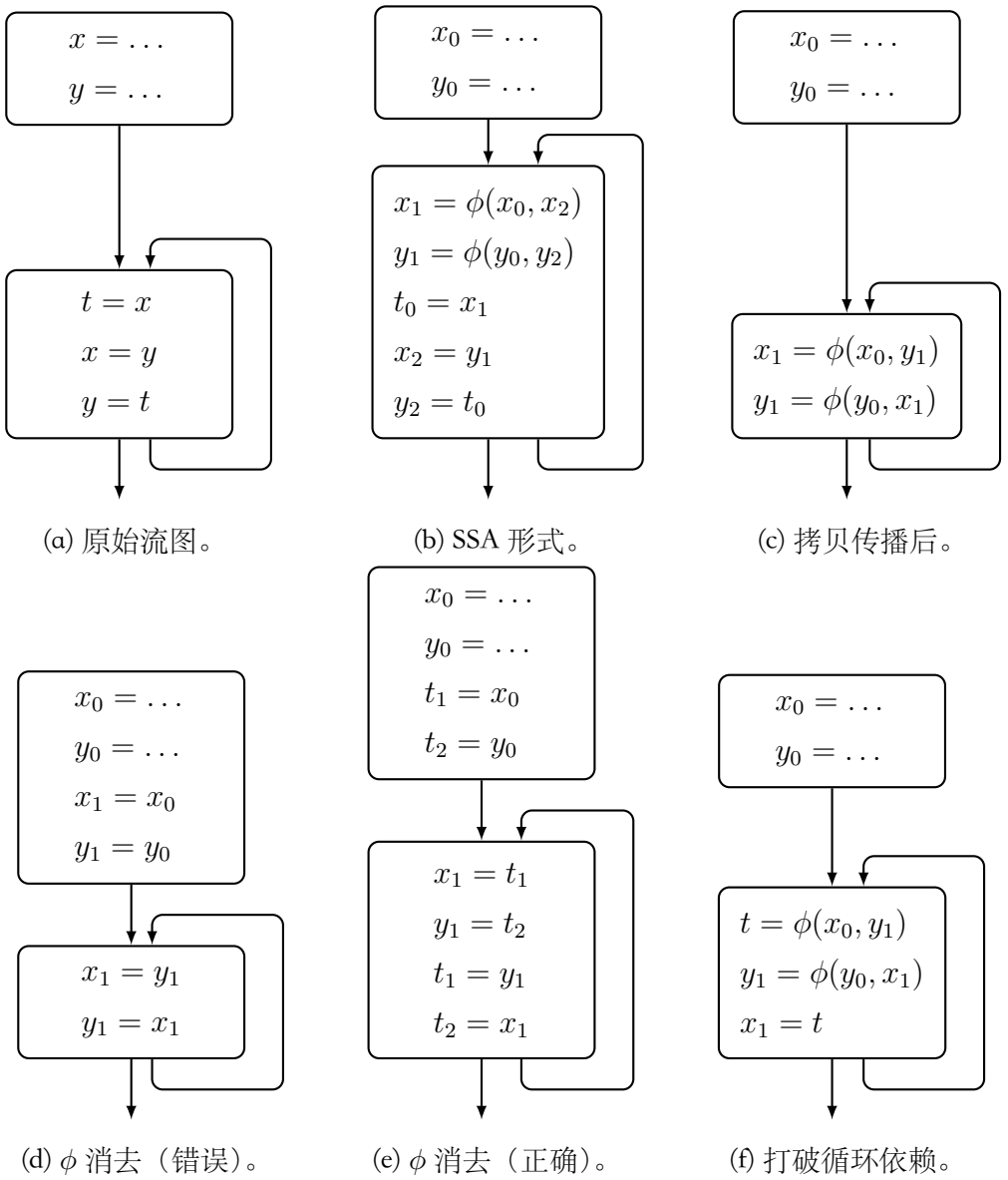
x_1 的当前值拷贝到 t 后再进行插入，并将后续对 x_1 的使用替换为 t，最终结果如图1.10f所示。这种做法避免了关键边切分所带来的额外跳转指令，对于一些性能敏感的场景会比较有用，例如在一个频繁执行循环中，减少跳转指令的数量可以提高指令流水线的效率。但是，这个策略需要对程序进行额外的活跃分析，导致了编译期的时间开销。

5.4.2 交换问题

除了拷贝丢失问题，静态单赋值形式消去还可能出现**交换问题**（Swap Problem），它由 ϕ 函数的并行执行语义所导致。ϕ 函数的并行执行语义指的是：当一个基本块 b 执行时，其开头的 ϕ 函数首先在其他语句执行之前同时*并行*执行，即所有的 ϕ 函数根据程序执行路径，同时读取适当的输入参数，然后同时并行赋值给目标变量。

然而在静态单赋值形式消去时，插入的拷贝操作是顺序执行的，这就可能违反 ϕ 函数的并行语义。换句话说，如果 ϕ 函数顺序执行，如果它们之间存在数据依赖关系，这种依赖关系会导致交换问题。

图1.11通过一个具体实例，展示了交换问题。程序原始的流图如图1.11a 所示，它在一个循环中不断交换变量 x 和 y 的值。图1.11b给出了该程序转变为静态单赋值形式后的程序。图1.11c展示了静态单赋值形式经过拷贝传播和死代码消除后的结果，注意到，循环体中的 ϕ 函数并行执行，同时读取 ϕ 函数的正确参数，并行赋值给变量 x_1 和 y_1。然而当通过插入拷贝指令进行 ϕ 函数消除后，ϕ 函数原本的并行语义被转变成了顺序语义，程序的行为发生了变化。如图1.11d 所示，在顺



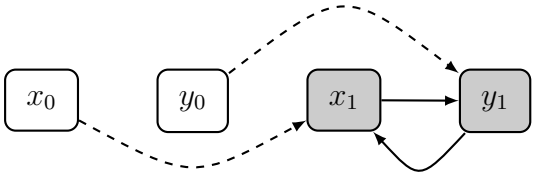
(a) 原始流图。 (b) SSA 形式。 (c) 拷贝传播后。

(d) ϕ 消去（错误）。 (e) ϕ 消去（正确）。 (f) 打破循环依赖。

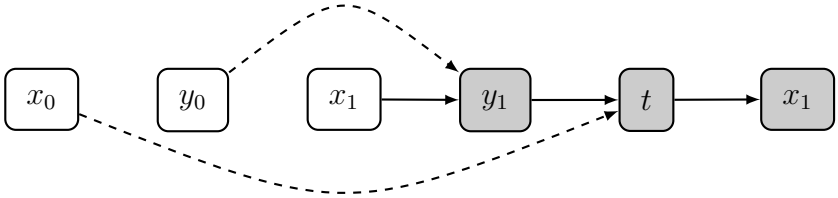
图 5.11: 交换问题。

序执行语义下，变量 x_1 的值始终等于 y_1，产生了错误的执行结果。另外，还需要注意的是，删除关键边，并不能解决交换问题，我们请读者自行在图1.11d 上进行验证。

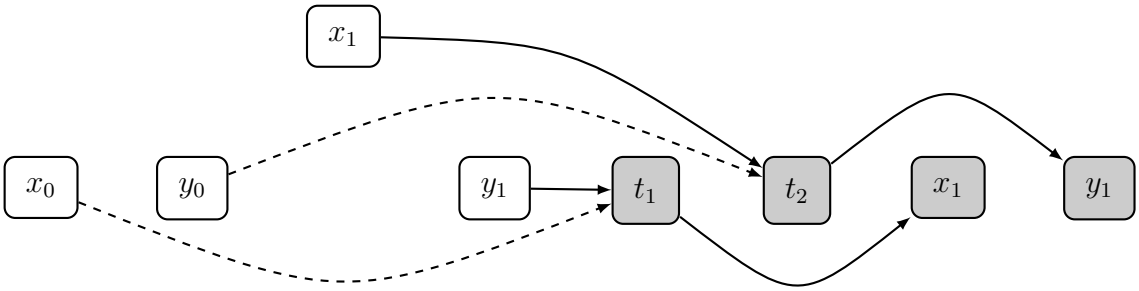
解决交换问题的一种简单的可行方案是使用两阶段拷贝来模拟 ϕ 的并行语义：首先，为每个 ϕ 函数都生成一个全新的临时变量，并将 ϕ 参数该变量中。然后，在所有 ϕ 函数之后，再将临时变量拷贝回原本 ϕ 函数的定值变量。例如，对



(a) ϕ 函数的使用和定值变量构成的依赖关系。



(b) 通过引入新变量 t，形成的满足拓扑排序的依赖关系。



(c) 通过引入两个新变量 t_1、t_2，形成的满足拓扑排序的依赖关系。

图 5.12: ϕ 函数的参数与定值变量间的依赖。

图1.11c中的两个 ϕ 函数，可以通过引入新的变量 t_1、t_2，将其转换成

$$
\begin{aligned}
t_1 &= \phi(x_0, y_1) \\
t_2 &= \phi(y_0, x_1) \\
x_1 &= t_1 \\
y_1 &= t_2
\end{aligned}
$$

最终，我们可得到如图1.11e所示的正确结果。

两阶段拷贝的方法尽管能够解决交换问题，但这种方法激进的为每个 ϕ 函数都额外引入了一个拷贝语句，当 ϕ 数量

较多时并不是很高效。交换问题产生的根因是源自同一基本块中 ϕ 函数之间的循环依赖，即它们的输入依赖其他 ϕ 函数的输出，而其输出又是其他 ϕ 函数的输入。仍以图1.11c中的两个 ϕ 函数为例，我们将其参数和定值变量的赋值依赖关系，可以画成如图1.12a 所示的依赖关系。我们用虚线表示沿着控制流图的一条边进入该节点的时，ϕ 函数取第一个参数（即分别为变量 x_0、y_0），并行赋值给变量 x_1、y_1；用实线表示沿着控制流图的另一条边进入该节点的时，ϕ 函数取第二个参数（即分别为变量 y_1、x_1），并行赋值给变量 x_1、y_1。白色节点表示节点是只读的，而灰色节点表示节点是可读写的。可以看到，由于变量 x_1 和 y_1 存在循环依赖，因此会导致将 ϕ 函数消去时产生错误结果。

事实上，我们可以通过引入临时变量来打破循环依赖，并通过合理安排每个 ϕ 的顺序，确保每个 ϕ 函数的输入不依赖它之前的 ϕ 函数的输出。具体的，如图 1.12b所示，我们为变量 x_1 引入临时变量 t，从而打破了循环依赖。直观上，引入临时变量，等价于将原来变量（如此处的变量 x_1）的写权限取消，并将原来所有对该变量的写操作，都重定向到新引入的临时变量（如此处的变量 t）。最终，算法为该代码生成的线性形式如图1.11f所示，注意这两条 ϕ 函数的相互位置不能交换。

最后，我们要指出的是，图1.11e中给出的 ϕ 函数消去规则，等价于图1.12中给出的拓扑关系。通过引入两个新变量 t_1、t_2，我们彻底消除了变量 x_0、y_0、x_1、y_1 间的依赖关系，从而两个 ϕ 函数可以以任意顺序被消去。

算法 5.7 SSA 形式的消去算法

输入： 程序 G 的 SSA 形式的表示

输出： 程序 G 的不包含 ϕ 函数的控制流图

```
1:  function outSSA(G)
2:      split critical edge for G                          ▷ fix lost-copy problem
3:      for each basic block B ∈ G do                      ▷ fix swap problem for B
4:          for each x_i = φ(...) ∈ B do
5:              rewrite to t_i = φ(...)
6:          add x_i = t_i after all φ functions, 1 ≤ i ≤ n
7:      for each basic block B ∈ G do
8:          for each t_i = φ(x_1, ..., x_n) ∈ B do
9:              insert t_i = x_i at the end of i-th predecessor B_i of B
10:     remove all φ functions in B
```

5.4.3 消去算法

我们在算法1.7给出了静态单赋值形式的消去流程。首先，算法对控制流图切分关键边，以消除潜在的拷贝丢失问题。接着，算法为每个 ϕ 函数都引入新的定值变量，以消除潜在的交换问题。完成这些预处理后，算法将当前基本块 B 中的 ϕ 函数，作为拷贝语句，放置在其前驱块的尾部，并将当前块 B 中的 ϕ 函数移除。

算法只需要处理每个 ϕ 函数，因此具有线性的最坏时间复杂度 $O(N)$，其中 N 是程序中 ϕ 函数的个数。

5.5 优化

正如前文所述，SSA 形式中每个变量只有一个定值，这样的特性使得其能够快速访问程序中的重要数据流。利用这样的优点，我们可以设计出基于 SSA 形式的高效数据流分析和

优化。在本小节中，我们将分别介绍基于 SSA 形式的无用代码删除，稀疏简单常量传播以及稀疏条件常量传播。

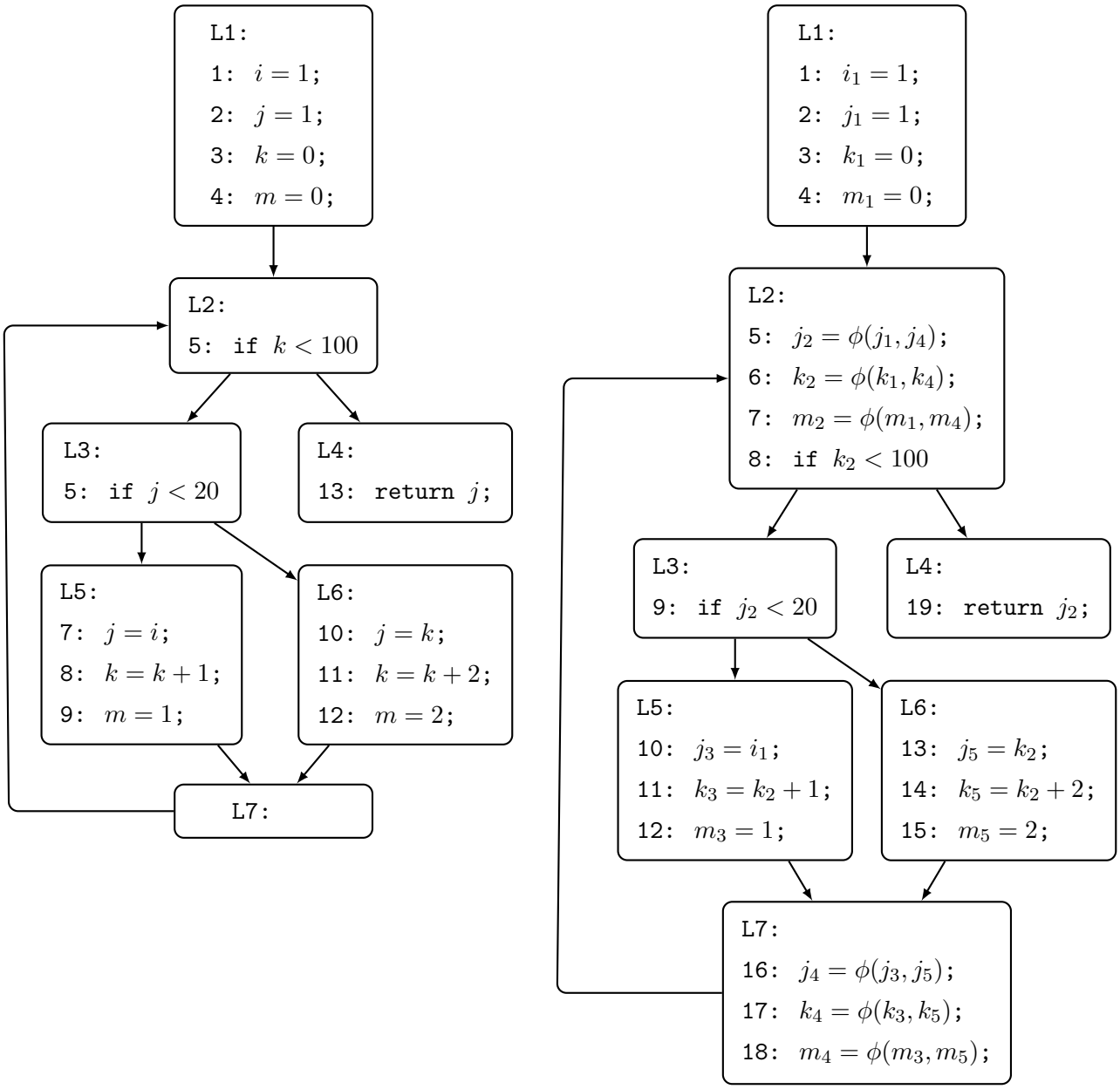


图 5.13: 原始控制流图（左）及其对应 SSA 形式（右）

5.5.1 无用代码删除

回想我们在之前章节（第??节）介绍的基于活跃分析的无用代码删除。一个变量是活跃的指的是存在一条边通向该变

量的一个使用点，并且该路径上不经过该变量的任何定值点。为了判断程序中所有变量的活跃性（活跃分析），我们需要迭代求解数据流方程。

算法 5.8 SSA 形式的无用代码删除算法

输入： SSA 形式的控制流图 G

输出： 经过无用代码删除后的控制流图

```
1:  function UCE-SSA(G)
2:      // usesites[ ] records all variables and their use sites
3:      while usesites[ ] changed do
4:          List = { x | usesites[x] changed }
5:          for each x ∈ List do
6:              if usesites[x] = ∅ then
7:                  // defsite[x] records the def site of x
8:                  s = defsite[x]
9:                  if no side-effort of s then
10:                     remove(G, s)
11:                     for each variable x_i ∈ s do
12:                         remove s from usesites[x_i]
```

在使用 SSA 形式后，无用代码删除的分析将会变得特别快速且简单。一个变量在它的定值点是活跃的当且仅当其使用列表不为空。而对于每个使用点 u，由于同一个变量不可能有其他的定值，并且变量的定值点 d 是其使用点的必经节点，因此一定存在一条从 $d \to u$ 的路径。以图1.13右侧 SSA 形式的控制流图为例，第 11 条语句是变量 k_2 的一个使用点，第 6 条语句是该语句的必经节点且也是变量 k_2 的唯一定值点，因此路径 $6 \to 11$ 便是变量 k_2 的一条定值-使用链。显然，如果程序中存在一个没有使用点的变量 x，且该变量的定值点语句 s 不存在副作用（如函数调用），我们便可以将语句 s 安全的删除。

算法1.8展示了基于迭代的 SSA 形式的无用代码删除。在

删除无用代码 $v = x \oplus y$ 或者 $v = \phi(x, y)$ 时，我们需要将该语句从变量 x 和 y 的使用列表中删除。如果 s 为变量 x 或者 y 的唯一使用点，那么其定值语句也会变为无用代码。另外，我们可以使用上一章介绍的工作表算法（第??节）对该算法进行改进，改进的算法将作为思考题由读者自行完成。

将该算法运用于图1.13右侧 SSA 形式的控制流图，其首先识别到第 7 条语句对变量 m_2 进行了定值但该变量没有使用点，且该语句不存在副作用，因此这条语句会被算法视为无用代码而被删除。被删除的语句是变量 m_1 和 m_4 的唯一使用点，因此需要进一步删除对应的定值点（即第 4 和 18 条语句）。在算法运行结束后，语句 4、7、12、15 和 18 均作为无用代码而删除。

5.5.2 稀疏简单常量传播

在之前的章节中，我们介绍了基于传统控制流图的常量传播，为了和本章节中介绍的常量传播区分开来，我们称之为简单常量传播。传统控制流图中的变量可能存在多个定值点，使得简单常量传播算法依赖于到达定值分析，严重影响了该优化算法的性能。而在 SSA 形式的控制流图中，一个变量只能被定值一次，这样的限制极大的方便了类似常量传播等优化，由此诞生出了*稀疏简单常量传播*（Sparse Simple Constant Propagation，SSCP）。

稀疏简单常量传播算法使用一个平坦格（如图1.14）对程序中所有的变量进行建模。算法首先将每个变量 x 映射到格的 $\perp$ 元素上（$\mathcal{V}[x] = \perp$），这表示我们没有证据表明曾经对变量 x 执行过赋值。如果我们有证据表明变量 x 被赋值为一个常

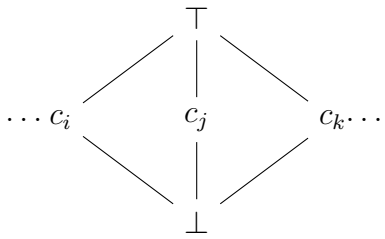


图 5.14: 用于常量传播的平坦格示例图

量 c, 那么算法会将变量 x 映射到格元素 c 上 ($\mathcal{V}[x] = c$)。如果我们有证据表明，变量 x 的值是可变的或者是编译时不可知的，那么算法则会将变量 x 映射到格的 $\top$ 元素上 ($\mathcal{V}[x] = \top$)。

表 5.2: 稀疏常量传播运算在格上的解释

(a) ϕ 运算规则（$c_i \neq c_j$）

ϕ	$\bot$	c_i	c_j	$\top$
$\bot$	$\bot$	c_i	c_j	$\top$
c_i	c_i	c_i	$\top$	$\top$
c_j	c_j	$\top$	c_j	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

(b) $\oplus$ 运算规则

$\oplus$	$\bot$	c_i	c_j	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$
c_i	$\bot$	$c_i \oplus c_i$	$c_i \oplus c_j$	$\top$
c_j	$\bot$	$c_i \oplus c_j$	$c_j \oplus c_j$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

稀疏简单常量传播如算法1.9所示，其包含一个初始化阶段和传播阶段。在初始化阶段，算法遍历 SSA 形式的控制流图中的每个变量，并根据以下规则设置该变量映射到适当的格元素：（1）如果变量 x 的值是一个已知的常量 c，那么设置 $\mathcal{V}[x] = c$；（2）如果变量 x 是某个编译时不可预知的值，那么设置 $\mathcal{V}[x] = \top$；（3）如果变量 x 不满足以上条件，那么设置 $\mathcal{V}[x] = \bot$。

在传播阶段，算法首先记录在上一轮迭代中映射关系 $\mathcal{V}[x]$ 改变的变量 x。对于每一个被记录的变量 x，考察其使用点 u。如果其中某个使用点对变量 v 进行了定值，则按照规则进行

算法 5.9 稀疏简单常量传播

输入： SSA 形式的控制流图 G

输出： 变量到平坦格的映射表

```
1:  function SSCP(G)
2:      for each variable x ∈ G do
3:          initialize V[x] by rules
4:      while V[ ] changed do
5:          List = { x | V[x] changed }
6:          for each x ∈ List do
7:              // usesites[x] records all use sites of x
8:              for each s ∈ usesites[x] do
9:                  if s defines a variable as v = φ(v1, v2, .., vn) then
10:                     V[v] = V[φ(v1, v2, .., vn)]
11:                 else if s defines a variable as v = x ⊕ y then
12:                     V[v] = V[x ⊕ y]
```

以下操作：（1）如果 $\mathcal{V}[v] = \top$，不进行任何操作；（2）如果 $\mathcal{V}[v] \neq \top$，并且使用点 u 具有形式 $v = x \oplus y$，则设置 $\mathcal{V}[v] = \mathcal{V}[x \oplus y]$，其中 $\oplus$ 运算在格上的解释见表1.2b；（3）如果 $\mathcal{V}[v] \neq \top$，并且使用点 u 具有形式 $v = \phi(x_1, .., x_n)$，则设置 $\mathcal{V}[v] = \mathcal{V}[\phi(x_1, .., x_n)]$，其中 ϕ 函数在格上的解释见表1.2a。算法在映射表 $\mathcal{V}$ 不再改变为止。

表 5.3: 稀疏简单常量传播在示例程序上的计算结果

x	i_1	j_1	j_2	j_3	j_4	j_5	k_1	k_2	k_3	k_4	k_5	m_1	m_2	m_3	m_4	m_5
$\mathcal{V}[x]$	1	1	$\top$	1	$\top$	$\top$	0	$\top$	$\top$	$\top$	$\top$	0	$\top$	1	$\top$	2

该算法最终得到控制流图中所有变量到平坦格的映射表。对于映射到常量元素的变量，我们可以替换该变量的所有使用点为对应的常量。仍然以图1.13右侧 SSA 形式的控制流图为例，上述算法在图中的计算结果如表1.3所示。该程序中变量 i_1 最终映射到 1 上，因此我们可以将其所有使用点（第 10 条

语句）处的 i_1 替换为常量 1。同理，变量 m_2 和 m_4 的使用也可以替换为常量。

5.5.3 稀疏条件常量传播

在一些程序中，某些分支可能因为某些原因并不会在运行时执行。但这些分支的存在可能会妨碍程序的常量传播优化。以图1.15为例，该程序包含一个调试变量 $debug$，该变量只在调试时设置为 $true$，而在其他情况下默认为 $false$。假定程序运行在非调试模式，那么程序在运行中只会执行路径 $L1 \rightarrow L3 \rightarrow L4$，并且在这种情况下，变量 i_2 与 i_3 均为常量 2。如果我们使用稀疏简单常量传播对该程序进行优化，那么由于算法并不知道 $L2$ 块不会被执行，因此其会认为变量 i_2 在运行时会出现两个值而拒绝对该变量的常量优化。

我们不希望程序中的调试从句中的语句妨碍到有用的优化，因此需要对程序中路径的执行情况进行分析：一条路径将会被执行当且仅当有证据表明该路径会被执行；而一个变量被认为是一个常量当且仅当有证据表明该变量不是常量。依据这种思路改进的稀疏简单常量传播被称为**稀疏条件常量传播**（Sparse Conditional Constant Propagation，SCCP）。

稀疏条件常量传播使用与稀疏简单常量传播相同的平坦格（图1.14）对程序中的变量进行建模，相应的格和变量到格元素映射的含义均相同，因此不再赘述。另外，除了程序的变量，我们还需要跟踪程序中每条路径的执行性：对于程序中的从基本块 B_x 指向基本块 B_y 的路径 (B_x, B_y)，$\varepsilon[(B_x, B_y)] = false$ 表示我们没有证据表明路径 (B_x, B_y) 曾经被执行，$\varepsilon[(B_x, B_y)] = true$ 则表示存在证据表明该路径会被执行。

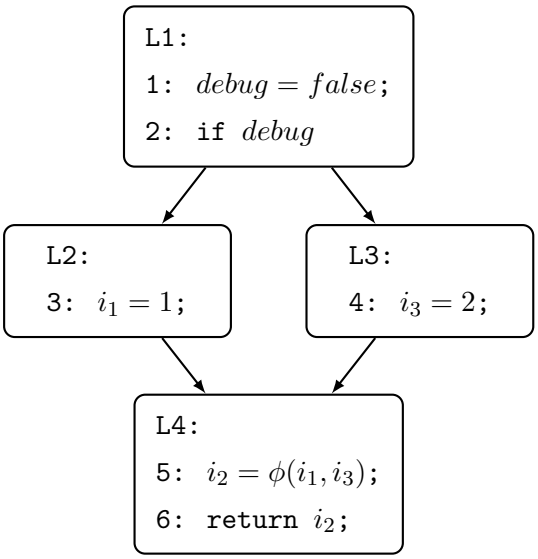


图 5.15: 带调试变量的 SSA 形式控制流图

算法 5.10 稀疏条件常量传播

输入： SSA 形式的控制流图 G

输出： 变量到平坦格的映射表及基本块可执行性

```
1:  function SCCP(G)
2:      for each variable x ∈ G do
3:          V[x] = ⊥
4:      for each edge (B_x, B_y) ∈ G do
5:          ε[(B_x, B_y)] = false
6:      ε[(−, ENTRY)] = true
7:      while ε[ ] or V[ ] changed do
8:          Elist = { (B_x, B_y) | ε[(B_x, B_y)] changed }
9:          Vlist = { x | V[x] changed }
10:         for each (B_x, B_y) ∈ Elist do
11:             for each statement s in block B_y do
12:                 VisitStatement(s)
13:         for each x ∈ Vlist do
14:             // usesites[x] records all use sites of x
15:             for each s ∈ usesites[x] do
16:                 VisitStatement(s)
```

稀疏条件常量传播如算法1.10所示，其包括一个初始化阶段和传播阶段。在初始化阶段，算法（1）遍历程序中所有的

变量 x，设置 $\mathcal{V}[x]$ 为 $\perp$ 元素；（2）遍历程序中所有的路径 (B_x, B_y)，设置 $\varepsilon[(B_x, B_y)]$ 为 $false$。在初始化阶段的最后，我们将程序的入口路径设置为可执行的。

在传播阶段，算法首先记录在上一轮迭代中映射关系 $\mathcal{V}[x]$ 改变的变量 x 以及执行情况 $\varepsilon[(B_x, B_y)]$ 改变的路径 (B_x, B_y)。对于每一个被记录的路径 (B_x, B_y)，需要分析基本块 B_y 中的每一条语句；对于每一个被记录的变量 x，需要查找并分析该变量的所有使用点 s。

算法 5.11 稀疏条件常量传播语句分析算法

```
1:  function VisitStatement(s)
2:      if s has formal as v = φ(v_1, v_2, .., v_n) then
3:          for each incoming edge (B_x, B_y) do
4:              // suppose (B_x, B_y) is the i^th incoming edge
5:              if ε[(B_x, B_y)] = true then
6:                  val_i = V[v_i]
7:              else
8:                  val_i = ⊥
9:          V[v] = V[φ(val_1, val_2, .., val_n)]
10:     else if s defines a variable as v = x ⊕ y then
11:         V[v] = V[x ⊕ y]
12:     else if s at block B_x controls a branch if(v, B_y, B_z) then
13:         if V[v] = ⊤ then
14:             ε[(B_x, B_y)] = ε[(B_x, B_z)] = true
15:         else if V[v] = c then
16:             set ε[(B_x, B_y)] = true or ε[(B_x, B_z)] = true by c
17:     else if s at block B_x has formal as jmp B_y then
18:         ε[(B_x, B_y)] = true
```

稀疏条件常量传播语句分析如算法1.11所示。该算法对四种不同类型的语句进行分析和操作：（1）对于形如 $v = \phi(v_1, v_2, .., v_n)$ 的语句 s，假定变量 $\{v_1, v_2, .., v_n\}$ 分别由入边 $\{(x_1, y), (x_2, y), .., (x_n, y)\}$ 提供。如果边 (x_i, y) 是可执行的，那

么设置该值 $val_i = \mathcal{V}[v]$；否则边 (x_i, y) 不是可执行的，那么设置该值为 $val_i = \perp$。根据路径的执行性，我们可以计算出变量 v 到格上新的映射 $\mathcal{V}[v] = \mathcal{V}[\phi(val_1, .., val_n)]$，其中 ϕ 函数在格上的解释与稀疏简单常量传播算法相同（见表1.2a）；（2）对于形如 $v = x \oplus y$ 的语句 s，计算变量 v 到格上新的映射 $\mathcal{V}[v] = \mathcal{V}[x \oplus y]$，其中 $\oplus$ 在格上的解释见表1.2b；（3）对于基本块 B_x 中的条件控制语句 $if(v, B_y, B_z)$，如果 $\mathcal{V}[v] = \top$，说明控制语句左右两边的路径都会被执行，因此需要将设置 $\varepsilon[(B_x, B_y)]$ 和 $\varepsilon[(B_x, B_z)]$ 均为 $true$；如果 $\mathcal{V}[v] = c$，说明目前只有一条路径会被执行，此时需要在 c 取值为 $true$ 的情况下设置 $\varepsilon[(B_x, B_y)]$ 为 $true$，在 c 取值为 $false$ 的情况下设置 $\varepsilon[(B_x, B_z)]$ 为 $true$；（4）对于基本块 B_x 中的无条件跳转语句 $jmp\ B_y$，在 B_x 执行时 B_y 一定会被执行，因此直接设置 $\varepsilon[(B_x, B_y)]$ 为 $true$。

将上述的稀疏条件常量传播运用于图1.13右侧 SSA 形式的控制流图，得到的程序基本块执行情况及各变量定值情况如表1.4所示。

表 5.4: 稀疏条件常量传播在示例程序上的计算结果

B	L1	L2	L3	L4	L5	L6	L7
$\varepsilon[B]$	true	true	true	true	true	false	true

x	i_1	j_1	j_2	j_3	j_4	j_5	k_1	k_2	k_3	k_4	k_5	m_1	m_2	m_3	m_4	m_5
$\mathcal{V}[x]$	1	1	1	1	1	$\perp$	0	$\top$	$\top$	$\top$	$\perp$	0	$\top$	1	1	$\perp$

5.6 函数式中间表示

在函数式语言的执行过程中变量会被绑定到值，并且变量一旦被绑定便不可被修改。函数式语言的这种性质对编译器进行等式推理非常有用：通常编译器通过重写程序使其变为一个高效的版本，当重写时，编译器不需要担心某个变量在另一个位置是否具有不同的值，这使得重写非常容易表达和实现。

前面的章节中，我们介绍了静态单赋值形式的中间表示，其单赋值的性质使得程序分析和优化变的非常简单，而函数式语言同样具有这种单赋值的性质。在过程式语言的编译器中，通常使用静态单赋值形式作为中间表示，使用基本块表示计算并且使用有向边表示控制流关系。而在函数式语言中，则通常使用函数式的中间表示，使用函数表示计算并使用函数调用表示控制流关系。尽管二者的表示方式不同，但它们本质上是等价的，并且可以互相转换。换句话说，静态单赋值形式本质上是一种函数式语言的图表示形式。

在介绍二者如何转换之前，我们先给出一个简单的函数式中间表示的上下文无关文法

$$
\begin{aligned}
fundef &::= \textbf{function}\ v(v*) = letexp \\
letexp &::= \textbf{let}\ binding*\ \textbf{in}\ jumpexp \\
jumpexp &::= \textbf{if}\ atom\ \textbf{then}\ callexp\ \textbf{else}\ callexp \\
&\quad\ \ callexp \\
&\quad\ \ \textbf{return}\ atom \\
callexp &::= v(atom*) \\
binding &::= fundef
\end{aligned}
$$

$$
\begin{aligned}
&v = atom \\
&v = v(atom*) \\
&v = atom\ binop\ atom \\
atom ::= &\ literal \mid v \\
binop ::= &\ + \mid - \mid * \mid / \mid < \mid == \mid \ldots
\end{aligned}
$$

函数式中间表示和函数式语言非常类似，它们具有相同的表达式类型和语法结构，只不过在函数式中间表示中类型都是原子的（如字面量 $literal$ 和变量 v）。函数式中间表示由函数定义 $fundef$ 开始，它由函数名称、参数列表以及一个`let`表达式构成。`let`表达式是函数式语言相比于其他语言比较特殊的地方，一个变量可以在 $binding*$ 中被绑定到一个值且只可被绑定一次，该值可以是表达式的结果也可以是一个新函数的定义，并且每个变量绑定都有一个作用域，使得该变量只能在相应的作用域内被使用。例如对于`let`表达式

$$
\begin{aligned}
&\textbf{let}\ v = 10, x = v, \\
&\quad \textbf{function}\ f_1() = letexp_1, \\
&\quad \textbf{function}\ f_2() = letexp_2, \\
&\textbf{in}\ jumpexp
\end{aligned}
$$

变量 v 被绑定到 10 并且只能在其后的绑定以及 $jumpexp$ 中使用，而函数绑定 f_1 和 f_2 则可以在 $letexp_1$、$letexp_2$ 和 $jumpexp$ 中使用。

函数式中间表示的作用域规则使得某些优化非常容易实现，一个最典型的例子是函数内联优化：对于函数定义 $f(x) = letexp$ 和该函数的使用 $f(y)$，我们可以将 $letexp$ 中对 x 的使

算法 5.12 SSA 形式到函数式中间表示的转换

输入：程序的静态单赋值形式的中间表示

输出：程序的函数式中间表示

```
1: function Translate(node)
2:     let a_1, ..., a_k be the target of φ-functions in node
3:     let S = all assignment statements in node except φ-functions
4:     switch jump statements of node do
5:         case jmp L_i
6:             suppose the function f_i is translated from L_i
7:                 and x_i1, ..., x_im is the input of φ-functions in L_i from node
8:             E = "f_i(x_i1, ..., x_im)"
9:         case if(cond, L_i, L_j)
10:            suppose f_i and f_j are translated from L_i and L_j, x_i1, ..., x_im and
11:                x_j1, ..., x_jn are corresponding input of their φ-functions
12:            E = "if cond then f_i(x_i1, ..., x_im) else f_j(x_j1, ..., x_jn)"
13:        case return x
14:            E = "return x"
15:    let C be the children of node in the dominator tree
16:    for each p_i(i = 1, ..., n) ∈ C do
17:        F_i = Translate(p_i)
18:    F = F_1, F_2, ..., F_n
19:    return "function(a_1, ..., a_k) = let S F in E"
```

用全部替换为 y，来替换函数调用 $f(y)$。由于在函数式中间表示中 y 的类型是原子的而不是其他非原子类型的表达式，这使得替换非常容易实现。

算法1.12说明了将 SSA 转换成函数式中间表示的流程。算法为流图每一个基本块都生成一个函数，并将基本块中 ϕ 函数的输出作为函数的形参（若无 ϕ 函数则无形参），将其他基本块到当前块的跳转转换成函数调用，调用函数的参数则为 ϕ 函数相应的输入参数。此外，如果基本块 f 支配 g，那么 g 对应的函数可以放到 f 对应函数的`let`表达式的绑定中。

算法采用自顶向下的方式来对支配树进行遍历。算法 2-14 行处理当前基本块 $node$，首先将 ϕ 函数的输出作为函数的参数，而除 ϕ 函数外的赋值语句作为函数的变量绑定，由于 SSA 中的赋值语句和函数式中间表示具有相同的形式，无需进行特殊处理。接下来，算法 4-14 行处理 $node$ 的跳转语句，根据具体的情况将跳转语句转换成函数调用，例如对于jmp语句，若跳转目标 L_i 对应的函数为 f_i，则生成一条到 f_i 的函数调用语句。算法 15-18 行递归处理当前 $node$ 在支配树中的孩子节点（直接支配节点）。最后，算法利用得到的参数列表 $a_1, \ldots, a_k$、变量绑定 S 、函数绑定 F 以及跳转语句转换的结果 E 为当前基本块组合成一个函数并返回。

如下所示为图??中流图转换成的函数式中间表示：

```
function f_1() = let
   i1 = 1, j1 = 1, k1 = 0,
   function f_2(j2, k2) = let
      function f_3() = let
         function f_5() = let
            j3 = i1, k3 = k2 + 1
         in f_7(j3, k3),
         function f_6() = let
            j5 = k2, k5 = k2 + 2
         in f_7(j5, k5),
         function f_7(j4, k4) = let in f_2(j4, k4)
      in
         if j2 < 20 then f_5() else f_6(),
      function f_4() = let in return j2,
   in
      if k2 < 100 then f_3() else f_4(),
in
   f_2(j1, k1)
```

函数式中间表示同样可以转换成等价的 SSA 形式，并且相对容易一些，算法 1.13说明了转换的流程，转换可以从函数

算法 5.13 函数式中间表示到 SSA 形式的转换

输入： 程序的函数式中间表示的最外层函数定义 F

输出： 程序的 SSA 形式的控制流图

```
1:  function Translate(F)
2:      B = createBasicBlock()
3:      for each argument x_i of F do
4:          insert x_i = φ(a_1, ..., a_k) into B, where a_i is the parameter of calling F
5:      for each variable binding b of F do
6:          if b is funcdef then
7:              Translate(b)
8:          else
9:              insert b at the end of B
10:     switch jumpexp of F do
11:         case callexp
12:             suppose the called function in callexp is translated from basic block B'
13:             insert jmp B' at the end of B
14:         case if atom then callexp_1 else callexp_2
15:             suppose the called function in callexp_1 and callexp_2 are translated
16:                 from B_1 and B_2
17:             insert if(cond, B_1, B_2) at the end of B
18:         case return atom
19:             insert return atom at the end of B
20:     emit(B)
```

式中间表示的最外层函数递归进行。算法首先为函数创建一个基本块，然后 3-4 行根据函数的参数，在当前基本块中插入 ϕ 函数，将函数的形参作为 ϕ 函数的输出，调用该函数使用的实参作为 ϕ 函数的输入。5-9 行遍历 F 中的所有绑定，若绑定的是一个函数，则递归处理该函数，否则将该绑定作为赋值语句插入到基本块末尾。算法 10-19 行处理函数中的 $jumpexp$ 部分，根据具体的情况转换成基本块的跳转指令。最后算法使用 `emit`函数发射基本块。

5.7 本章小结

编译器使用各种中间表示，来更好地进行程序分析和程序优化；同时，中间表示还可以使得编译器更好地进行前后端解耦。本章讨论了编译器中常用的中间表示，包括在基本块内部常用的线性表示、语法树和抽象语法树等常用的树状表示，以及控制流图和依赖图等图状表示；讨论了编译器将高层表示编译成中间表示的方法。

本章还重点讨论了在现代编译器中常用的静态单赋值形式，尽管构造静态单赋值形式有一定代价，但静态单赋值形式通过保证每个变量都只有一个静态赋值，大大简化了很多程序分析和优化，已经成为现代优化编译器的主流选择。

5.8 深入阅读

静态单赋值形式是由 Wegman、Zadeck、Alpern 和 Rosen [1, 8] 提出的，旨在高效地解决数据流分析问题，包括全局值编号、变量同余性分析、激进的死代码删除，以及带有条件分支的常量传播 [9]。Cytron 等人 [5] 则描述了一种基于支配边界的方法来高效计算静态单赋值形式和控制依赖图，之后，Cooper 等人 [4] 提出了一种更符合直觉的支配边界计算算法。Briggs 等人 [3] 详细描述了静态单赋值形式中的变量重命名算法以及从静态单赋值形式重新生成可执行代码的过程，并指出在恢复代码的过程中，关键边会带来复杂性问题。

Wolfe[10] 对静态单赋值形式（他称之为分解的 use-def 链）进一步拓展，提出了一些优化算法，其中包括循环优化中的归纳变量分析。此外，Reif 和 Lewis[7, 6] 提出了稀疏简单常量

传播算法（SSCP），随后 Wegman 和 Zadeck[9] 对该算法进行了重定义，使其能够应用于 SSA 形式。静态单赋值形式的函数式中间表示在许多编译器相关教材 [2] 中均有详细说明。

5.9 思考题

1. 分别使用算法??和??，计算出图1.16中各节点的支配边界。

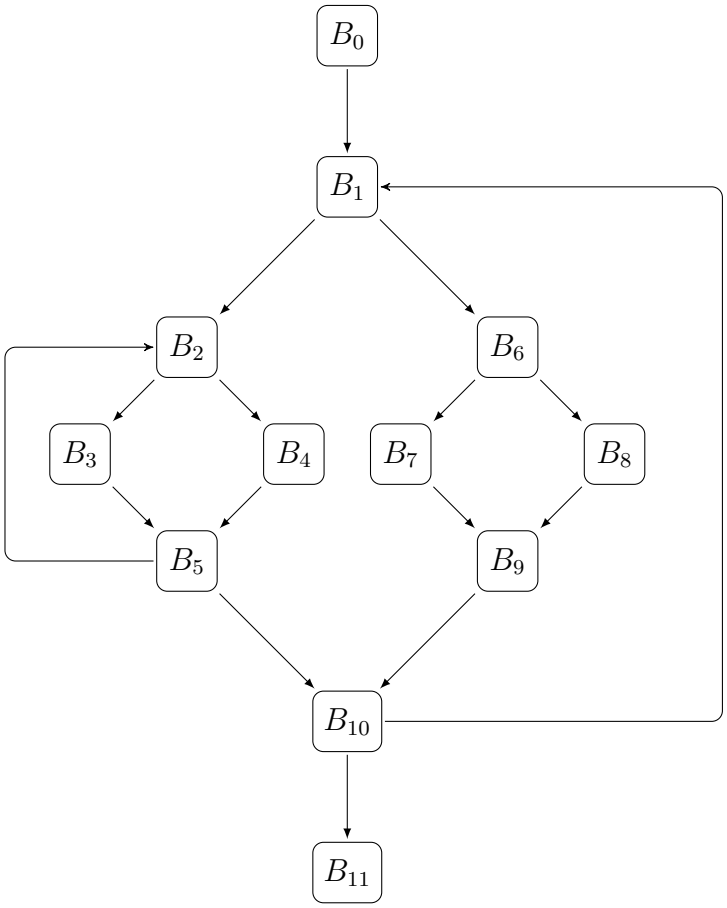


图 5.16: 示例控制流图

2. 考察如下图所示的代码片段。绘制其控制流图，并给出其对应的静态单赋值形式。

```
x =...
y =...
a=y+2
b = 0
```

```
while(x<a)
  if(y<x)
    x = y+1
    y = b*2
  else
    x =y+2
    y= a/2
  w = x+2
  z = y*a
  y = y+1
```

3. 图1.17为一个计算前 n 项斐波那契数列中偶数元素之和的程序，请使用1.5小节介绍的无用代码消除、稀疏简单常量传播以及稀疏条件常量传播三种算法，对该程序进行分析并给出结果。

4. 正如前文所述，本章介绍的三种优化算法均基于迭代计算，在实际实现中可以使用第??节介绍的工作表算法进行改进。请尝试写出基于工作表的无用代码消除、稀疏常量传播和稀疏条件常量传播算法。

5. 理解图1.17程序的执行流程并给出执行 SSA 消去后的结果。

5. 给出图1.17的函数式中间表示。观察你得到的结果，它一共包含多少个函数，有哪些函数是不必要的或者可以简化的？结合你学习过的函数式编程语言，改进1.6节中的函数式中间表示的文法，使其更加简洁高效，并给出一个 SSA 形式到它的转换算法。

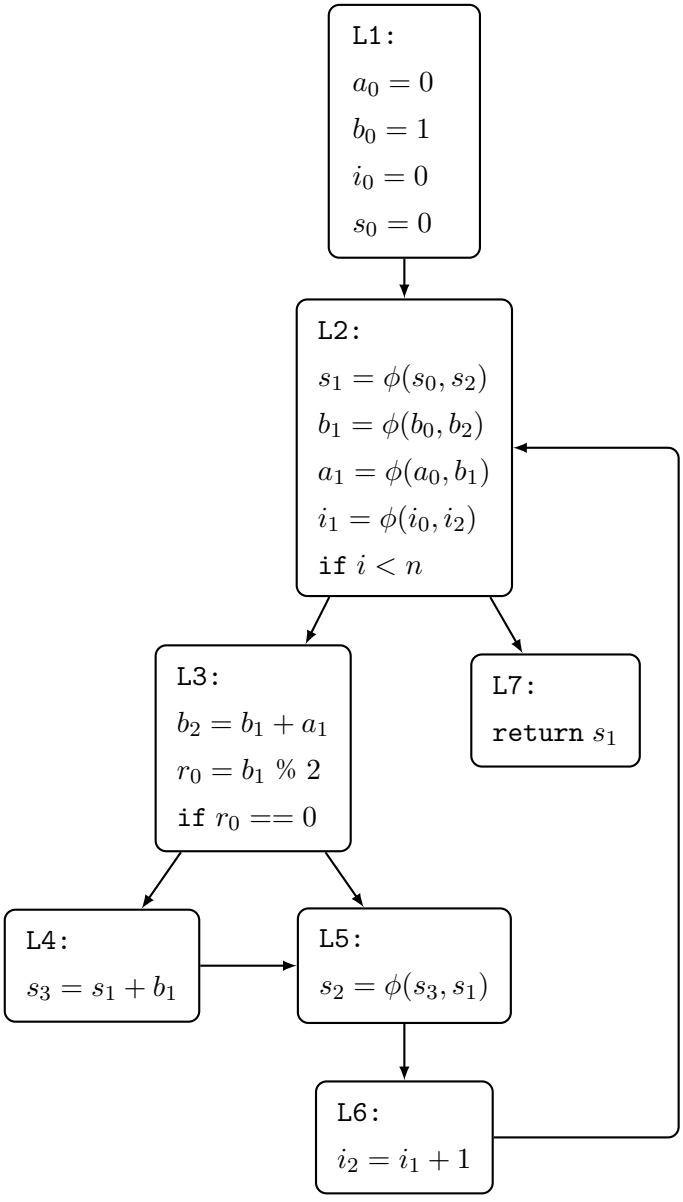


图 5.17: 思考题 SSA 形式控制流图

参考文献

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '88, pages 1 – 11. ACM Press.

[2] Andrew W Appel. Modern Compiler Implementation in ML. Cambridge university press.

[3] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. 28(8):859 – 881.

[4] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. 13(4):451 – 490.

[6] John H. Reif and Harry R. Lewis. Efficient symbolic analysis of programs. 32(3):280 – 314.

[7] John H. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '77, pages 104 – 118. ACM Press.

[8] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '88, pages 12 – 27. ACM Press.

[9] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. 13(2):181 – 210.

[10] Michael J. Wolfe and Michael Joseph Wolfe. High Performance Compilers for Parallel Computing. Addison-Wesley.