# Lecture 10: Hoare Logic

## 1 从手工证明到自动验证器

Hoare 逻辑用三元组 $\{P\}\ C\ \{Q\}$ 描述程序规约：从满足前置条件 $P$ 的状态执行 $C$，若正常终止，则结果状态满足后置条件 $Q$。赋值、顺序、分支与循环规则把整个程序的证明分解为局部证明。本讲要回答的是：验证器怎样把这些规则组织成一个可以机械执行的过程?

验证器接收程序以及用户给出的前置条件、后置条件和循环不变式。它先把程序表示为控制流图，再把规约与不变式放入图中，最后从程序末端向入口反向计算，得到验证条件（**Verification Condition**，**VC**）。证明这个公式对所有初始状态都成立，就完成了相应的程序正确性证明。**SMT** 求解器是自动检查逻辑公式的工具：它判断能否找到一组变量取值，使公式成立，也能处理本讲中的整数运算和比较。

程序与规约 $\longrightarrow$ 带验证语句的 CFG $\longrightarrow$ 逻辑公式 VC $\longrightarrow$ SMT 求解器

最弱自由前置条件。$\mathrm{wlp}(C, Q)$（weakest liberal precondition）描述所有这样的起始状态：从这些状态执行代码语句 $C$，不会出现断言检查失败，而且正常结束时满足 $Q$。“最弱”意味着允许的起始状态最多：任何足以保证这些要求的前置条件 $P$ 都必须蕴含它。因此，验证 $\{P\}\ C\ \{Q\}$，就是证明下面的公式对所有变量取值都成立：

$$P \to \mathrm{wlp}(C, Q).$$

这里的 $\to$ 表示“如果前面的条件成立，那么后面的条件也成立”。引入验证语句后，还需要保证执行不发生断言错误。

记号说明。程序状态记录各变量当前的值，断言是这些值需要满足的条件。例子中的 `int` 按数学整数理解；特别地，$a > 0 \to a - 1 \geq 0$ 使用了整数这一前提。代码采用带规约的伪代码：赋值写作 $x := E$，相等关系写作 $x = E$。$Q[E/x]$ 表示把条件 $Q$ 中代表程序变量 $x$ 的位置替换为表达式 $E$。例如，若 $Q$ 是 $x > 0$，把 $x$ 换成 $a+1$ 后，就得到 $a + 1 > 0$。

## 2 控制流图：把程序结构变成节点和边

### 2.1 基本块与四种构造方式

控制流图（**Control-Flow Graph**，**CFG**）用有向图表示程序执行时可能经过的路径。节点通常对应基本块（**basic block**）：一段顺序执行的代码，其内部没有跳转，也没有其他位置跳入的目标；跳转目标开始一个块，发生跳转时结束当前块。有向边表示接下来可能执行哪个块。一个块的后继，就是沿一条箭头可以直接到达的块。条件判断的真边、假边分别表示条件为真、为假时选择的路径。

| 程序结构 | **CFG** 的连接方式 |
|---|---|
| 顺序 $C_1; C_2; \ldots; C_n$ | 按执行顺序连接。连续的直线代码可放在同一个块中；若子程序含分支，则递归构图后连接其出口与后继入口。 |
| `if` $(b)$ $C_1$ `else` $C_2$ | 判断节点的真边进入 $\mathrm{CFG}(C_1)$，假边进入 $\mathrm{CFG}(C_2)$；两条分支结束后汇合，继续执行后续代码。 |
| `if` $(b)$ $C$ | 真边进入 $\mathrm{CFG}(C)$，假边直接到后续代码；相当于另一分支是 `skip`，即不执行任何操作。 |
| `while` $(b)$ $C$ | 在检查条件 $b$ 的位置（循环头），真边进入循环体，假边通向循环后的代码。循环体结束后返回循环头的边称为回边，它使图中出现环。 |

例如，`a := a+1; b := 3; b := a+b;` 可以放入一个基本块；`if(a==0) b:=3; else b:=4;` 则需要两条分支。`return a` 在验证表示中记为 $result := a$，随后检查关于返回值的后置条件。

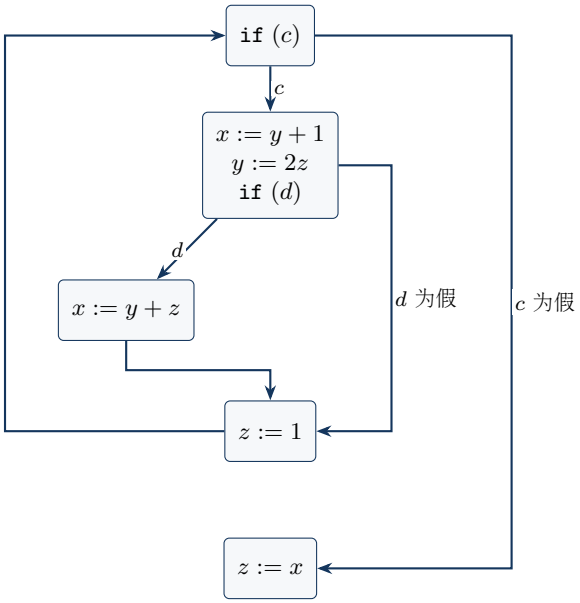
读图时要区分两种方向：程序沿箭头向前执行，而后续的 wlp 计算把证明要求向后传播。图记录执行顺序；反向计算并不改变程序的执行顺序。

### 2.2 构图示例：循环中嵌套条件语句

下面的程序把一个条件语句嵌入循环体。先画外层 `while` 的真、假两条边，再在真分支内展开顺序语句和 `if`。

```
while (c) {
    x := y + 1;
    y := 2 * z;
    if (d) x := y + z;
    z := 1;
}
z := x;
```

$c$、$d$ 表示相应位置的布尔条件。每次回到循环头，都重新检查 $c$；若 $d$ 为假，只跳过 $x := y+z$，仍须执行 $z := 1$。




检查构图的两个位置：循环体末尾 $z := 1$ 的后继是循环头，而不是 $z := x$；只有循环条件 $c$ 为假时，才执行循环后的 $z := x$。这些连接决定了后面必须证明哪些执行路径。

# 3 用 assume 与 assert 表达规约

仅有赋值和边，还不能表达“我们允许哪些输入”与“程序必须保证什么”。因此，验证器引入两个不修改变量值的语句：

| 语句 | 条件为真时 | 条件为假时 |
|---|---|---|
| `assume` $P$ | 保持当前状态，继续执行。 | 此路径不满足假设，不再继续检查；这不会产生断言错误。 |
| `assert` $P$ | 保持当前状态，继续执行。 | 进入错误状态，验证必须排除这种情况。 |

因此，它们的反向规则不同：

$$\boxed{\mathrm{wlp}(\mathsf{assume}\ P, Q) = P \to Q} \qquad \boxed{\mathrm{wlp}(\mathsf{assert}\ P, Q) = P \wedge Q}$$

对于 `assume`，只有满足 $P$ 的状态需要保证后续要求 $Q$；对于 `assert`，当前检查 $P$ 和后续要求 $Q$ 都必须成立。假设条件为假时，路径不再继续；断言条件为假时，则发生错误。因此，不能把待证明的断言换成假设，否则只是排除了失败路径。

把 $\{P\}\ C\ \{Q\}$ 编码为 `assume` $P$；$C$；`assert` $Q$ 后，在最后一个断言之后没有其他要求，用 $T$ 表示恒真的条件。整个验证任务可写成

$$\mathrm{wlp}(\mathsf{assume}\ P; C; \mathsf{assert}\ Q, T) = P \to \mathrm{wlp}(C, Q).$$

这正是 Hoare 三元组所要求的蕴含关系。前置条件是调用者提供的假设，后置条件是实现必须兑现的保证；把它们放进图中后，验证器只需统一检查所有 `assert`。

# 4 把无环 CFG 转换成逻辑公式

## 4.1 每个块的正确性条件

对基本块 $A$，用 $C_1$ 表示其块内代码语句，用 $\mathrm{Succ}(A)$ 表示它的所有后继。$A_{\mathrm{ok}}$ 是进入块 $A$ 时各变量需要满足的条件：满足这个条件后，从 $A$ 开始，沿所有符合分支条件和假设的路径执行，都不会发生断言错误。它概括的是从 $A$ 开始的整个剩余程序，不只是块内几条语句。

$$\boxed{A_{\mathrm{ok}} \equiv \mathrm{wlp}\left(C_1, \bigwedge_{B \text{ is } \mathrm{Succ}(A)} B_{\mathrm{ok}}\right)}$$

大合取符号表示把所有后继块的正确性条件用 $\wedge$ 连起来；下标说明 $B$ 遍历 $A$ 的所有后继。这个式子把“块内顺序执行”和“块间控制转移”分开处理：

- 没有后继：后续没有额外要求，所以 $A_{\text{ok}} = \text{wlp}(C_1, T)$。后置条件已作为 `assert` 放入块内，取 $T$ 不会把它丢掉。
- 一个后继 $B$：$A_{\text{ok}} = \text{wlp}(C_1, B_{\text{ok}})$，就是把后继的要求当作当前块的后置条件。
- 多个后继：执行可能进入其中任意一个，故必须保证所有满足分支条件的后续路径都正确。用 $\wedge$ 表示这些要求必须同时成立；$\vee$ 只要求其中一个成立，不能满足这里的要求。路径能否继续，由后继中的 `assume` 条件决定。

无环图的计算次序。先计算无后继块，再计算所有后继都已处理的块，一直回到入口。块内的三条基本规则分别处理不改变状态、赋值替换和顺序组合：

$$\begin{aligned}\text{wlp}(\mathsf{skip}, Q) &= Q,\\ \text{wlp}(x := E, Q) &= Q[E/x],\\ \text{wlp}(C_1; C_2, Q) &= \text{wlp}(C_1, \text{wlp}(C_2, Q)).\end{aligned}$$

这里的 $B_{\text{ok}}$ 代表一个具体条件，例如 $a = 2$。当前块若对 $a$ 赋值，就要在这个条件中替换 $a$；不能把 $B_{\text{ok}}$ 当作固定的真或假。入口已包含前置条件的 `assume` 时，最终要证明入口条件对所有变量取值都成立，也就是恒真。

### 4.2 顺序程序：从后置条件反推输入要求

原规约是 $\{a = 1\}\ a := a + 1; \mathit{result} := a\ \{\mathit{result} = 2\}$。将程序分为 $A \to B$ 两块：

```
A: assume a = 1;
   goto B;
B: a := a + 1;
   result := a;
   assert result = 2;
   // No successor.
```

这里的 `goto` 只是对 CFG 边的文字表示；无后继表示当前验证路径结束。反向从块 $B$ 开始：

$$\begin{aligned}B_{\text{ok}} &= \text{wlp}(a := a + 1; \mathit{result} := a; \mathsf{assert}\ (\mathit{result} = 2), T)\\ &= \text{wlp}(a := a + 1; \mathit{result} := a, \mathit{result} = 2)\\ &= \text{wlp}(a := a + 1, a = 2)\\ &= (a + 1 = 2),\\ A_{\text{ok}} &= \text{wlp}(\mathsf{assume}\ (a = 1), B_{\text{ok}})\\ &= (a = 1 \to a + 1 = 2) \equiv T.\end{aligned}$$

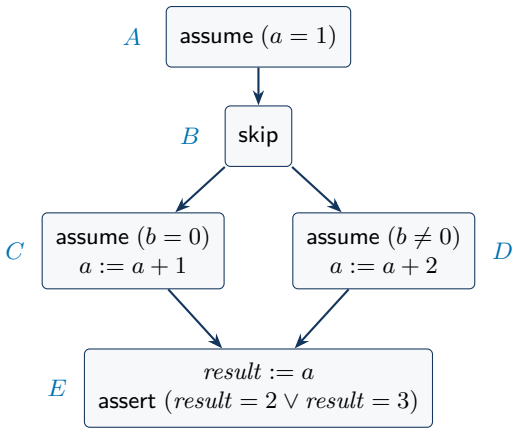
注意先跨过最后一条语句，再跨过前一条语句；例如 $\mathit{result} := a$ 把关于返回值的要求变成关于 $a$ 的要求。这与直接证明 $a = 1 \to \text{wlp}(a := a + 1; \mathit{result} := a, \mathit{result} = 2)$ 完全一致。

## 5 分支：把条件放进路径，再合并证明要求

### 5.1 从条件判断到带假设的两条路径

下面的程序根据 $b = 0$ 是否成立，分别把 $a$ 增加 1 或 2。若在 CFG 中只保留两条边而丢掉 $b = 0$ 与 $b \neq 0$，后面的公式就无法知道一条路径在什么条件下可执行。因此，把分支点改为 `skip`，在两条后继路径入口分别加入 `assume`。

```
// Pre: a = 1
int f(int a, int b) {
    if (b == 0)
        a := a + 1;
    else
        a := a + 2;
    return a;
}
// Post: result = 2
//       or result = 3
```



### 5.2 从汇合点向入口逐块计算

第一步：共同后继 $E$。两条路径最后都要满足同一个后置条件：

$$\begin{aligned} E_{\text{ok}} &= \text{wlp}(result := a; \textsf{assert}\ (result = 2 \lor result = 3), T) \\ &= \text{wlp}(result := a, result = 2 \lor result = 3) = (a = 2 \lor a = 3). \end{aligned}$$

第二步：分别处理 $C$ 与 $D$。每条路径先逆向处理赋值，再处理入口假设：

$$\begin{aligned} C_{\text{ok}} &= \text{wlp}(\textsf{assume}\ (b = 0); a := a + 1, E_{\text{ok}}) \\ &= b = 0 \to (a + 1 = 2 \lor a + 1 = 3), \\ D_{\text{ok}} &= \text{wlp}(\textsf{assume}\ (b \neq 0); a := a + 2, E_{\text{ok}}) \\ &= b \neq 0 \to (a + 2 = 2 \lor a + 2 = 3). \end{aligned}$$

第三步：回到分支点 $B$，再回到入口 $A$。

$$\begin{aligned} B_{\text{ok}} &= \text{wlp}(\textsf{skip}, C_{\text{ok}} \land D_{\text{ok}}) = C_{\text{ok}} \land D_{\text{ok}}, \\ A_{\text{ok}} &= \text{wlp}(\textsf{assume}\ (a = 1), B_{\text{ok}}) \\ &= (a = 1) \to \Big[ \big(b = 0 \to (a + 1 = 2 \lor a + 1 = 3)\big) \\ &\qquad\qquad \land \big(b \neq 0 \to (a + 2 = 2 \lor a + 2 = 3)\big) \Big]. \end{aligned}$$

当 $a = 1$ 时，两条蕴含式的结论分别为 $2 = 2 \lor 2 = 3$ 与 $3 = 2 \lor 3 = 3$，均为真，因此入口条件对任意 $a, b$ 都成立。

为什么合取不会要求同时走两个分支？若 $b = 0$，则 $D$ 中的假设为假，$D_{\text{ok}}$ 的蕴含式自动成立，实际只需满足 $C$ 的要求；$b \neq 0$ 时反之。因此，每条分支都只需在自己的入口条件成立时保证后续执行正确，再将两条分支的要求用 $\land$ 连接。

## 6 循环：为什么需要不变式和 havoc？

### 6.1 有环图不能直接按刚才的次序计算

考虑下面的递减循环：输入满足 $a \geq 100$，程序要求正常返回时的值为 $0$。注解 `invariant a >= 0` 声明在每次检查循环条件时都应保持 $a \geq 0$。

```
// Pre: 100 <= a
int example(int a) {
    while (0 < a)
        invariant a >= 0
    {
        a := a - 1;
    }
    return a;
}
// Post: result = 0
```

像处理分支一样，循环头的两个后继分别以 `assume` $(a > 0)$ 和 `assume` $(a \leq 0)$ 开始。但循环体又返回循环头，导致循环头的正确性条件依赖循环体，而循环体又依赖循环头。无环图中“先算后继”的有限计算顺序不再适用。

直接删除回边只保留了从初始状态出发的一次循环体检查，无法覆盖后续迭代。要切断循环，必须用一个对任意迭代成立的证明来补偿，而不能简单少检查一些执行。

### 6.2 while 规则的三个证明任务

设前置条件为 $P$，循环条件为 $b_1$，循环体为 $C$，不变式为 $I$。$istrue(b_1)$ 表示循环条件为真，$isfalse(b_1)$ 表示循环条件为假。while 规则说明：只要循环体保持不变式，循环正常退出时，不变式仍成立，且循环条件为假：

$$\frac{\{I \land istrue(b_1)\}\ C\ \{I\}}{\{I\}\ \texttt{while}\ (b_1)\ C\ \{I \land isfalse(b_1)\}}.$$

令 $R$ 表示循环结束后、执行剩余代码前所需满足的条件。再将原始前置条件和循环后的要求接到这条规则两端，就得到三个证明任务：

| | | |
|---|---|---|
| 初始化： | $P \to I$, | 第一次进入循环头时，不变式成立； |
| 保持性： | $I \land istrue(b_1) \to \text{wlp}(C, I)$, | 任意一次执行循环体都保持不变式； |
| 退出： | $I \land isfalse(b_1) \to R$, | 退出循环时，足以证明剩余代码正确。 |

本例中 $I$ 为 $a \geq 0$，$b_1$ 为 $0 < a$，而 $R = \mathrm{wlp}(result := a, result = 0) \equiv a = 0$。三个任务分别是

$$100 \leq a \to a \geq 0, \qquad (a \geq 0 \wedge a > 0) \to a - 1 \geq 0, \qquad (a \geq 0 \wedge a \leq 0) \to a = 0.$$

不变式使我们可以从“第一次成立”和“每次都保持”推出“任意次迭代后都成立”。这是归纳证明，而不是把循环实际展开有限次。

### 6.3 只有 assert 与 assume，还缺少什么？

按照这三个任务，可以在首次进入循环前加入 `assert` $I$，在循环体末尾加入 `assert` $I$，并在循环头加入 `assume` $I$。但若此时直接切断回边，进入循环体的状态仍带着 $100 \leq a$ 这一初始值约束。

这会把保持性错误地削弱为只检查

$$(100 \leq a \wedge a \geq 0 \wedge a > 0) \to a - 1 \geq 0.$$

它只覆盖第一次迭代可能见到的 $a$；真正的归纳步骤必须覆盖之后的 $a = 99, 98, \ldots, 1$ 等情况。更明显地，若退出分支仍携带 $100 \leq a$，再加上 $a \leq 0$ 就产生矛盾，使退出检查自动通过，根本没有证明循环若干次之后的返回值。

因此，需要在开始检查“任意一次迭代”之前，忘掉循环可能改变的变量的旧值，再只用不变式约束其新值。这就是 `havoc` 的作用。

### 6.4 havoc：用任意值代表某次迭代的状态

`havoc` $x$ 把 $x$ 设为其类型中的任意值，其他变量保持不变。验证必须对每一种可能选择都成立，而不是挑一个有利的值。因此：

$$\boxed{\mathrm{wlp}(\text{havoc } x, Q) = (\forall v)\ Q[v/x]}$$

这里选用公式中尚未使用的名字 $v$，表示 $x$ 可能得到的新值；$v$ 的取值范围与 $x$ 相同。$(\forall v)$ 表示后面的条件必须对每个可能的 $v$ 都成立。 特别地，

$$\mathrm{wlp}(\text{havoc } x; \text{assume } I, Q) = (\forall v)\ \big(I[v/x] \to Q[v/x]\big).$$

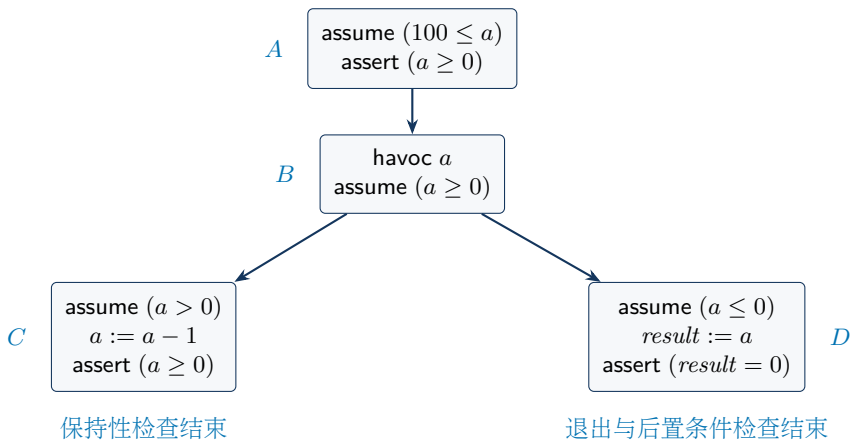
这要求每个满足不变式的新值都能保证后续要求成立；不能只检查一个或几个选定的值。

循环体中可能被更新的变量称为循环修改变量（**loop targets**）。切断循环前，需要对这些变量执行 `havoc`。本例循环体只有 $a := a - 1$，所以只需处理变量 $a$；$result$ 在循环结束后才被赋值，不属于此循环的修改变量。第 2.2 节构图示例中的循环修改变量则为 $x$、$y$、$z$。

“忘掉旧值”是有范围的。应消除的是循环修改变量上的旧约束；循环不修改的变量仍保持原值。例如本例不能继续把任意迭代时的 $a$ 限定为 $a \geq 100$，但随后可以通过 `assume` $(a \geq 0)$ 恢复不变式提供的信息。

## 7 把循环改写成无环的验证任务

对本例完成上述处理后，得到下图。把四个基本块命名为 $A, B, C, D$，边为 $A \to B$、$B \to C$、$B \to D$；$C$ 与 $D$ 都没有后继。




1. 块 $A$：先证明初始化。在原始输入满足 $100 \leq a$ 的前提下检查 $a \geq 0$。此处还需要初始信息，所以 `assert` 必须位于 `havoc` 之前。

2. 块 $B$：建立任意迭代的起点。先 `havoc` $a$，再 `assume` $(a \geq 0)$；两者共同表示任意满足不变式的循环头状态。
3. 块 $C$：证明一次循环体保持不变式。假设循环条件为真，执行循环体，再检查 $I$。随后切断原来的回边，检查路径在此结束。
4. 块 $D$：证明循环退出后的要求。假设循环条件为假，执行返回值赋值，再检查原始后置条件。

为什么没有 $C \to D$? 块 $C$ 检查的是“某一次继续迭代”，它执行一次后未必退出循环。块 $D$ 检查的是“某个循环头状态已经满足退出条件”。两者是从任意循环头状态分出的两个证明任务，不能串成“一次循环体后立即返回”。

得到的无环图用来分别检查三个证明任务，不是把原循环实际运行一遍。初始化证明第一次进入循环时 $I$ 成立，保持性证明每执行一轮后 $I$ 仍成立，退出任务证明条件为假时能得到所需结论。因此，循环正常退出时，所需的后置条件成立。

# 8 循环例子的完整反向推导

## 8.1 先计算两个没有后继的块

退出块 $D$。无后继，所以从 $T$ 开始：

$$\begin{aligned} D_{\text{ok}} &= \text{wlp}(\textsf{assume } (a \leq 0); \mathit{result} := a; \textsf{assert } (\mathit{result} = 0), T) \\ &= \text{wlp}(\textsf{assume } (a \leq 0); \mathit{result} := a, \mathit{result} = 0) \\ &= \text{wlp}(\textsf{assume } (a \leq 0), a = 0) \\ &= (a \leq 0 \to a = 0). \end{aligned}$$

保持性块 $C$。末尾的不变式检查同样不能省略：

$$\begin{aligned} C_{\text{ok}} &= \text{wlp}(\textsf{assume } (a > 0); a := a - 1; \textsf{assert } (a \geq 0), T) \\ &= \text{wlp}(\textsf{assume } (a > 0); a := a - 1, a \geq 0) \\ &= \text{wlp}(\textsf{assume } (a > 0), a - 1 \geq 0) \\ &= (a > 0 \to a - 1 \geq 0). \end{aligned}$$

此时 $D_{\text{ok}}$ 不必单独恒真，例如 $a < 0$ 时它为假。这不是证明失败：它描述的是从 $D$ 入口开始所需的条件，接下来还要结合块 $B$ 中的假设 $a \geq 0$。

## 8.2 在块 B 合并路径，再处理 havoc

先处理后面的 **`assume`**，再处理前面的 **`havoc`**：

$$\begin{aligned} B_{\text{ok}} &= \text{wlp}(\textsf{havoc } a; \textsf{assume } (a \geq 0), C_{\text{ok}} \wedge D_{\text{ok}}) \\ &= \text{wlp}(\textsf{havoc } a, (a \geq 0) \to (C_{\text{ok}} \wedge D_{\text{ok}})) \\ &= (\forall v)\ \Big[ v \geq 0 \to \big((v > 0 \to v - 1 \geq 0) \wedge (v \leq 0 \to v = 0)\big) \Big]. \end{aligned}$$

这里 $v$ 取任意整数，$C_{\text{ok}}$ 和 $D_{\text{ok}}$ 中的 $a$ 也必须一起替换为 $v$。公式要求每个可能的新值都满足相应条件；不能继续把这些新值限制在原始输入的范围 $a \geq 100$ 内。

这个公式恒真：任取整数 $v \geq 0$，若 $v > 0$，因为 $v$ 是整数，所以 $v \geq 1$，进而 $v - 1 \geq 0$；若 $v \leq 0$，结合 $v \geq 0$ 得 $v = 0$。这分别验证了保持性与退出任务。

## 8.3 回到入口 A，得到最终 VC

最后处理初始化：

$$\begin{aligned} A_{\text{ok}} &= \text{wlp}(\textsf{assume } (100 \leq a); \textsf{assert } (a \geq 0), B_{\text{ok}}) \\ &= \text{wlp}(\textsf{assume } (100 \leq a), (a \geq 0) \wedge B_{\text{ok}}) \\ &= (100 \leq a) \to \big((a \geq 0) \wedge B_{\text{ok}}\big). \end{aligned}$$

将 $B_{\text{ok}}$ 展开，就得到完整的验证条件：

$$\boxed{A_{\text{ok}} \equiv (100 \leq a) \to \Big[ a \geq 0 \wedge (\forall v)\ \big(v \geq 0 \to ((v > 0 \to v - 1 \geq 0) \wedge (v \leq 0 \to v = 0))\big) \Big].}$$

初始化由 $a \geq 100$ 立即推出；量词部分已在上一步证明。因此，入口条件 $A_{\text{ok}}$ 对任意整数 $a$ 都成立。从满足给定前置条件的状态执行原程序，若正常返回，返回值必为 0。

# 9 把整套过程连起来

从带规约的程序到最终的逻辑公式，验证器完成的是下面这条证明流程：

1. 把程序变成 **CFG**。保留顺序、分支、汇合和循环回边；用 *result* 表示返回值。
2. 把条件变成验证语句。入口加入 `assume` 前置条件，出口加入 `assert` 后置条件；分支入口加入对应条件或其否定。
3. 用不变式处理循环。检查不变式的初始化和保持性；对循环修改变量执行 `havoc`，随后假设不变式；保留退出分支并切断回边。
4. 从没有后继的块开始反向计算。每块先用 $\wedge$ 连接所有后继的正确性条件，再用 wlp 从后往前处理块内语句，直到得到入口条件。
5. 证明入口条件有效。把所得公式交给 SMT 求解器，检查它是否对所有输入状态成立。