数据流优化之GVN

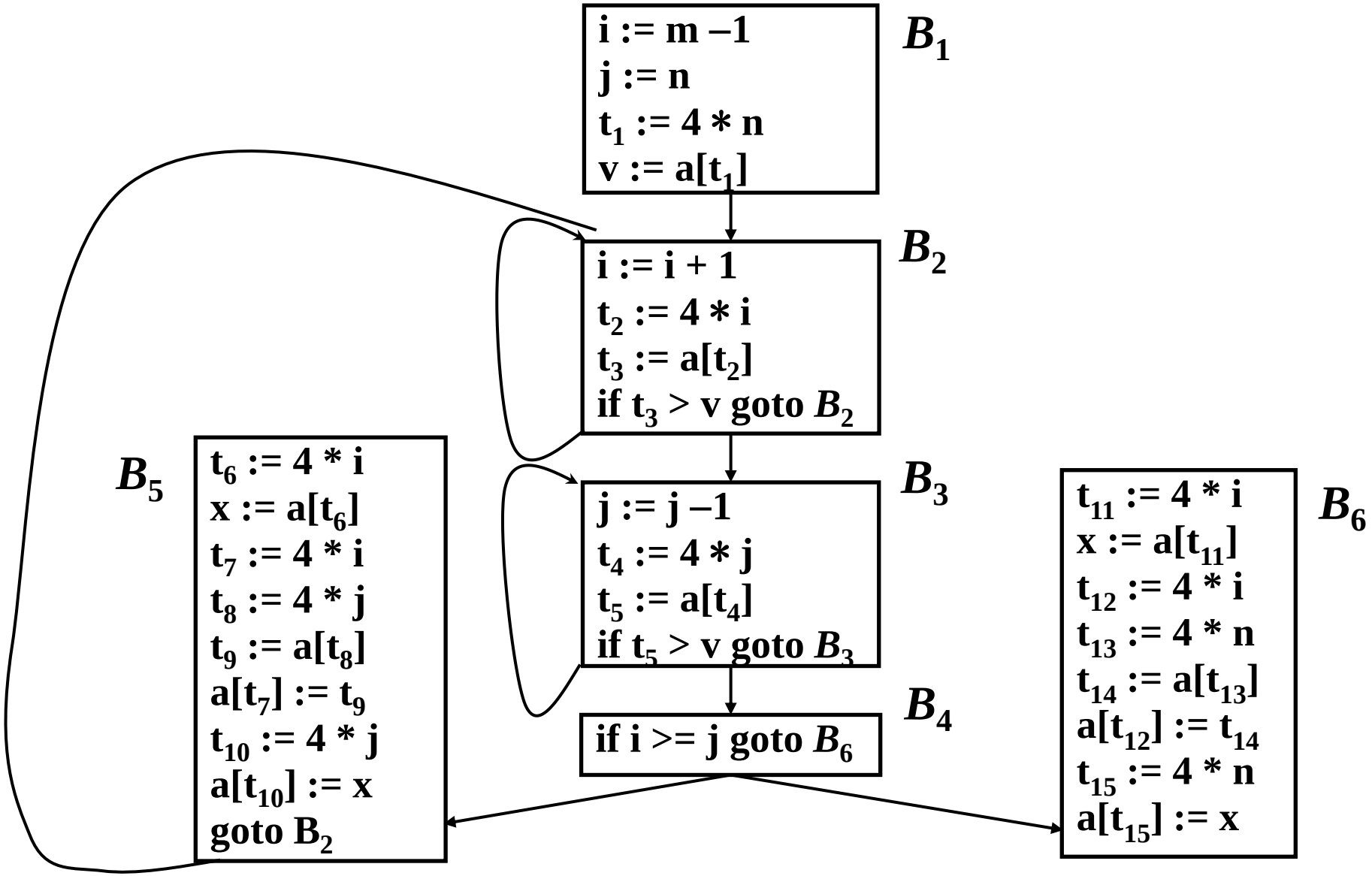
李 诚

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

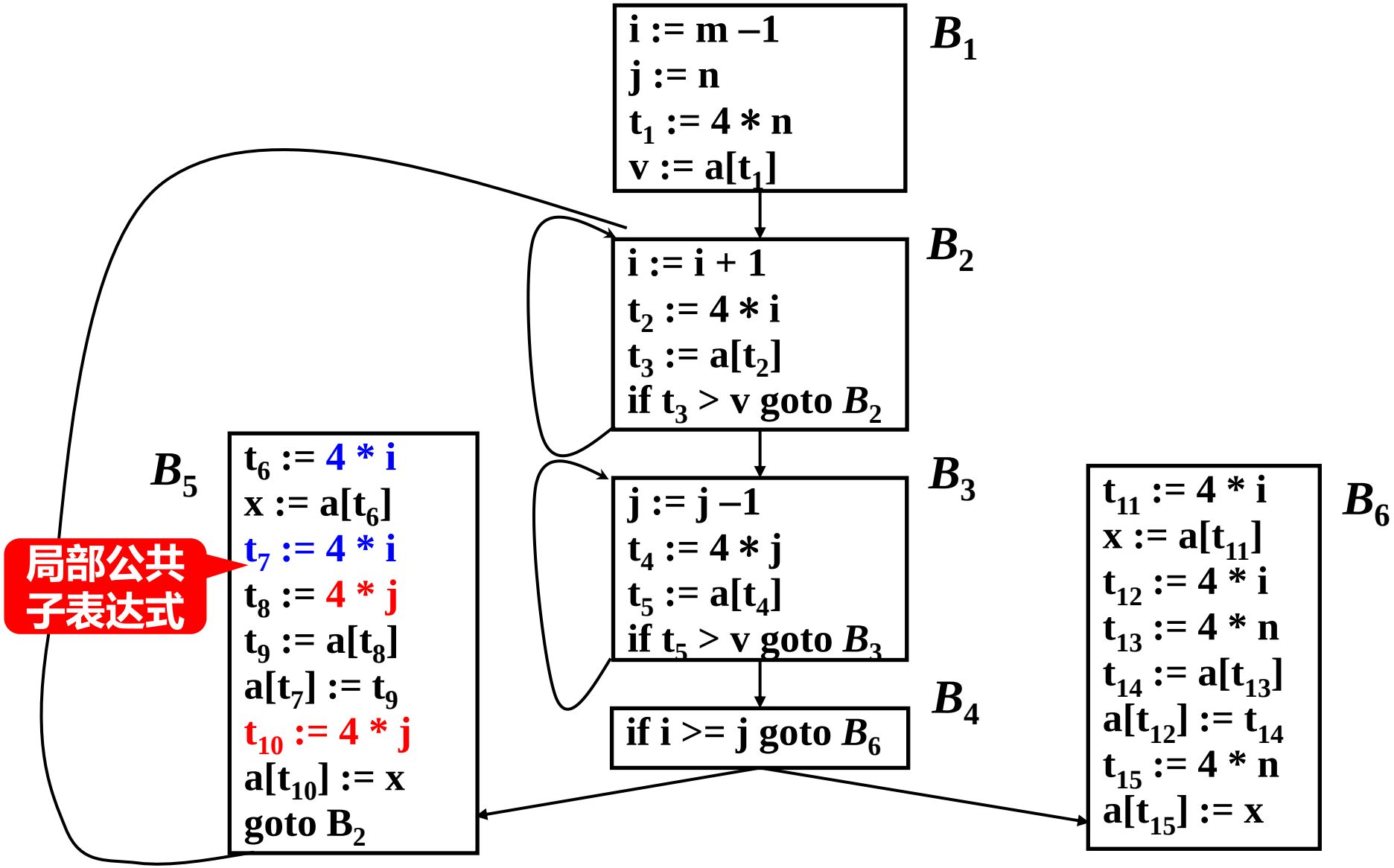
计算机科学与技术学院

2025年12月10日

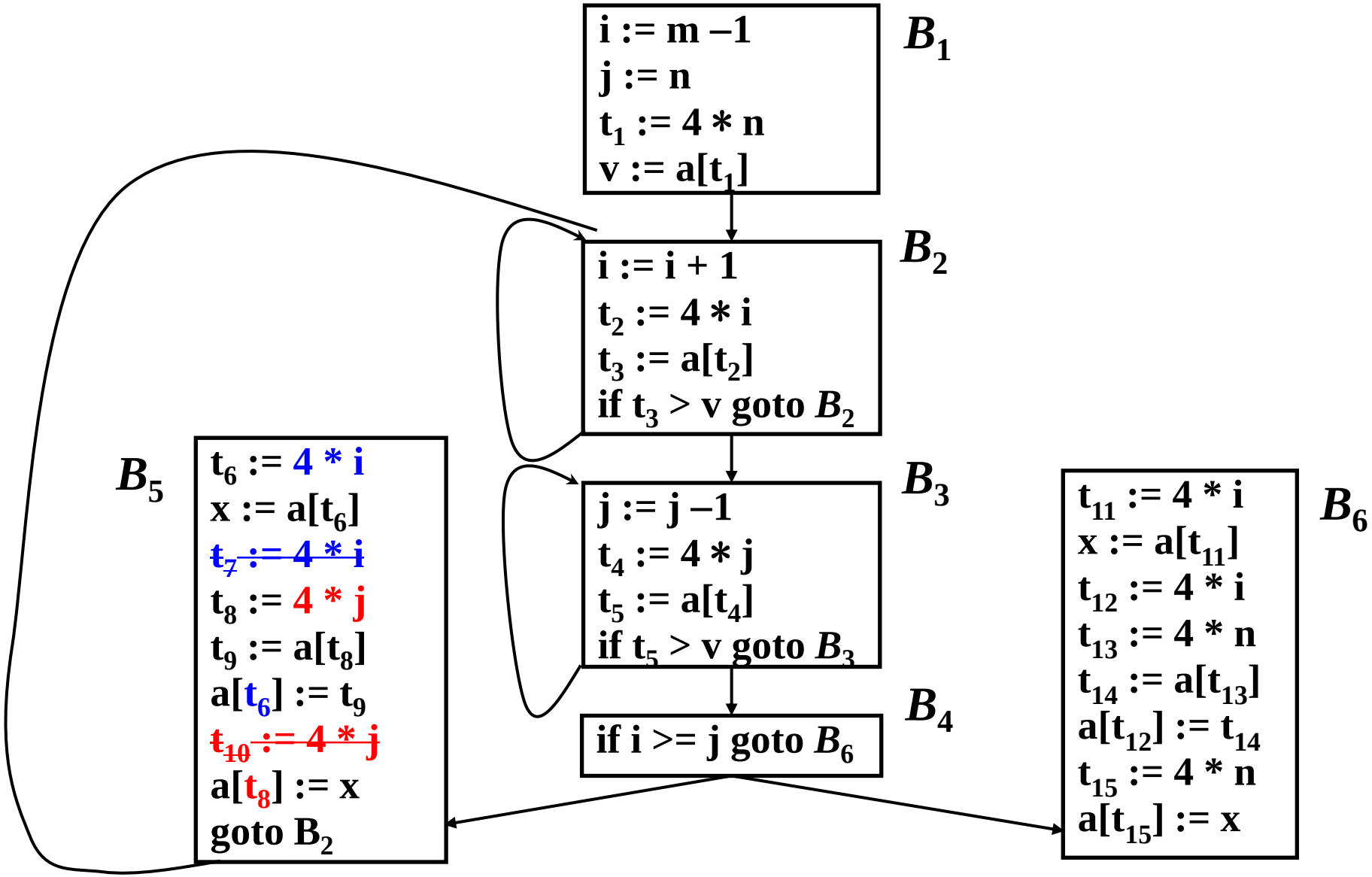
公共子表达式删除-回顾


i := m –1
j := n
t1 := 4 * n
v := a[t1]
B1
i := i + 1
t2 := 4 * i
t3 := a[t2]
if t3 > v goto B2
B2
j := j –1
t4 := 4 * j
t5 := a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
t6 := 4 * i
x := a[t6]
t7 := 4 * i
t8 := 4 * j
t9 := a[t8]
a[t7] := t9
t10 := 4 * j
a[t10] := x
goto B2
B5
t11 := 4 * i
x := a[t11]
t12 := 4 * i
t13 := 4 * n
t14 := a[t13]
a[t12] := t14
t15 := 4 * n
a[t15] := x
B6

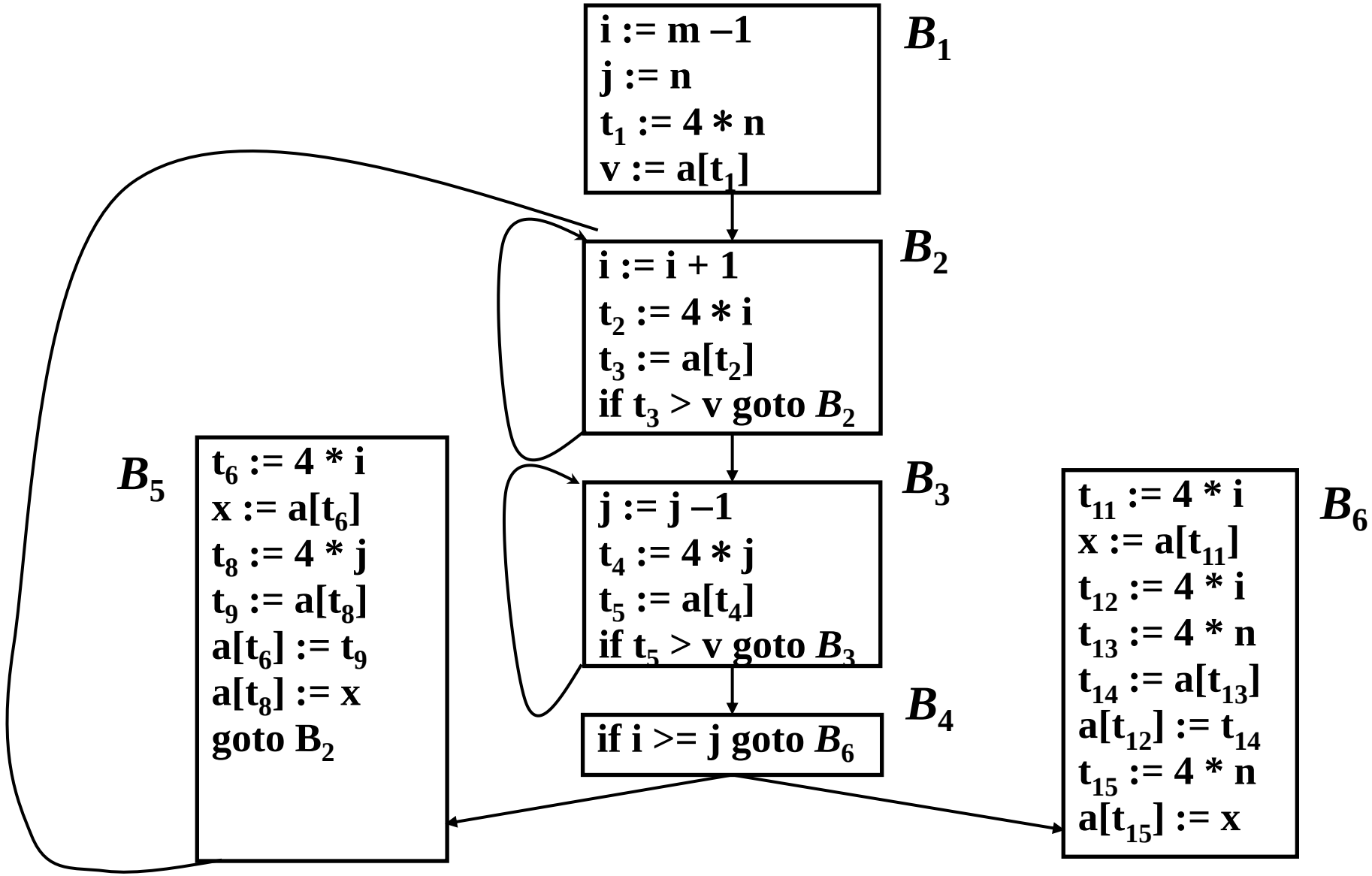
快排中的公共子表达式删除



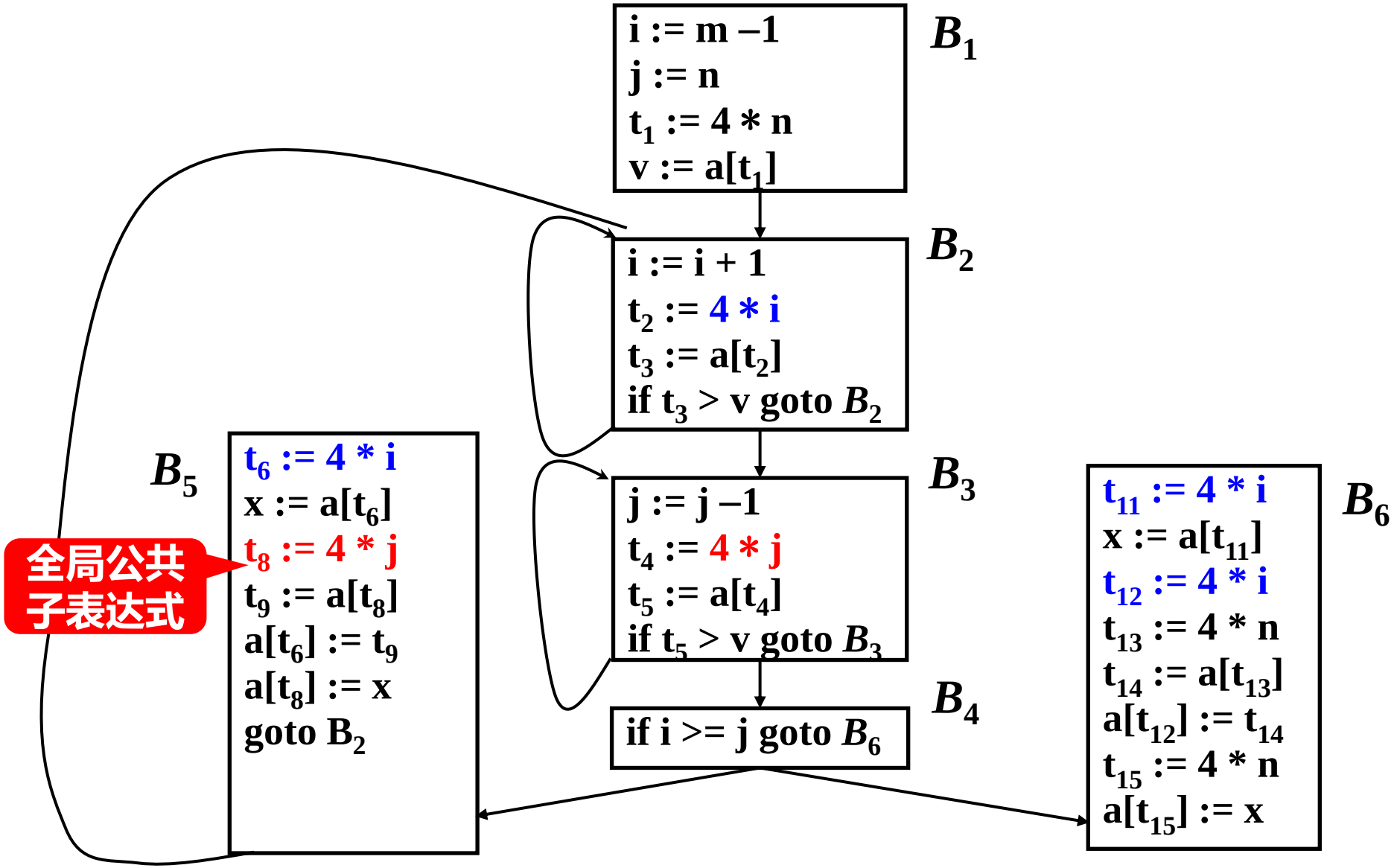
快排中的公共子表达式删除



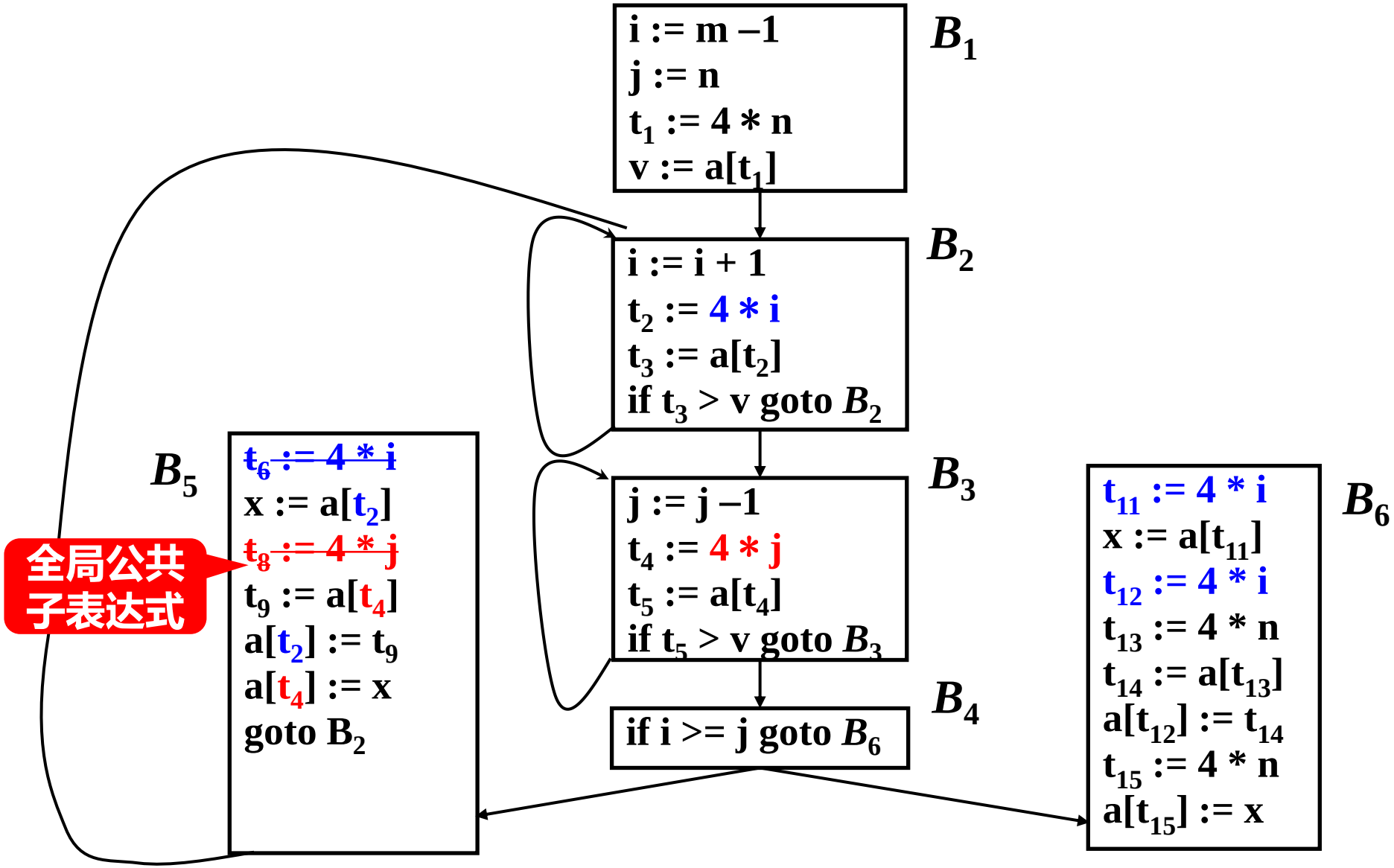
快排中的公共子表达式删除



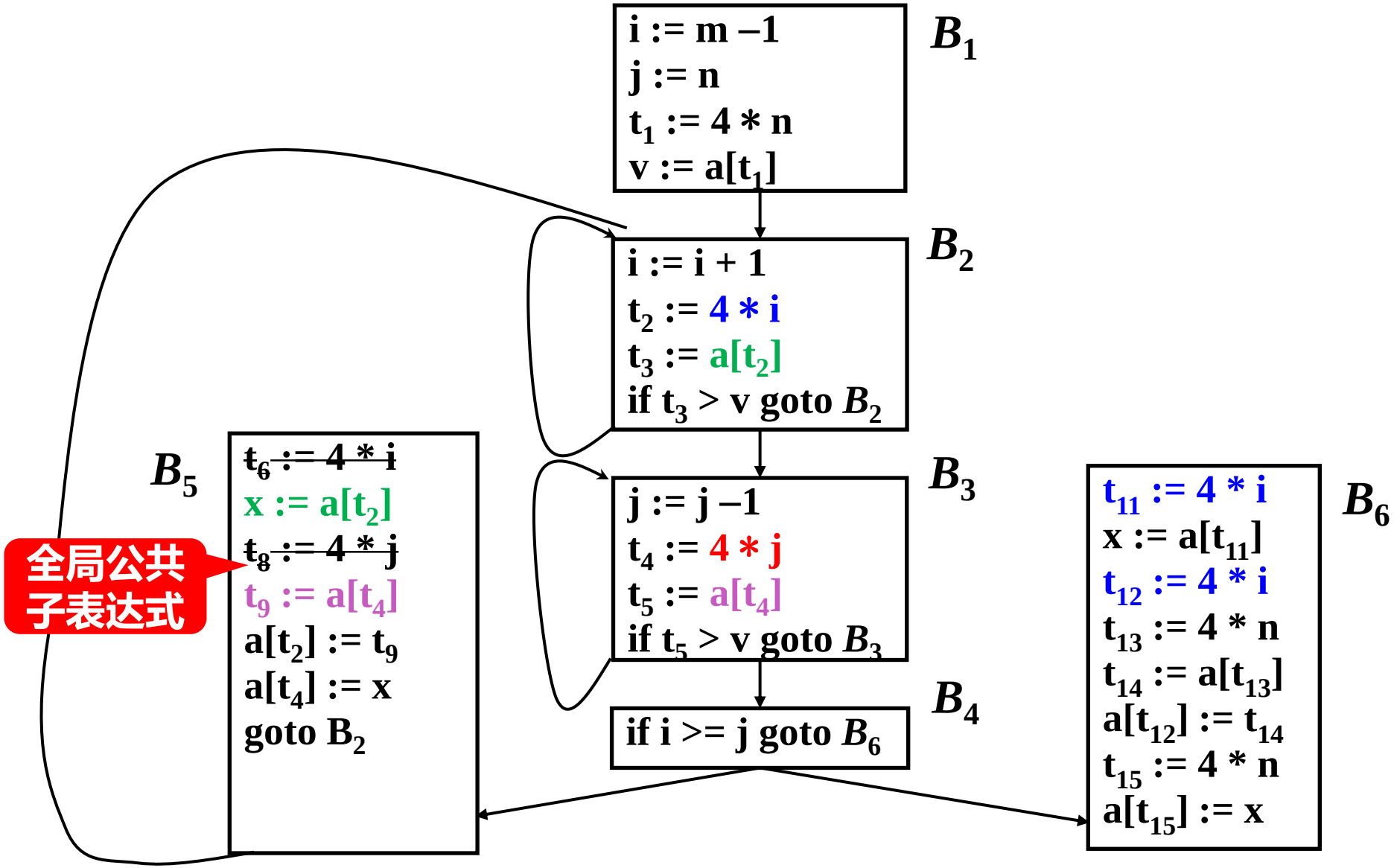
快排中的公共子表达式删除



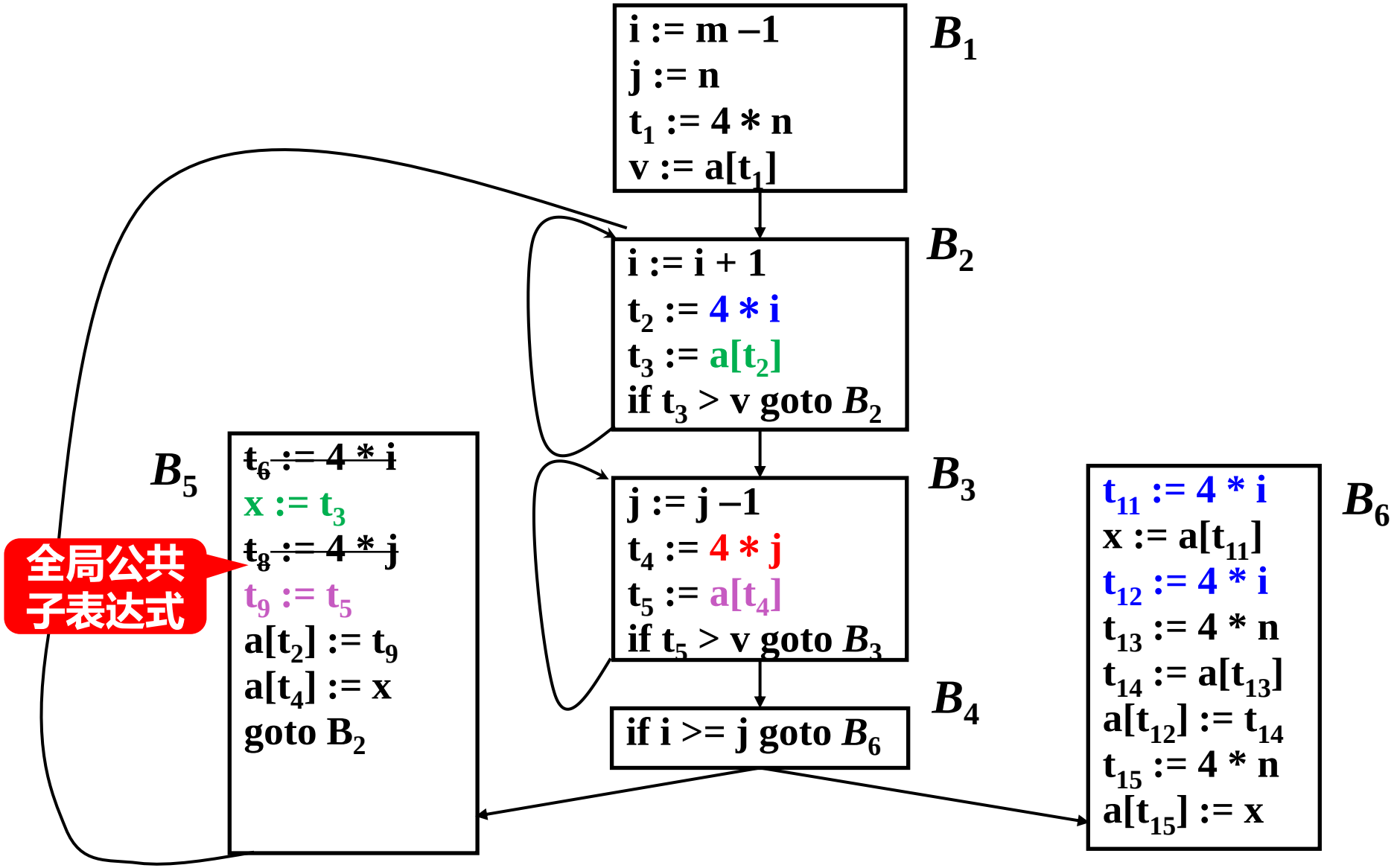
快排中的公共子表达式删除



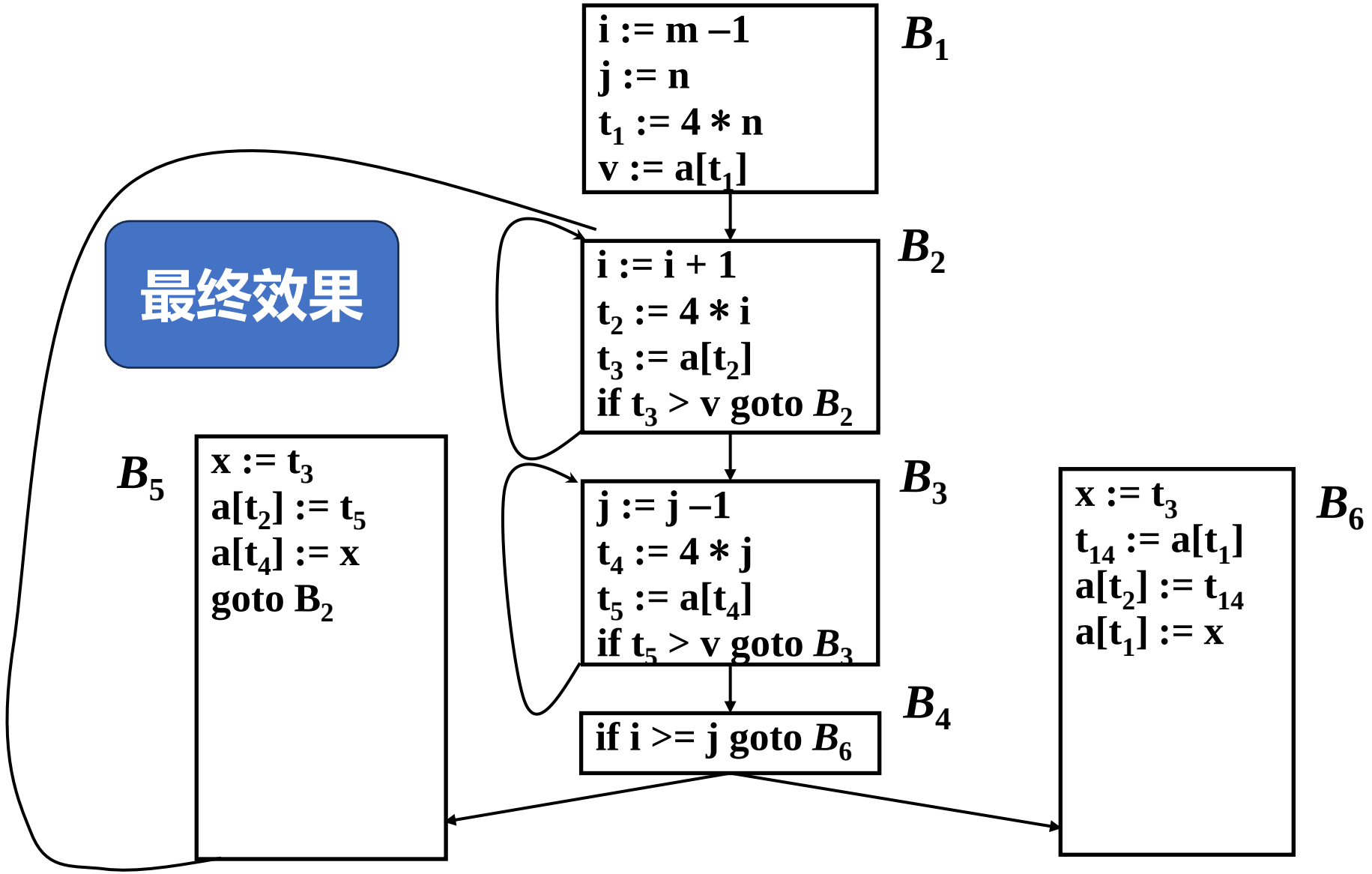
快排中的公共子表达式删除



快排中的公共子表达式删除



快排中的公共子表达式删除



如何实现冗余代码的删除?

- **关键在于冗余代码的识别**
  - 在程序点p和到达p的路径l，x = e，在l上p之前，是否存在一个与e计算结果相同的子表达式e′
- **通过值编号value numbering方法识别**
- **全局值编号global value numbering需要借助数据流分析方法**

值编号 Value Numbering

- 考虑以下基本块代码

```
x = a + 37
y = x + 42
x = a + 37
y = x + 42
```

- 冗余计算是 a + 37 和 x + 42
- 最后两条指令的计算可以删除，直接用前面的x和y来代替

值编号 Value Numbering

- 转变成静态单赋值格式SSA

```
x = a + 37          x1 = a + 37
y = x + 42          y1 = x1 + 42
x = a + 37          x2 = a + 37
y = x + 42          y2 = x2 + 42
```

- 每一次赋值产生一个新的变量
- 每一次引用使用唯一确定的变量
  - x2 = x1; y2 = y1
  - 如何识别？

值编号 Value Numbering

- 在SSA中，为每一个变量或表达式的结果分配一个值编号v_i

```
x = a + 37
y = x + 42
x = a + 37
y = x + 42
```

```
x1 = a + 37
y1 = x1 + 42
x2 = a + 37
y2 = x2 + 42
```

```
a: v1
v1 + 37: v2 // a + 37
x1: v2
```

值编号 Value Numbering

- 在SSA中，为每一个变量或表达式的结果分配一个值编号v_i

```
x = a + 37
y = x + 42
x = a + 37
y = x + 42
```

```
x1 = a + 37
y1 = x1 + 42
x2 = a + 37
y2 = x2 + 42
```

```
a:  v1
v1 + 37:  v2 // a + 37
x1:  v2

v2 + 42: v3 //x1 + 42
y1: v3
```

值编号 Value Numbering

- 在SSA中，为每一个变量或表达式的结果分配一个值编号v_i

```
x = a + 37
y = x + 42
x = a + 37
y = x + 42
```

```
x1 = a + 37
y1 = x1 + 42
x2 = a + 37
y2 = x2 + 42
```

```
a:  v1
v1 + 37:  v2 // a + 37
x1:  v2

v2 + 42: v3 //x1 + 42
y1: v3

v1 + 37:  v2 // a + 37
x2:  v2
```

值编号 Value Numbering

- 在SSA中，为每一个变量或表达式的结果分配一个值编号v_i

```
x = a + 37
y = x + 42
x = a + 37
y = x + 42
```

```
x1 = a + 37
y1 = x1 + 42
x2 = a + 37
y2 = x2 + 42
```

```
a:  v1
v1 + 37:  v2 // a + 37
x1:  v2

v2 + 42: v3 //x1 + 42
y1: v3

v1 + 37:  v2 // a + 37
x2:  v2

v2 + 42:  v3 // x2 + 42
y2: v3
```

值编号 Value Numbering

- 在SSA中，为每一个变量或表达式的结果分配一个值编号v_i

```
x = a + 37
y = x + 42
x = a + 37
y = x + 42
```

```
x1 = a + 37
y1 = x1 + 42
x2 = a + 37
y2 = x2 + 42
```

```
a:  v1
v1 + 37:  v2 // a + 37
x1:  v2

v2 + 42: v3 //x1 + 42
y1: v3

v1 + 37:  v2 // a + 37
x2:  v2

v2 + 42:  v3 // x2 + 42
y2: v3
```

- 后面的值用前面的相同值编号的计算结果替代

值编号 Value Numbering

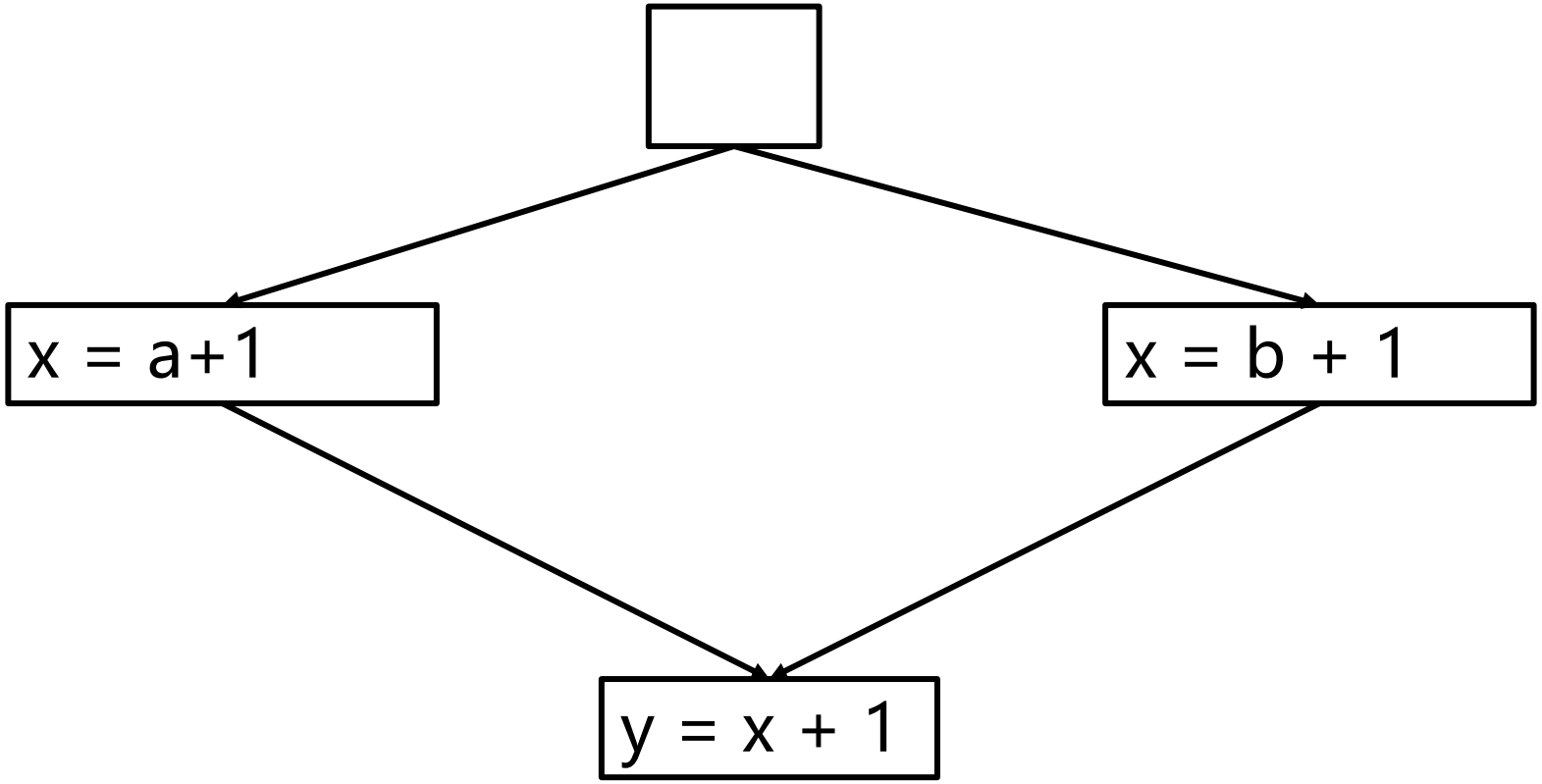
几个概念

- 每一个a；37；a + 37；x1 + 42都是表达式
- 每一个v1；v1 + 37；v2 + 42都是值表达式
- 每一个集合{v1，a}；{v2，x1，v1+37}；都是一个等价类 equivalence class，记为C
- 在程序点p，所有等价类的集合P={C1，C2，C3，…，Cn}叫做一个分区partition

	a：v1 v1 + 37：v2 // a + 37
x1 = a + 37	x1：v2
y1 = x1 + 42	v2 + 42: v3 //x1 + 42 y1: v3
x2 = a + 37	v1 + 37：v2 // a + 37 x2：v2
y2 = x2 + 42	v2 + 42：v3 // x2 + 42 y2: v3

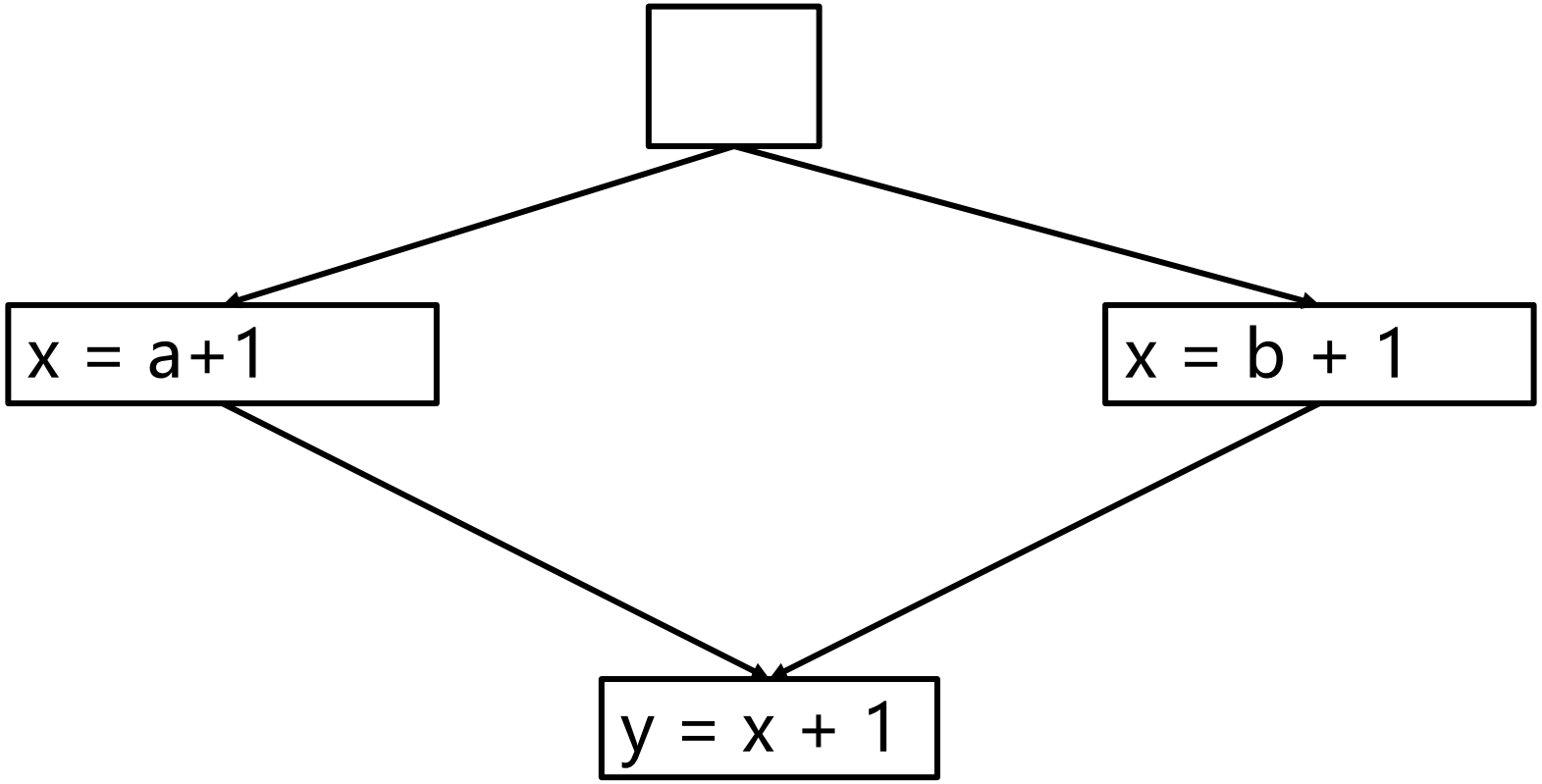
全局值编号 Global Value Numbering

- 把上述在基本块内部的value numbering扩展到全程序，跨基本块



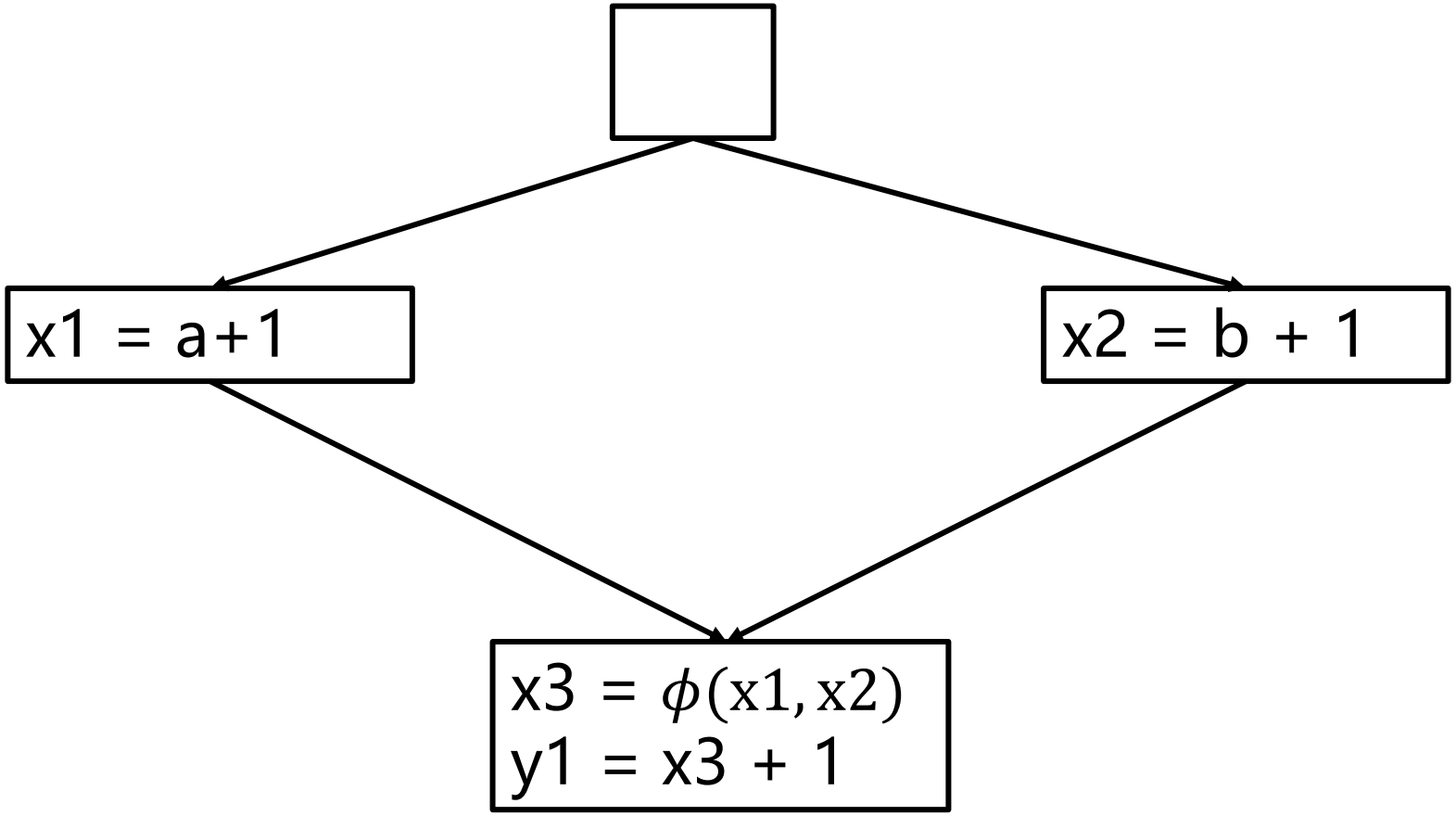
全局值编号 Global Value Numbering

- 难点在于如何处理多条路径的汇聚？也就是说一个基本块有多个前驱节点的时候，如何值编码？



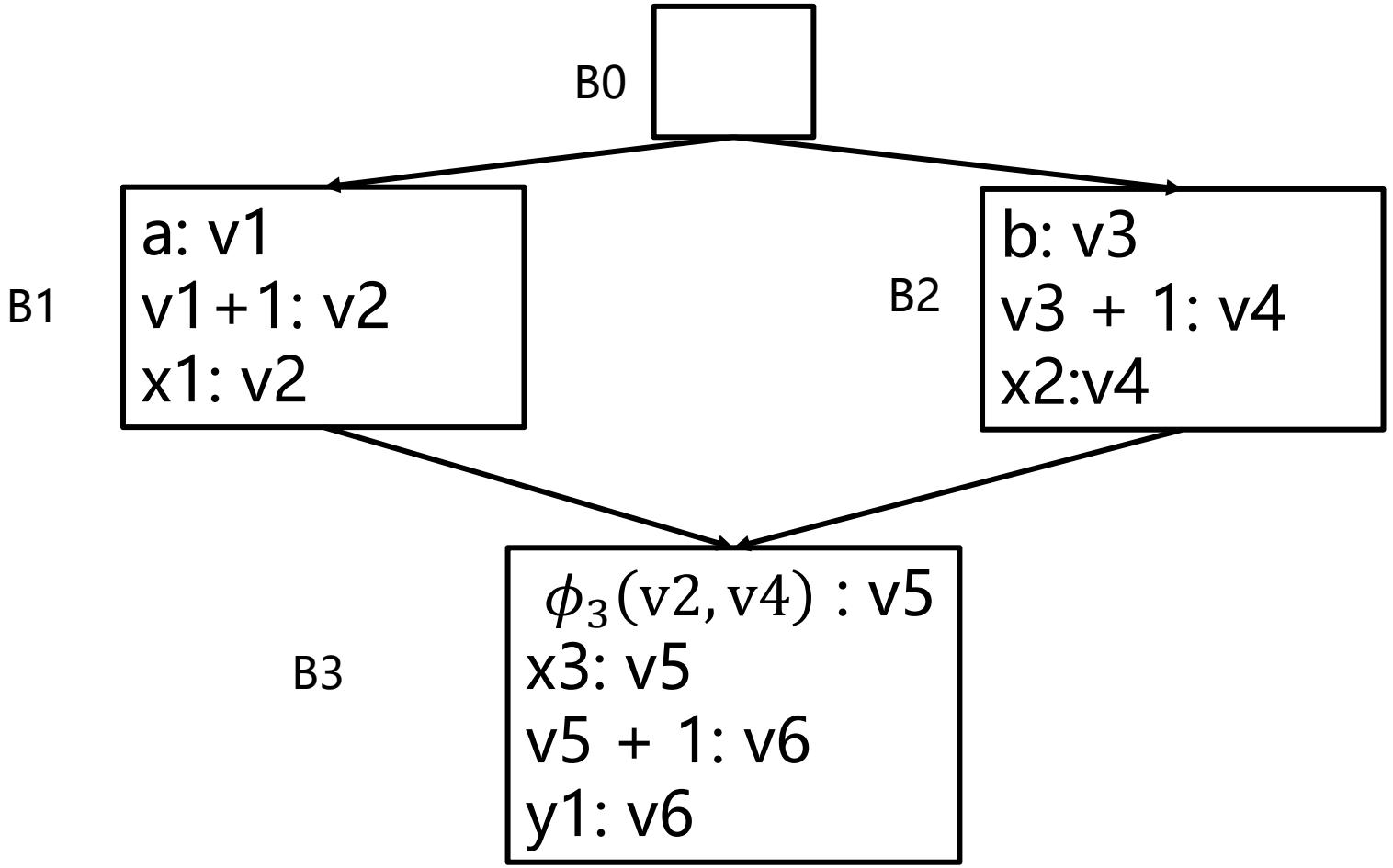
全局值编号 Global Value Numbering

- 难点在于如何处理多条路径的汇聚？也就是说一个基本块有多个前驱节点的时候，如何值编码？ -> 引入phi值表达式



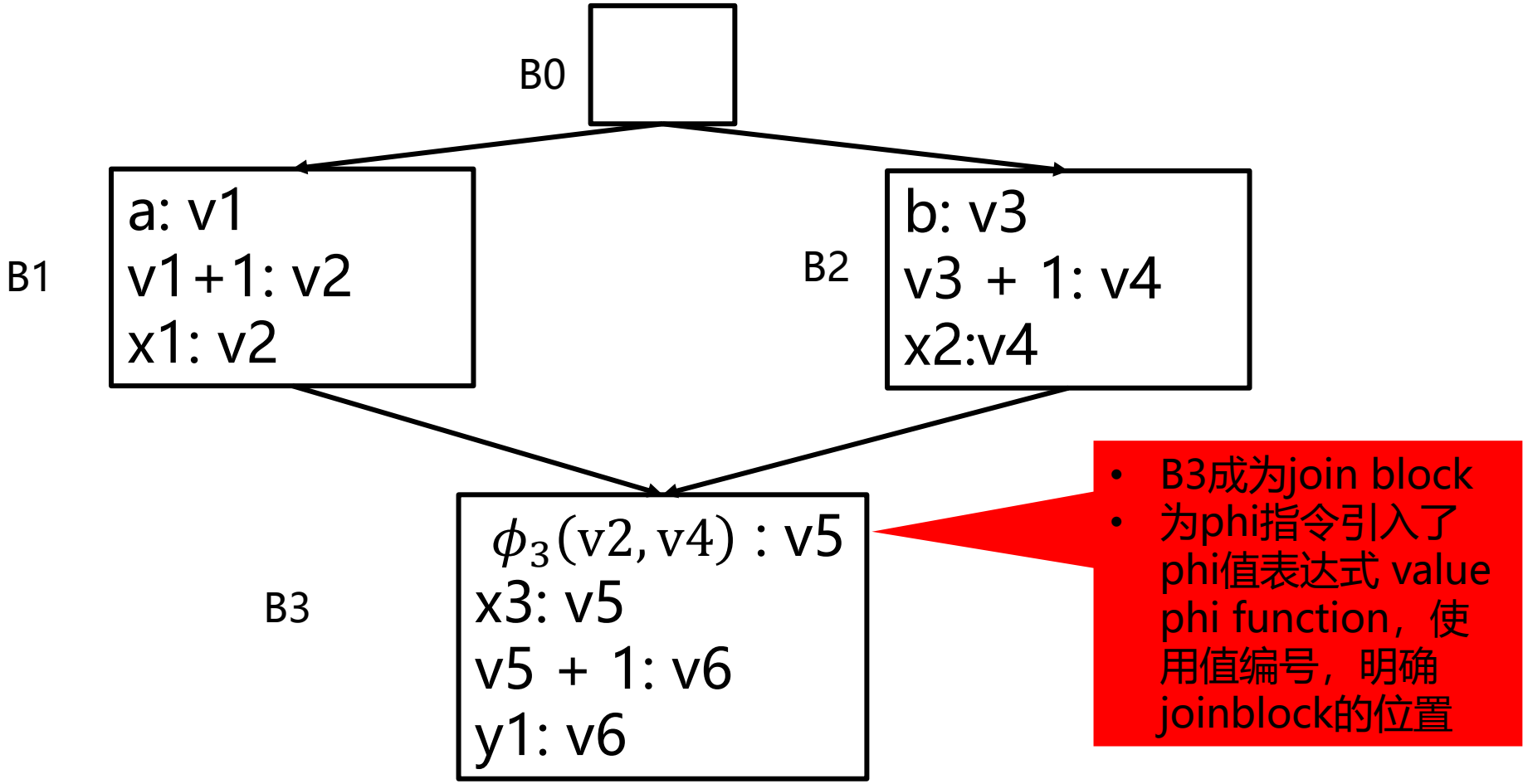
全局值编号 Global Value Numbering

- 难点在于如何处理多条路径的汇聚？也就是说一个基本块有多个前驱节点的时候，如何值编码？ -> 引入phi值表达式



全局值编号 Global Value Numbering

- 难点在于如何处理多条路径的汇聚？也就是说一个基本块有多个前驱节点的时候，如何值编码？ -> 引入phi值表达式



全局值编号 Global Value Numbering

- 核心任务：沿着控制流图control flow graph，计算在每一个程序点（或者说，基本块的开始和结尾处）的表达式等价类分区
- 这些等价类既包含普通的值表达式，有包含复杂的phi值表达式
- 数据流分析是关键！

数据流分析模式

- 数据流值代表在任一程序点能观测到的所有可能程序状态集合的一个**抽象**
- 对于一个语句*s*
  - *s* 之前的程序点对应的数据流值用 IN[*s*] 表示
  - s 之后的程序点对应的数据流值用 OUT[*s*] 表示
- 对于一个基本块呢?

数据流分析模式

传递函数(transfer function) f

- 语句前后两点的数据流值受该语句的语义约束
- 若沿执行路径正向传播，则$\mathrm{OUT}[s] = f_s(\mathrm{IN}[s])$
- 若沿执行路径逆向传播，则$\mathrm{IN}[s] = f_s(\mathrm{OUT}[s])$

若基本块B由语句$s_1, s_2, \ldots, s_n$依次组成，则

- $\mathrm{IN}[s_{i+1}] = \mathrm{OUT}[s_i]$，$i = 1, 2, \ldots, n-1$

考虑的是在语句执行后输入输出之间的变化关系

基本块上的数据流模式

- **IN[*B*]：紧靠基本块B之前的数据流值**
  - $IN[B] = IN[s_1]$
- **OUT[*B*]：紧靠基本块B之后的数据流值**
  - $OUT[B] = OUT[s_n]$
- **f_B:基本块B的传递函数**
  - 前向数据流： $OUT[B] = f_B(IN[B])$
    - $f_B = f_n \circ \ldots \circ f_2 \circ f_1$
  - 逆向数据流： $IN[B] = f_B(OUT[B])$
    - $f_B = f_1 \circ \ldots \circ f_{n-1} \circ f_n$

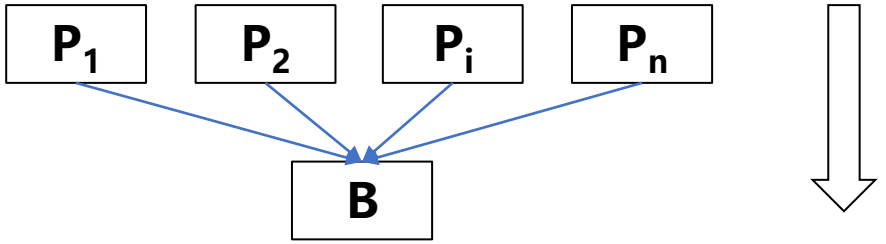
基本块间的数据流分析模式

控制流约束

- 正向传播

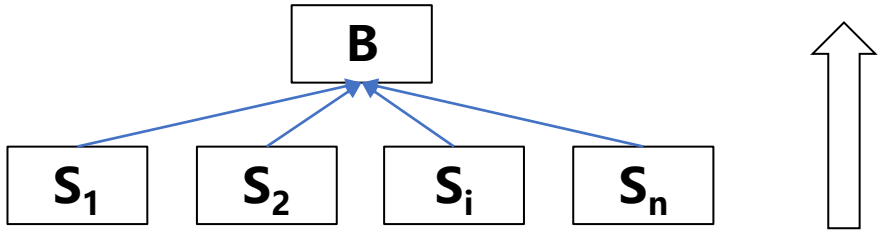
$$IN[B] = \cup_{P\text{是}B\text{的前驱}} OUT[P]$$

U 是汇合的意思，并不一定代表并集，也可能是交集等运算



- 逆向传播

$$OUT[B] = \cup_{S\text{是}B\text{的后继}} IN[S]$$



约束方程组的解通常不是唯一的

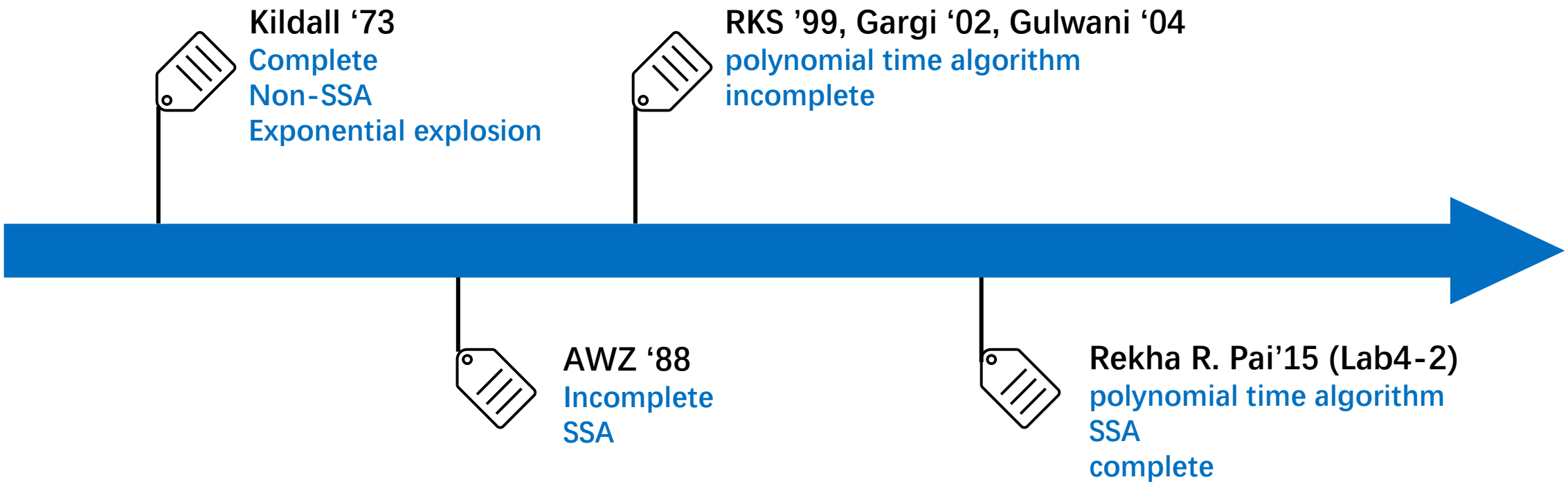
- 求解的目标是要找到满足这两组约束（控制流约束和迁移约束）的最“精确”解

考虑的是在其他语句或块对于输入的影响和本次执行的输出对其他语句和块的影响

GVN的数据流分析方法

- **正向传播**，从前驱向后继结点传播信息，先算IN，再算OUT
- 数据流值，即状态
  - **IN/OUT**：分别对应基本块入口处和出口处的等价类分区
- **传递函数f**
  - 对于每一条指令s，OUT[s] = f(IN[s])
- 约束方程组
  - IN[B] = $\cup_{P\text{是}B\text{的前驱}}$OUT[*P*] //这里需要考虑一个**join operator**
  - OUT[*B*] = f_B(IN[*B*]) //考虑到基本块内若干指令的f串起来

GVN 算法发展



参考：Detection of Redundant Expressions: A Complete and Polynomial-Time Algorithm in SSA

基本概念

程序及IR假设

- 假设程序是SSA IR格式
- 流图中有entry和exit基本块，均为空
- 每个基本块至多有两个前驱结点

join block

- 有两个前驱的基本块

基本概念

- **表达式expression，有如下形态**
  - 常量
  - 变量
  - $x \oplus y$，x, y是常量或变量， $\oplus$是二元运算符
  - $\phi(\mathrm{x1}, \mathrm{x2})$

基本概念

- **表达式等价 Herbrand equivalence [1]**
  - 如果两个表达式的运算符相同，且操作数也是Herbrand等价的，那么他们是Herbrand等价的。
  - 仅考虑结构等价，递归判断
- **对phi表达式的等价判断需要考虑每条路径等价，因此，需要对phi指令进行额外处理**

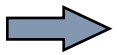
[1] source: The Value Flow Graph - A Program Representation for Optimal Program Transformations

基本概念

- 对phi指令的特殊处理：数据流分析中，phi 指令视作为 join block 的前驱的copy指令进行处理

例如：

```
bb1:
        x1 = 1 + 1
        br bb3
bb2:
        x2 = 2 + 2
        br bb3
bb3:
        x3=phi(x1,x2)
        ...
```



```
bb1:
        x1 = 1 + 1
        x3 = x1
        br bb3
bb2:
        x2 = 2 + 2
        x3 = x2
        br bb3
bb3:
        ...
```

基本概念

- **值表达式value expression，有如下形态**
  - $v_i \oplus v_j = \{x \oplus y \mid x \in Ci,\ y \in Cj,$ Ci和Cj是两个等价类，分别对应值编号vi和vj} //简写为ve
  - $\phi_k(v_i, v_j)$ // value phi function，简写为vpf
- **一个值表达式对应具有相同值编码的一组表达式的集合**

基本概念

等价类equivalence class

- vr, x1, y1
- vs, z1, vr +1
- vm, xn :ϕk(vi, vj)
  - 其中vr、vs、vm为值编号;
  - x1、x2、y1、xn为变量;
  - vr +1为值表达式，ϕk(vi, vj)为phi值表达式

分区partition

- $P = \{\cdots, \{vr, x1, y1\}, \{vs, z1, vr+1\}, \{vm, xn:\phi_k(vi, vj)\}, \cdots\}$

GVN算法过程

- Step1：通过前向数据流分析，算出每个基本块的出口数据流值（即Partition）
- Step2：通过基本块出口Partition，对基本块内冗余计算指令进行替换

数据流分析伪代码

```
detectEquivalences(G){
    PIN[B1] = ∅ // “B1” is the first block
    POUT[B1] = transferFunction(PIN[B1])
    foreach block do
        POUT[B] = ⊤ // 特殊符号Top
    while any POUT changes
        foreach block B do
            if B has two predecessors then
                PIN[B] = join(POUT[P1], POUT[P2])
            else
                PIN[B] = POUT[P]
            POUT[B] = transferFunction(PIN[B])
}
```

数据流分析伪代码

```
detectEquivalences(G){
    PIN[B1] = ∅ // “B1” is
    POUT[B1] = transferFunc
    foreach block do
        POUT[B] = ⊤ // 特殊符号Top
    while any POUT changes
        foreach block B do
            if B has two predecessors then
                PIN[B] = join(POUT[P1], POUT[P2])
            else
                PIN[B] = POUT[P]
            POUT[B] = transferFunction(PIN[B])
}
```

detectEquivalences是数据流分析的主体过程，通过多次迭代获得稳定的Partition，G为IR控制流图

数据流分析伪代码

GVN采用前向数据流分析，此处对入口块Partition初始化

```
detectEquivalences(G){
    PIN[B1] = ∅ // “B1” is the first block
    POUT[B1] = transferFunction(PIN[B1])
    foreach block do
        POUT[B] = ⊤ // 特殊符号Top
    while any POUT changes
        foreach block B do
            if B has two predecessors then
                PIN[B] = join(POUT[P1], POUT[P2])
            else
                PIN[B] = POUT[P]
            POUT[B] = transferFunction(PIN[B])
}
```

数据流分析伪代码

```
detectEquivalences(G){
    PIN[B1] = ∅ // “B1” is the first block
    POUT[B1] = transferFunction(PIN[B1])
    foreach block do
        POUT[B] = ⊤ // 特殊符号Top
    while any POUT changes
        foreach block B do
            if B has two predecessors then
                PIN[B] = join(POUT[P1], POUT[P2])
            else
                PIN[B] = POUT[P]
            POUT[B] = transferFunction(PIN[B])
}
```

通过入口Partition ，以及转移函数（transferFunction）计算出B1的出口Partition

数据流分析伪代码

```
detectEquivalences(G){
    PIN[B1] = ∅ // “B1” is the first block
    POUT[B1] = transferFunction(PIN[B1])
    foreach block do
        POUT[B] = ⊤ // 特殊符号Top
    while any POUT changes
        foreach block B do
            if B has two predecessors then
                PIN[B] = join(POUT[P1], POUT[P2])
            else
                PIN[B] = POUT[P]
            POUT[B] = transferFunction(PIN[B])
}
```

对每个基本块出口Partition初始化为Top class（定义Top class：P1 = Join(P1,Top)，for any P1）

数据流分析伪代码

```
detectEquivalences(G){
    PIN[B1] = ∅ // “B1” is the first block
    POUT[B1] = transferFunction(PIN[B1])
    foreach block do
        POUT[B] = ⊤ // 特殊符号t
    while any POUT changes
        foreach block B do
            if B has two predecessors then
                PIN[B] = join(POUT[P1], POUT[P2])
            else
                PIN[B] = POUT[P]
            POUT[B] = transferFunction(PIN[B])
}
```

迭代主体，判断是否有基本块的 POUT 发生了修改

数据流分析伪代码

```
detectEquivalences(G){
    PIN[B1] = ∅ // “B1” is the first block
    POUT[B1] = transferFunction(PIN[B1])
    foreach block do
        POUT[B] = ⊤ // 特殊符号top
    while any POUT changes
        foreach block B do
            if B has two predecessors then
                PIN[B] = join(POUT[P1], POUT[P2])
            else
                PIN[B] = POUT[P]
            POUT[B] = transferFunction(PIN[B])
}
```

迭代过程中，如果一条语句有两个前驱，使用Join函数获得其入口Partition

数据流分析伪代码

```
detectEquivalences(G){
    PIN[B1] = ∅ // “B1” is the first block
    POUT[B1] = transferFunction(PIN[B1])
    foreach block do
        POUT[B] = ⊤ // 特殊符号top
    while any POUT changes
        foreach block B do
            if B has two predecessors then
                PIN[B] = join(POUT[P1], POUT[P2])
            else
                PIN[B] = POUT[P]
            POUT[B] = transferFunction(PIN[B])
}
```

使用转移函数（transferFunction）计算当前基本块出口Partition

数据流分析伪代码

转移函数

```
transferFunction(x = e, PIN[s]){
    POUT[s] = PIN[s]
    if x in a class Ci of POUT[s] then
        Ci = Ci - {x}
    ve = valueExpr(e) // 计算/找到值表达式
    vpf = valuePhiFunc(ve, PIN[s]) // 找到值phi表达式
    if ve or vpf in a class Ci of POUT[s] then
        Ci = Ci ∪ {x, ve} // 扩充Ci
    else
        // vn是新值编号，添加新的等价类
        POUT[s] = POUT[s] ∪ {vn, x, ve : vpf }
    return POUT[s]
}
```

此处给出的是对于一条指令的转移函数（transferFunction）x = e，x 代表变量，e代表表达式，PIN[s]代表 x = e 指令的入口Partition；对基本块的转移函数即为块内一连串指令的转移函数的计算结果。

数据流分析伪代码

转移函数

初始化 x = e 的出口Partition

```
transferFunction(x = e, PIN[s]){
    POUT[s] = PIN[s]
    if x in a class Ci of POUT[s] then
        Ci = Ci - {x}
    ve = valueExpr(e) // 计算/找到值表达式
    vpf = valuePhiFunc(ve, PIN[s]) // 找到值phi表达式
    if ve or vpf in a class Ci of POUT[s] then
        Ci = Ci ∪ {x, ve} // 扩充Ci
    else
        // vn是新值编号，添加新的等价类
        POUT[s] = POUT[s] ∪ {vn, x, ve : vpf }
    return POUT[s]
}
```

数据流分析伪代码

转移函数

```
transferFunction(x = e, PIN[s]){
    POUT[s] = PIN[s]
    if x in a class Ci of POUT[s] then
        Ci = Ci - {x}
    ve = valueExpr(e) // 计算/找到
    vpf = valuePhiFunc(ve, PIN[s]) // 找到值phi表达式
    if ve or vpf in a class Ci of POUT[s] then
        Ci = Ci ∪ {x, ve} // 扩充Ci
    else
        // vn是新值编号，添加新的等价类
        POUT[s] = POUT[s] ∪ {vn, x, ve : vpf }
    return POUT[s]
}
```

将x从出口Partition中的等价类中删去，因为x被e重新赋值，需要根据e来重新划分x到某个等价类中

数据流分析伪代码

转移函数

```
transferFunction(x = e, PIN[s]){
    POUT[s] = PIN[s]
    if x in a class Ci of POUT[s] then
        Ci = Ci - {x}
    ve = valueExpr(e) // 计算/找到值表达式
    vpf = valuePhiFunc(ve, PIN[s]) // 找到值phi表达式
    if ve or vpf in a class Ci of POUT[s] then
        Ci = Ci ∪ {x, ve} // 扩充Ci
    else
        // vn是新值编号，添加新的等价类
        POUT[s] = POUT[s] ∪ {vn, x, ve : vpf }
    return POUT[s]
}
```

通过e找到对应的值表达式

数据流分析伪代码

转移函数

```
transferFunction(x = e, PIN[s]){
    POUT[s] = PIN[s]
    if x in a class Ci of POUT[s] then
        Ci = Ci - {x}
    ve = valueExpr(e) // 计算/找到值表达式
    vpf = valuePhiFunc(ve, PIN[s]) // 找到值phi表达式
    if ve or vpf in a class Ci of POUT[s] then
        Ci = Ci ∪ {x, ve} // 扩充Ci
    else
        // vn是新值编号，添加新的等价类
        POUT[s] = POUT[s] ∪ {vn, x, ve : vpf }
    return POUT[s]
}
```

通过ve与PIN[s]，创建相应phi值表达式（可能为空），即该表达式可能可以转化为某个等价的phi函数，属于某个phi值表达式

数据流分析伪代码

转移函数

```
transferFunction(x = e, PIN[s]){
    POUT[s] = PIN[s]
    if x in a class Ci of POUT[s] then
        Ci = Ci - {x}
    ve = valueExpr(e) // 计算/找到值表达式
    vpf = valuePhiFunc(ve, PIN[s]) // 找到值phi表达式
    if ve or vpf in a class Ci of POUT[s] then
        Ci = Ci ∪ {x, ve} // 扩充Ci
    else
        // vn是新值编号，添加新的等价类
        POUT[s] = POUT[s] ∪ {vn, x, ve : vpf }
    return POUT[s]
}
```

若ve或者vpf存在于POUT[s]中的某个等价类Ci中，则将x加入对应等价类中

数据流分析伪代码

转移函数

```
transferFunction(x = e, PIN[s]){
    POUT[s] = PIN[s]
    if x in a class Ci of POUT[s] then
        Ci = Ci - {x}
    ve = valueExpr(e) // 计算/找到值表达式
    vpf = valuePhiFunc(ve, PIN[s]) // 找到值phi表达式
    if ve or vpf in a class Ci of POUT[s] then
        Ci = Ci ∪ {x, ve} // 扩充Ci
    else
        // vn是新值编号，添加新的等价类
        POUT[s] = POUT[s] ∪ {vn, x, ve : vpf }
    return POUT[s]
}
```

若不存在，则创建新的等价类，vn为新等价类的值编号

数据流分析伪代码

- valuePhiFunc：判断当前ve是否等价某个phi值表达式

```
valuePhiFunc(ve, PIN[s]){
    if ve is of the form ϕk(vi1, vj1) ⊕ ϕk(vi2, vj2) then
        vi = getVN(POUT[kl], vi1 ⊕ vi2) //左前驱
        if (vi = = NULL) then
            vi = valuePhiFunc(vi1 ⊕ vi2, POUT[kl])
        vj = getVN(POUT[kr], vj1 ⊕ vj2) //右前驱
        if (vj == NULL) then
            vj = valuePhiFunc(vj1 ⊕ vj2, POUT[kr])

    if vi, vj are non-NULL then
        return  ϕk(vi, vj) // vi, vj are non-NULL
    else
        return NULL
}
```

数据流分析伪代码

- valuePhiFunc：判断当前v

valuePhiFunc 是用来判断ve是否是phi值表达式，输入项ve是值表达式，PIN[s]是语句s的入口Partition

```
valuePhiFunc(ve, PIN[s]){
    if ve is of the form ϕk(vi1, vj1) ⊕ ϕk(vi2, vj2) then
        vi = getVN(POUT[kl], vi1 ⊕ vi2) //左前驱
        if (vi = = NULL) then
            vi = valuePhiFunc(vi1 ⊕ vi2, POUT[kl])
        vj = getVN(POUT[kr], vj1 ⊕ vj2) //右前驱
        if (vj == NULL) then
            vj = valuePhiFunc(vj1 ⊕ vj2, POUT[kr])

    if vi, vj are non-NULL then
        return  ϕk(vi, vj) // vi, vj are non-NULL
    else
        return NULL
}
```

数据流分析伪代码

valuePhiFunc：判断当前ve是否等价某个phi值表达式

```
valuePhiFunc(ve, PIN[s]){
    if ve is of the form ϕk(vi1, vj1) ⊕ ϕk(vi2, vj2) then
        vi = getVN(POUT[kl], vi1 ⊕ vi2) //左前驱
        if (vi = = NULL) then
            vi = valuePhiFunc(vi1 ⊕ vi2, POUT[
        vj = getVN(POUT[kr], vj1 ⊕ vj2) //右前驱
        if (vj == NULL) then
            vj = valuePhiFunc(vj1 ⊕ vj2, POUT[kr])

    if vi, vj are non-NULL then
        return  ϕk(vi, vj) // vi, vj are non-NULL
    else
        return NULL
}
```

此处判断ve是否可以表示成两条路径值表达式的汇聚，为了简化理解，此处仅选取了$\phi_k(v_{i1}, v_{j1}) \oplus \phi_k(v_{i2}, v_{j2})$一种情况进行判断

数据流分析伪代码

valuePhiFunc：判断当前ve是否等价某个phi值表达式

```
valuePhiFunc(ve, PIN[s]){
    if ve is of the form ϕk(vi1, vj1) ⊕ ϕk(vi2, vj2)
        vi = getVN(POUT[kl], vi1 ⊕ vi2) //左前驱
        if (vi = = NULL) then
            vi = valuePhiFunc(vi1 ⊕ vi2, POUT[kl])
        vj = getVN(POUT[kr], vj1 ⊕ vj2) //右前驱
        if (vj == NULL) then
            vj = valuePhiFunc(vj1 ⊕ vj2, POUT[kr])

    if vi, vj are non-NULL then
        return  ϕk(vi, vj) // vi, vj are non-NULL
    else
        return NULL
}
```

判断形如$v_{i1} \oplus v_{i2}$的表达式在当前基本块的左前驱中是否存在相应的值表达式，kl表示该基本块的左前驱

判断形如$v_{j1} \oplus v_{j2}$的表达式在当前基本块的右前驱中是否存在相应的值表达式，kr表示该基本块的右前驱

数据流分析伪代码

valuePhiFunc：判断当前ve是否等价某个phi值表达式

```
valuePhiFunc(ve, PIN[s]){
    if ve is of the form ϕk(vi1, vj1) ⊕ ϕk(vi2, vj2) then
        vi = getVN(POUT[kl], vi1 ⊕ vi2) //左前驱
        if (vi = = NULL) then
            vi = valuePhiFunc(vi1 ⊕ vi2, POUT[kl])
        vj = getVN(POUT[kr], vj1 ⊕ vj2) //右前驱
        if (vj == NULL) then
            vj = valuePhiFunc(vj1 ⊕ vj2, POUT[kr])

    if vi, vj are non-NULL then
        return  ϕk(vi, vj) // vi, vj are non-NULL
    else
        return NULL
}
```

若前驱中不存在形如$v_{i1} \oplus v_{i2}$的值表达式，则递归调用valuePhiFunc判断$v_{i1} \oplus v_{i2}$在前驱是否是Phi值表达式

对于$v_{j1} \oplus v_{j2}$以及右前驱同理

数据流分析伪代码

valuePhiFunc：判断当前ve是否等价某个phi值表达式

```
valuePhiFunc(ve, PIN[s]){
    if ve is of the form ϕk(vi1, vj1) ⊕ ϕk(vi2, vj2) then
        vi = getVN(POUT[kl], vi1 ⊕ vi2) //左前驱
        if (vi = = NULL) then
            vi = valuePhiFunc(vi1 ⊕ vi2, POUT[kl])
        vj = getVN(POUT[kr], vj1 ⊕ vj2) //右前驱
        if (vj == NULL) then
            vj = valuePhiFunc(vj1 ⊕ vj2, POUT[kr])

    if vi, vj are non-NULL then
        return  ϕk(vi, vj) // vi, vj are non-NULL
    else
        return NULL
}
```

当前vi, vj都不为空，则该ve等价于某个phi值表达式，用该phi值表达式表示ve

数据流分析伪代码

join函数

join操作用于处理有两个前驱的基本块的Partition汇聚问题，其中P1，P2代表前驱基本块的出口Partition

```
join(P1, P2){
    P = {}
    foreach pair of classes Ci ∈ P1 and Cj ∈ P2
        Ck = Intersect(Ci, Cj)
        P = P ∪ Ck
    return P  // Ignore when Ck is empty
}
```

数据流分析伪代码

join函数

```
join(P1, P2){
    P = {}
    foreach pair of classes Ci ∈ P1 and Cj ∈ P2
        Ck = Intersect(Ci, Cj)
        P = P ∪ Ck
    return P  // Ignore when Ck is
}
```

在处理两个Partition的Join操作时，对每一对等价类做求交集操作，并将交集结果加入Join输出Partition

数据流分析伪代码

Intersect函数:

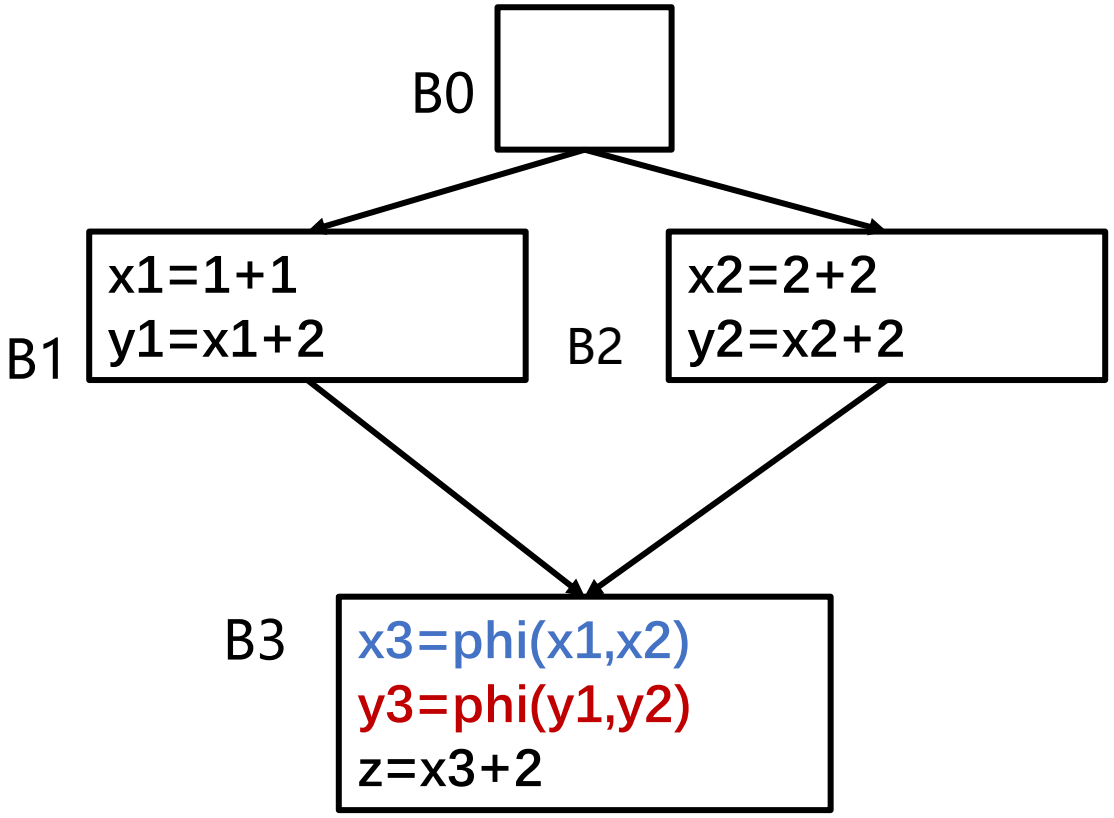
- 对每一对等价类做求交集操作并在必要的情况下赋予新的值编号

```
Intersect(Ci, Cj){
    Ck = Ci ∩ Cj // set intersection
    if Ck ≠ ∅ and Ck does not have value number then
        Ck = Ck ∪ {vk} // vk is new value number
        Ck = (Ck - {vpf}) ∪ {ϕb(vi, vj)}
        // vpf is value ϕ-function in Ck, vi ∈ Ci,
        // b is join block
    return Ck
}
```

对于v_i, v_j非空的求交集结果，添加新的值编号vk，以及添加phi值表达式：$\phi_b(v_i, v_j)$

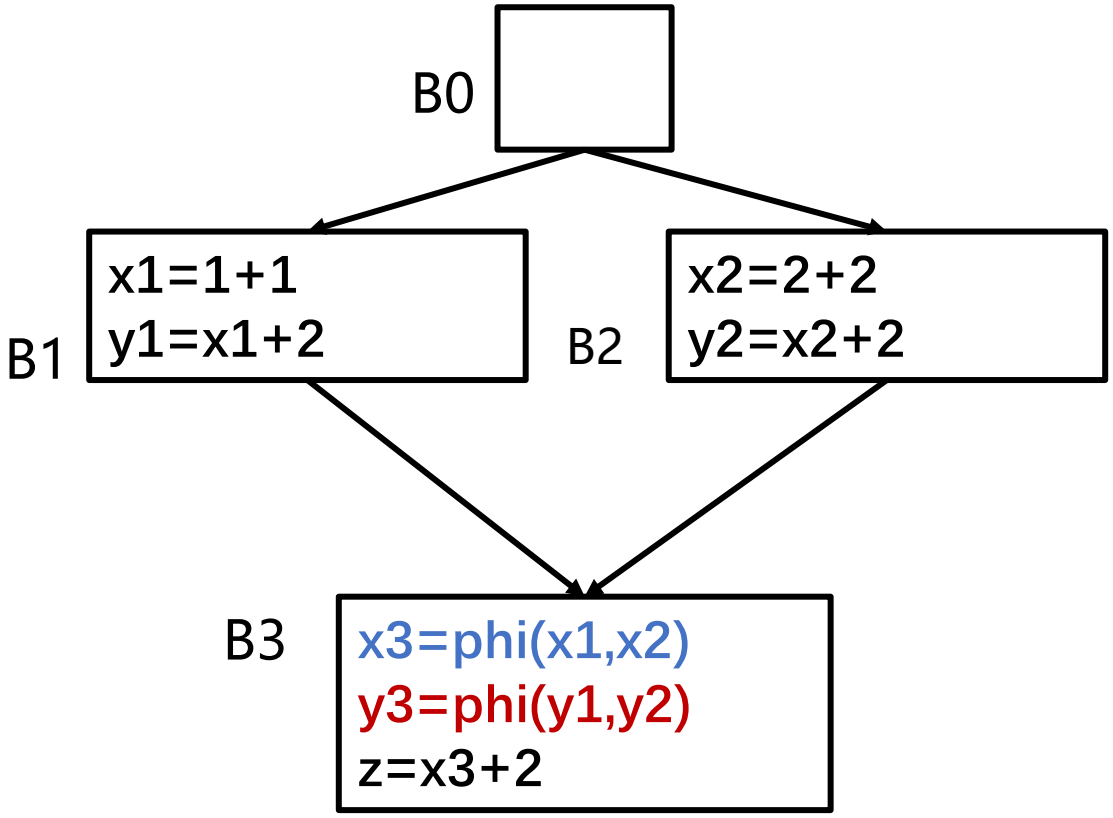
GVN数据流分析举例演示

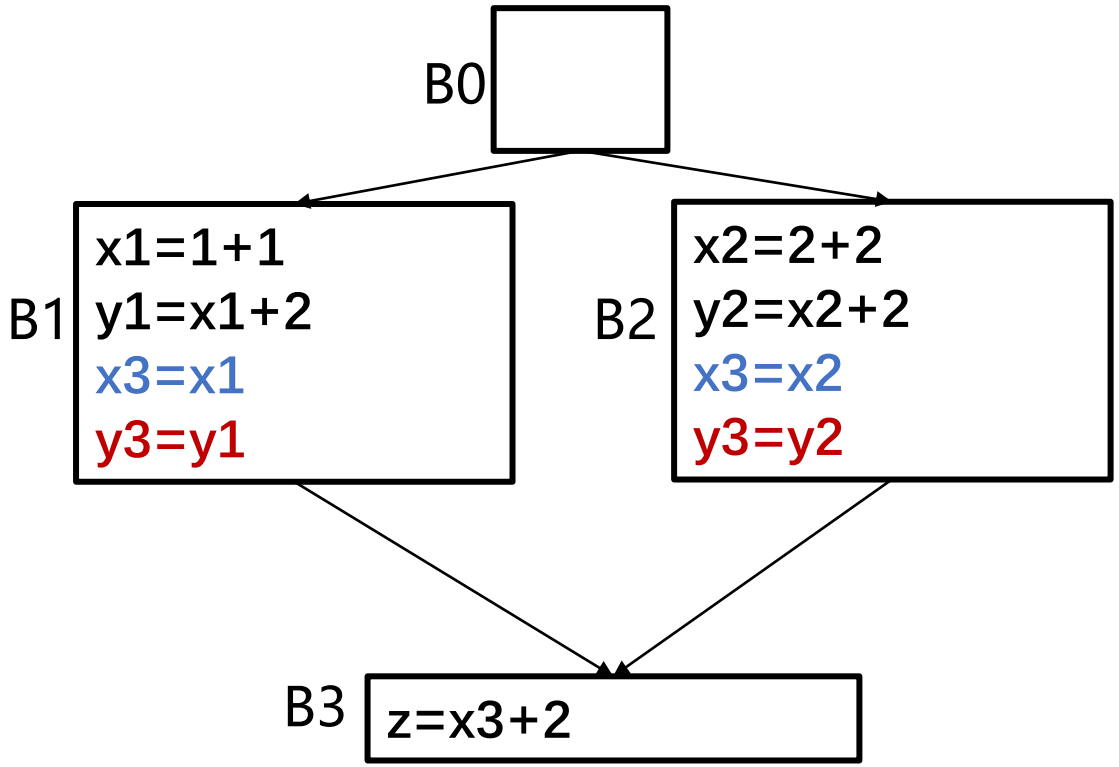
- 借助phi指令向copy语句的转换，在数据流分析中，我们可以在下例中应用



GVN数据流分析举例演示

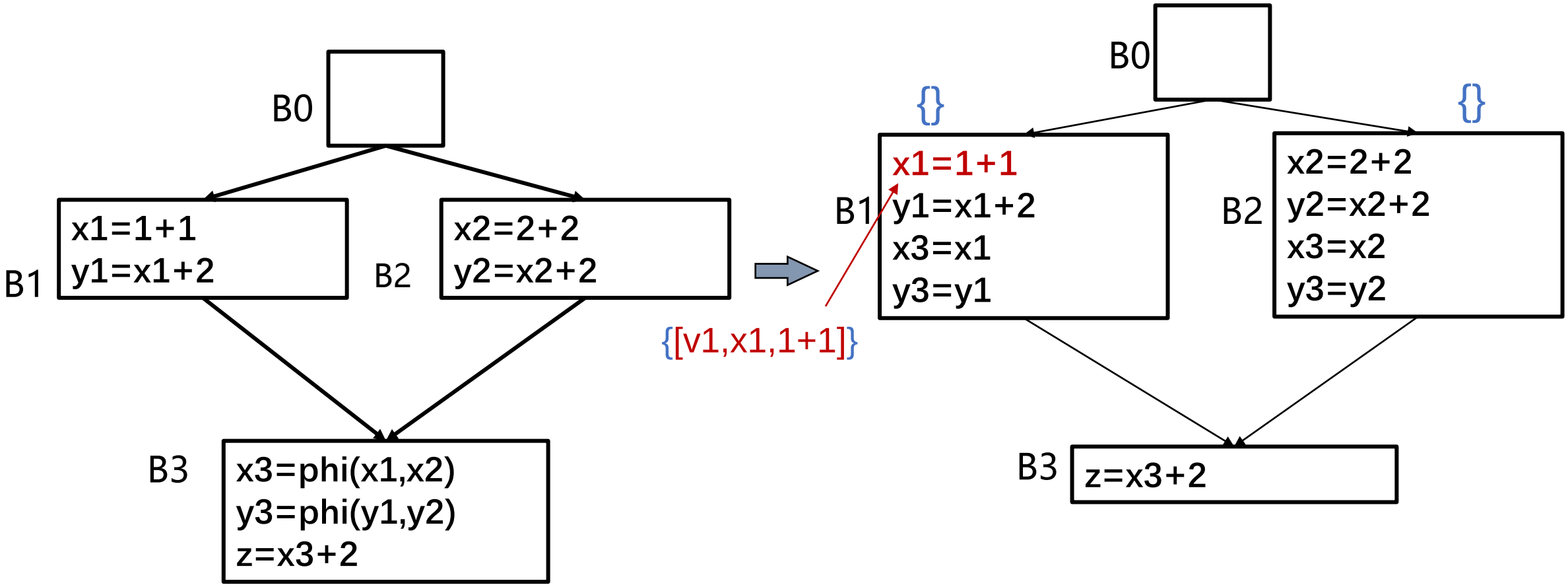
- 借助phi指令向copy语句的转换，在数据流分析中，我们可以在下例中应用





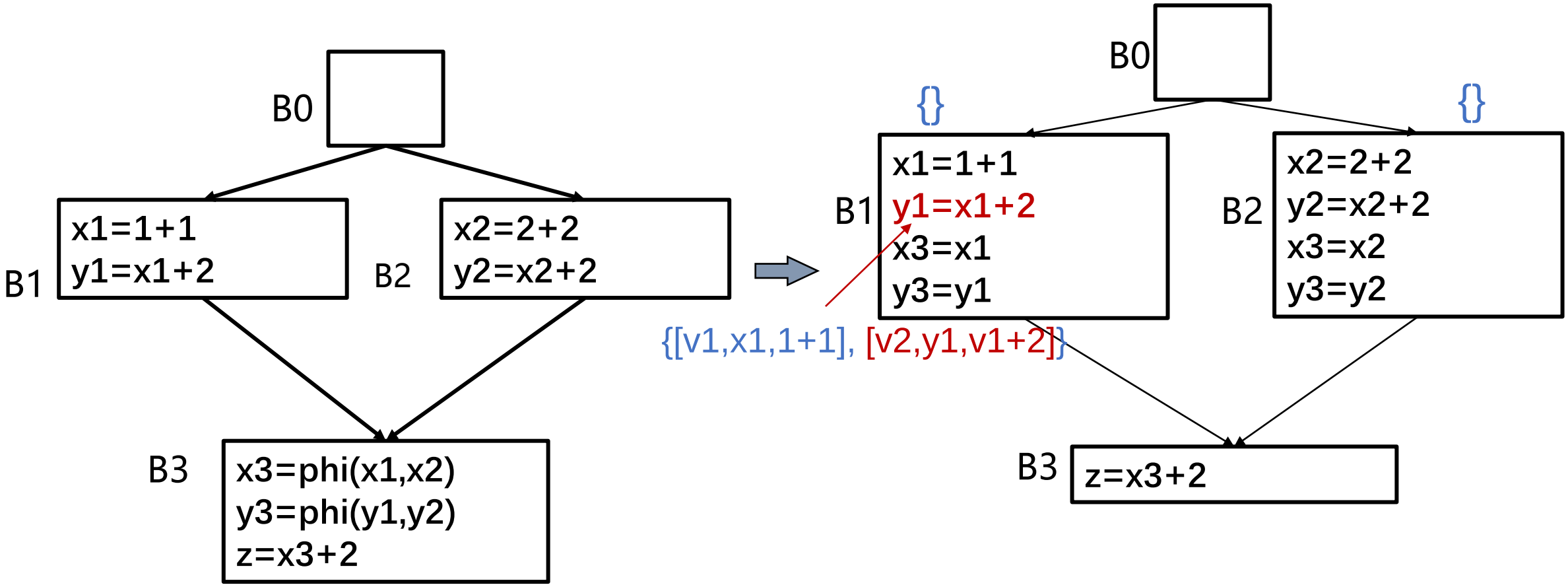
GVN数据流分析举例演示

- 对B1内每条语句执行转移函数（transferFunction）



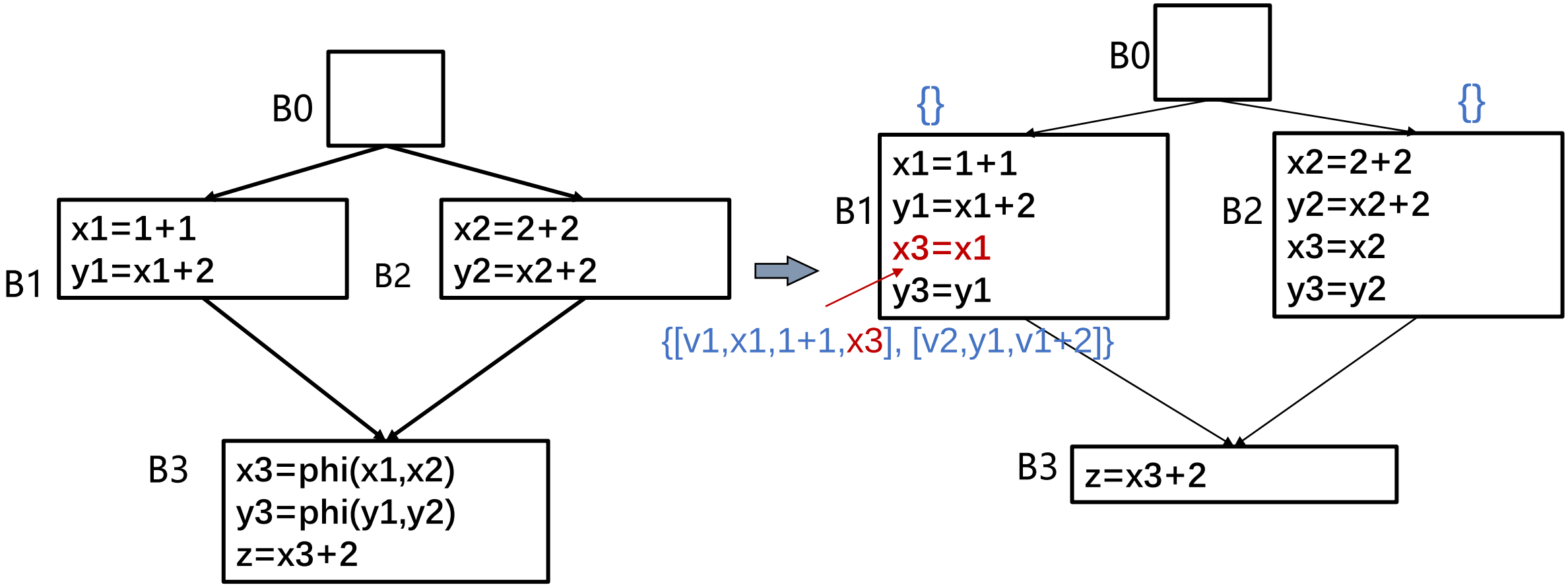
GVN数据流分析举例演示

- 对B1内每条语句执行转移函数（transferFunction）



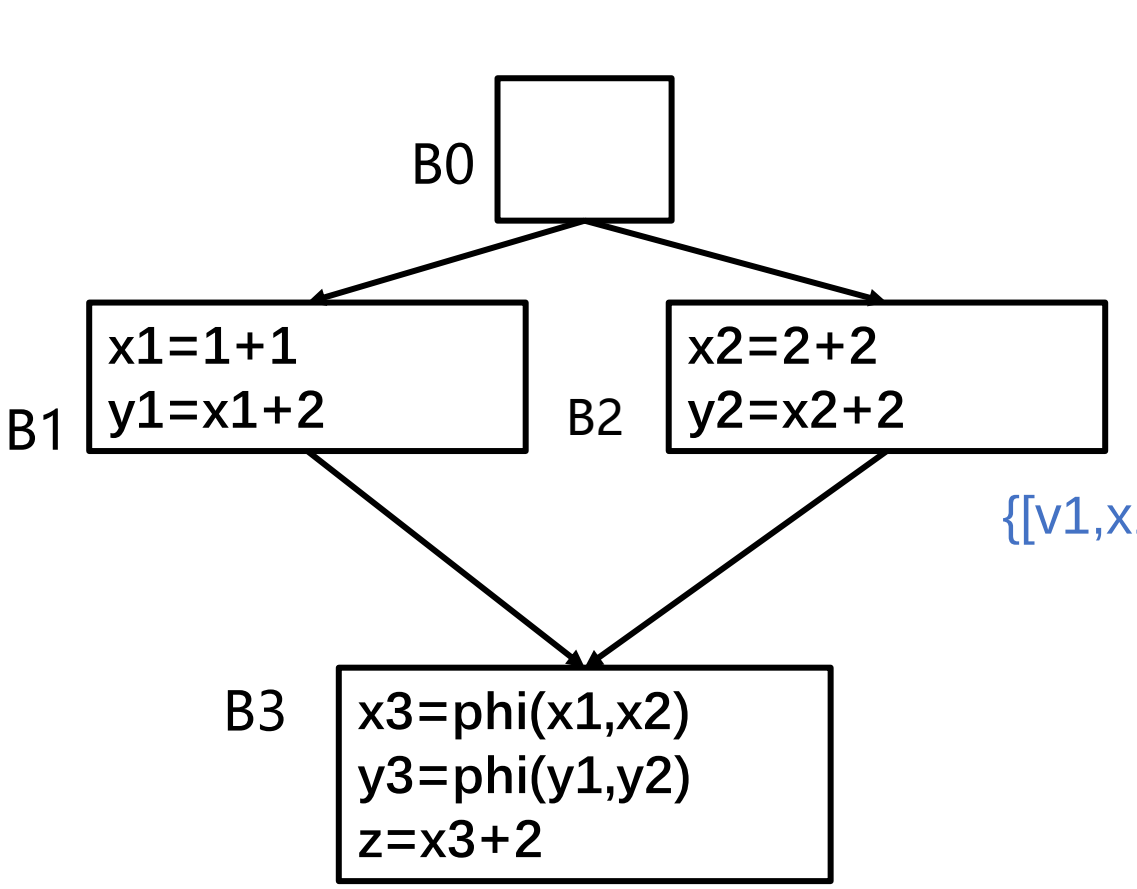
GVN数据流分析举例演示

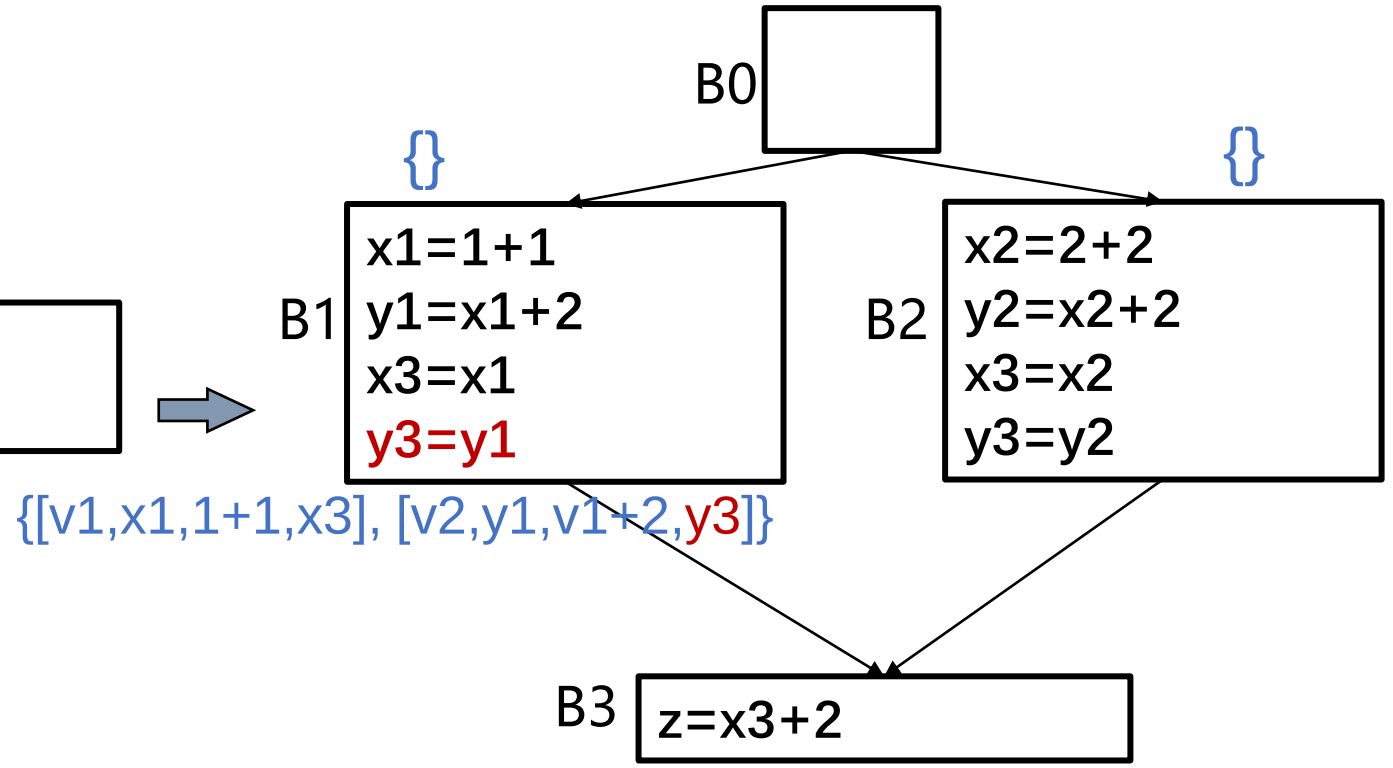
- 对B1内每条语句执行转移函数（transferFunction）



GVN数据流分析举例演示

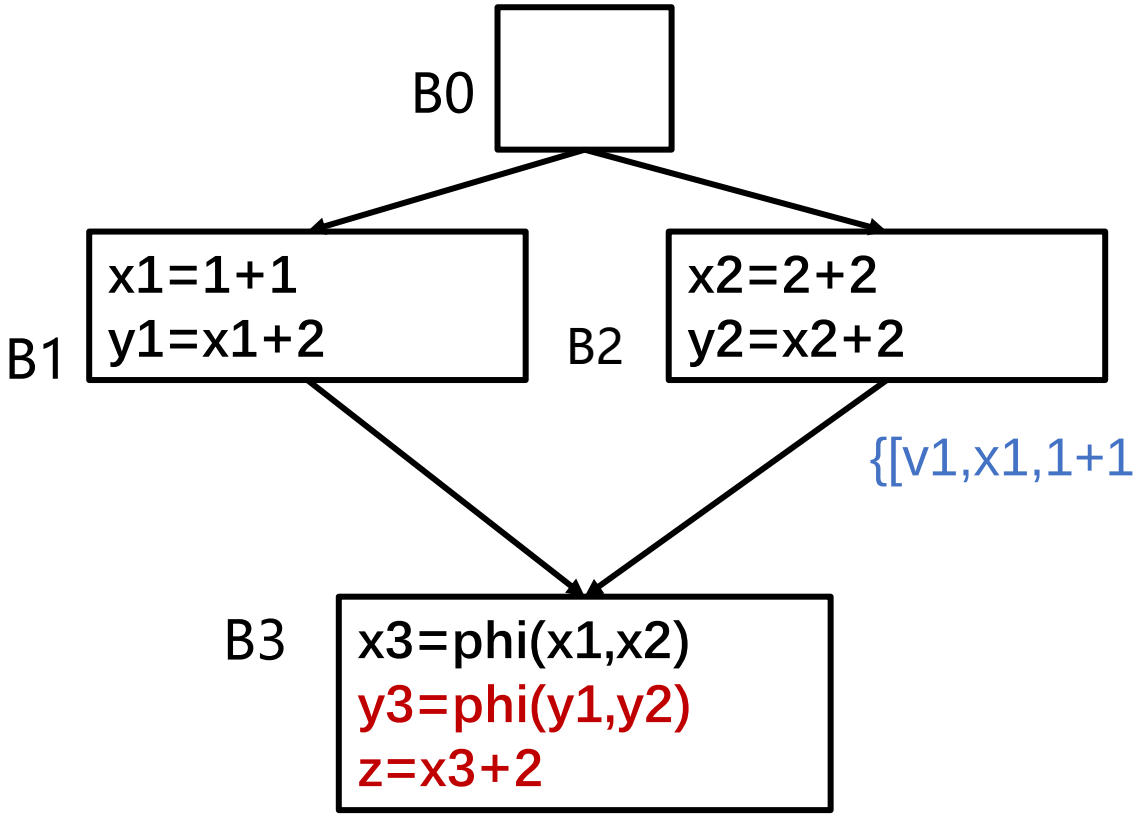
- 对B1内每条语句执行转移函数（transferFunction）

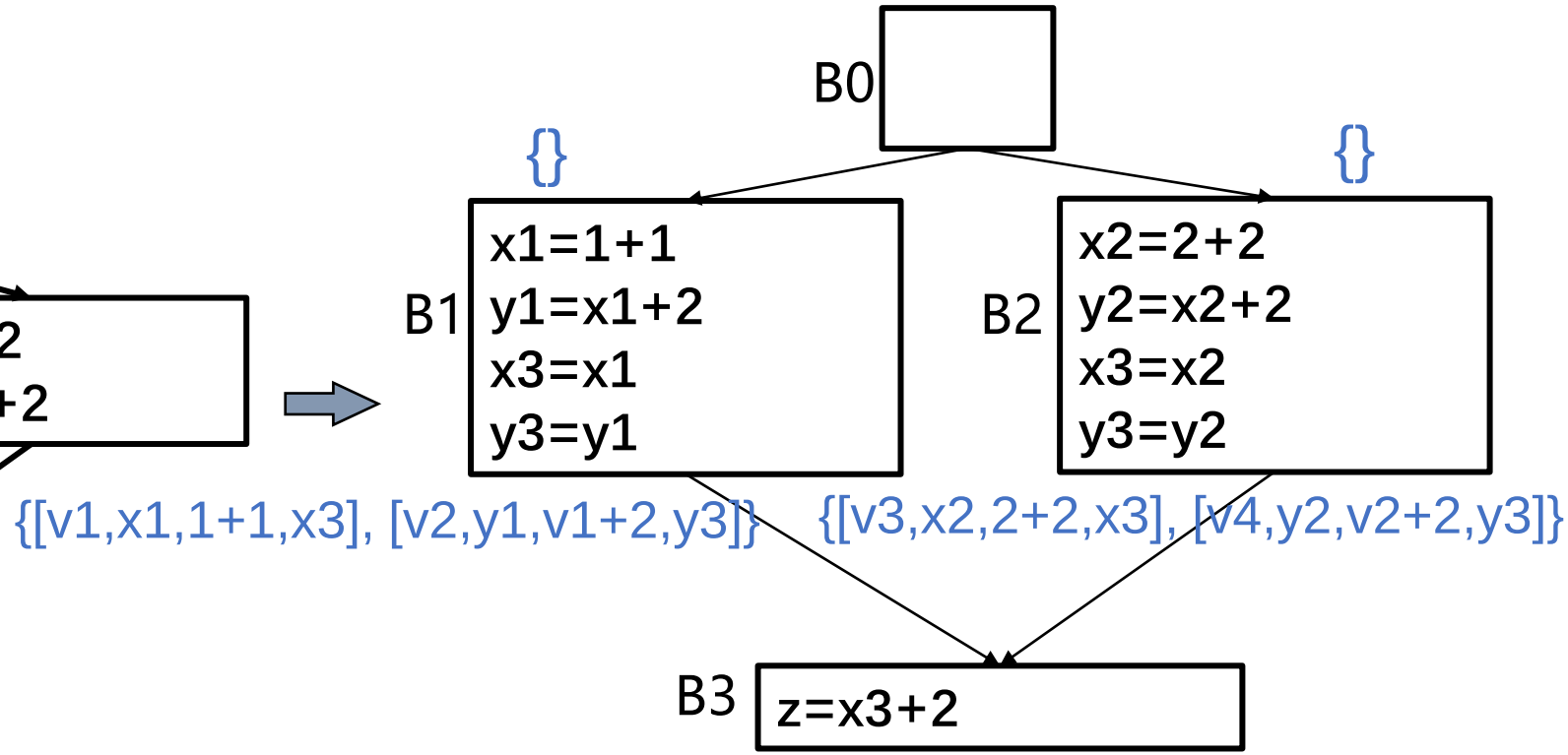




GVN数据流分析举例演示

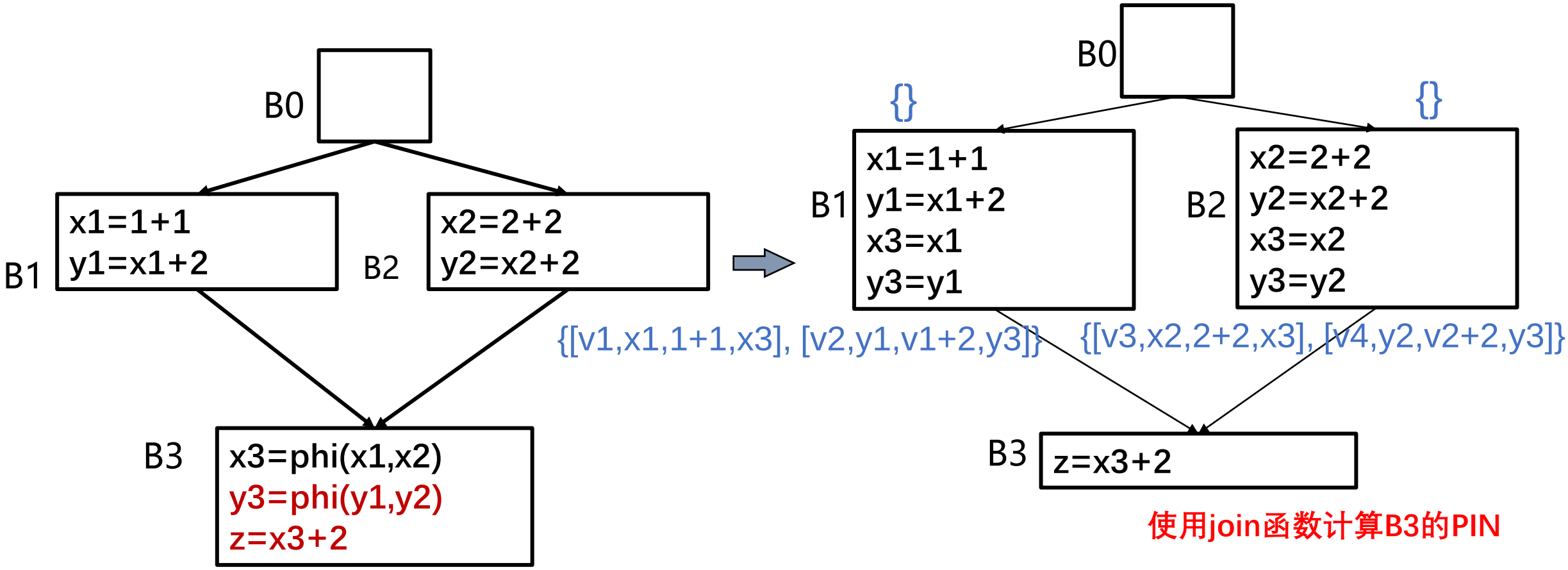
- 同理对B2内每条语句执行转移函数（transferFunction），得到B1B2的出口Partition





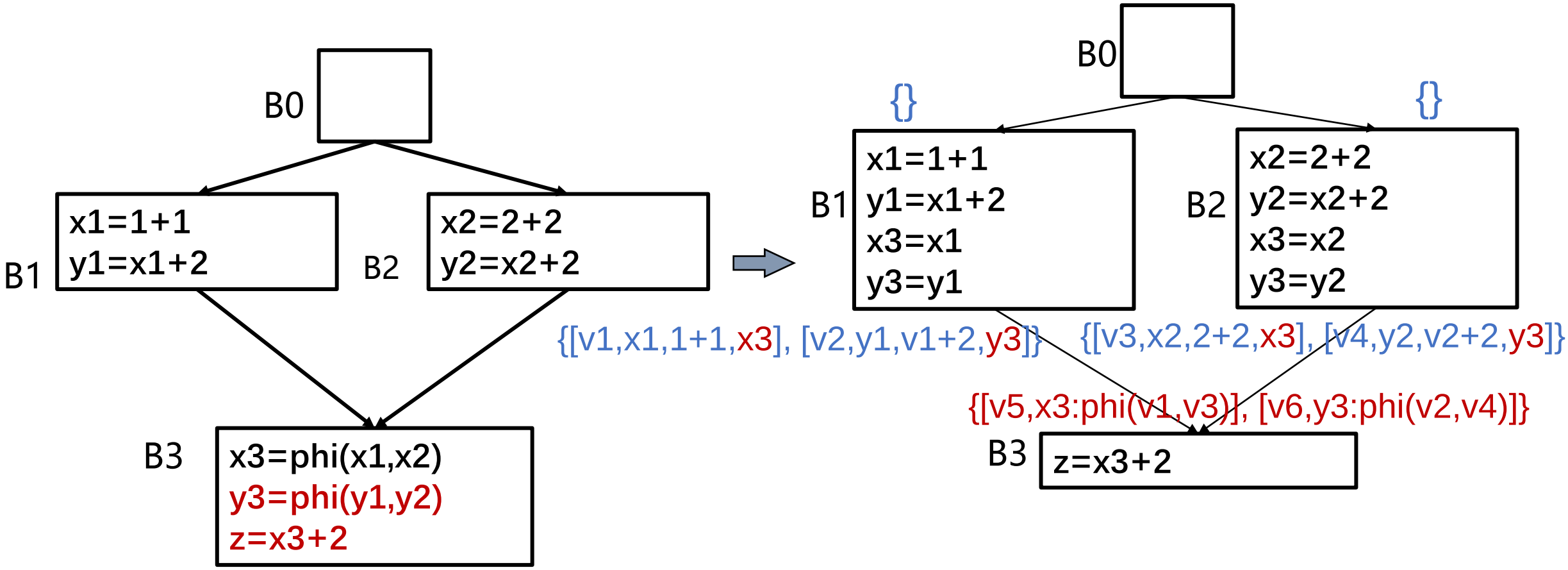
GVN数据流分析举例演示

- 使用join函数计算B3的PIN



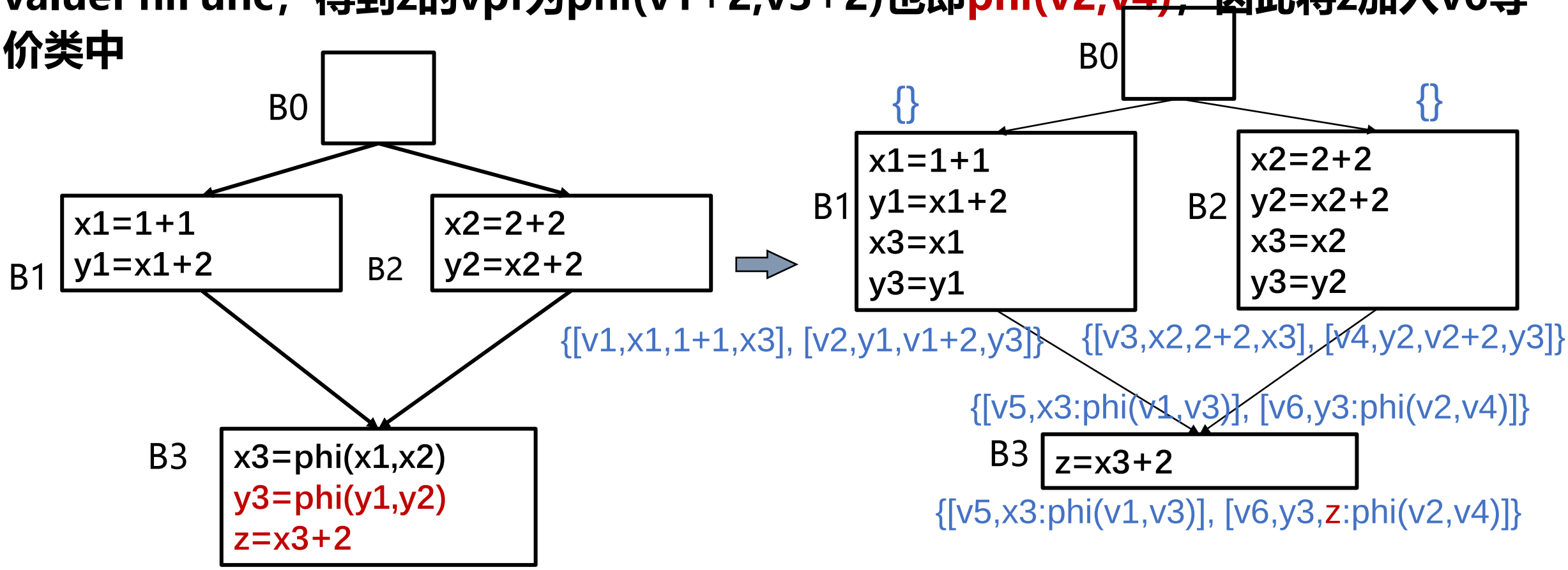
GVN数据流分析举例演示

- 通过Join操作得到B3的入口Partition



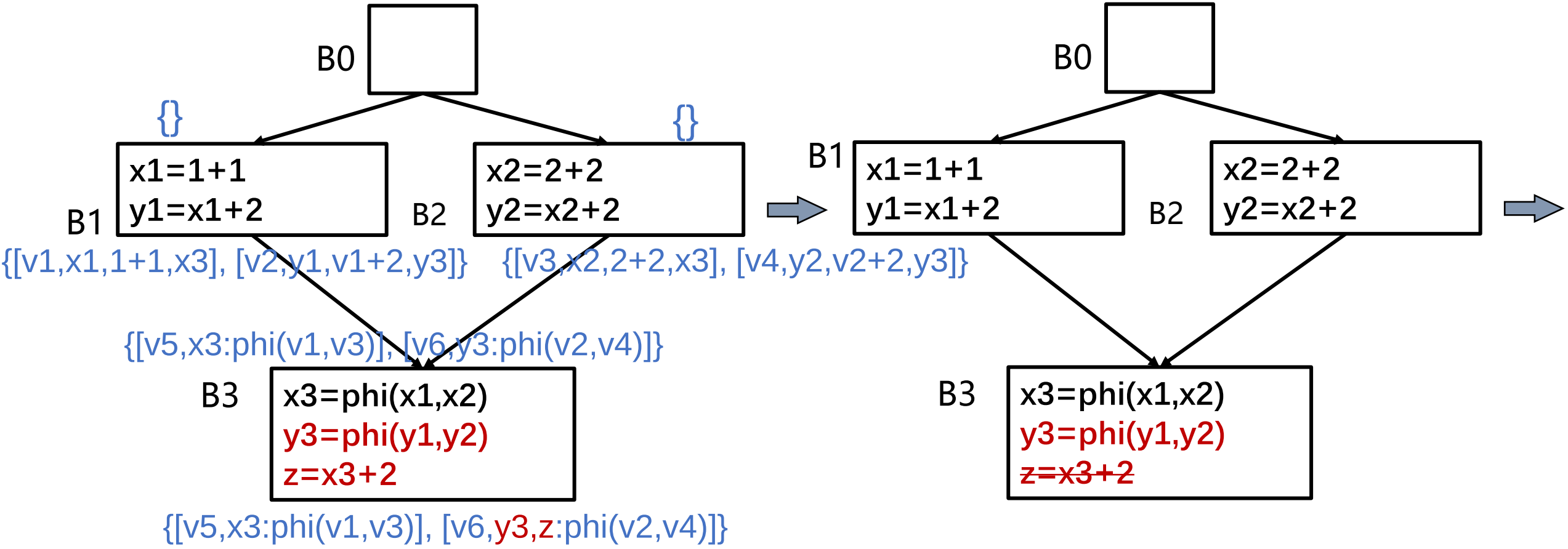
GVN数据流分析举例演示

- 对B3指令执行TransferFunction，发现指令z的ve为phi(v1,v3)+2，执行valuePhiFunc，得到z的vpf为phi(v1+2,v3+2)也即phi(v2,v4)，因此将z加入v6等价类中



GVN数据流分析举例演示

- 依据每个基本块出口数据流分析结果，在原始代码上每个基本块内进行冗余删除，下例中，z 与 y3 等价，可用 y3 替代 z，因此可以删除 z=x3+2 指令



一起努力
打造国产基础系统软件体系！

李 诚

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年12月10日