中国科学技术大学
University of Science and Technology of China

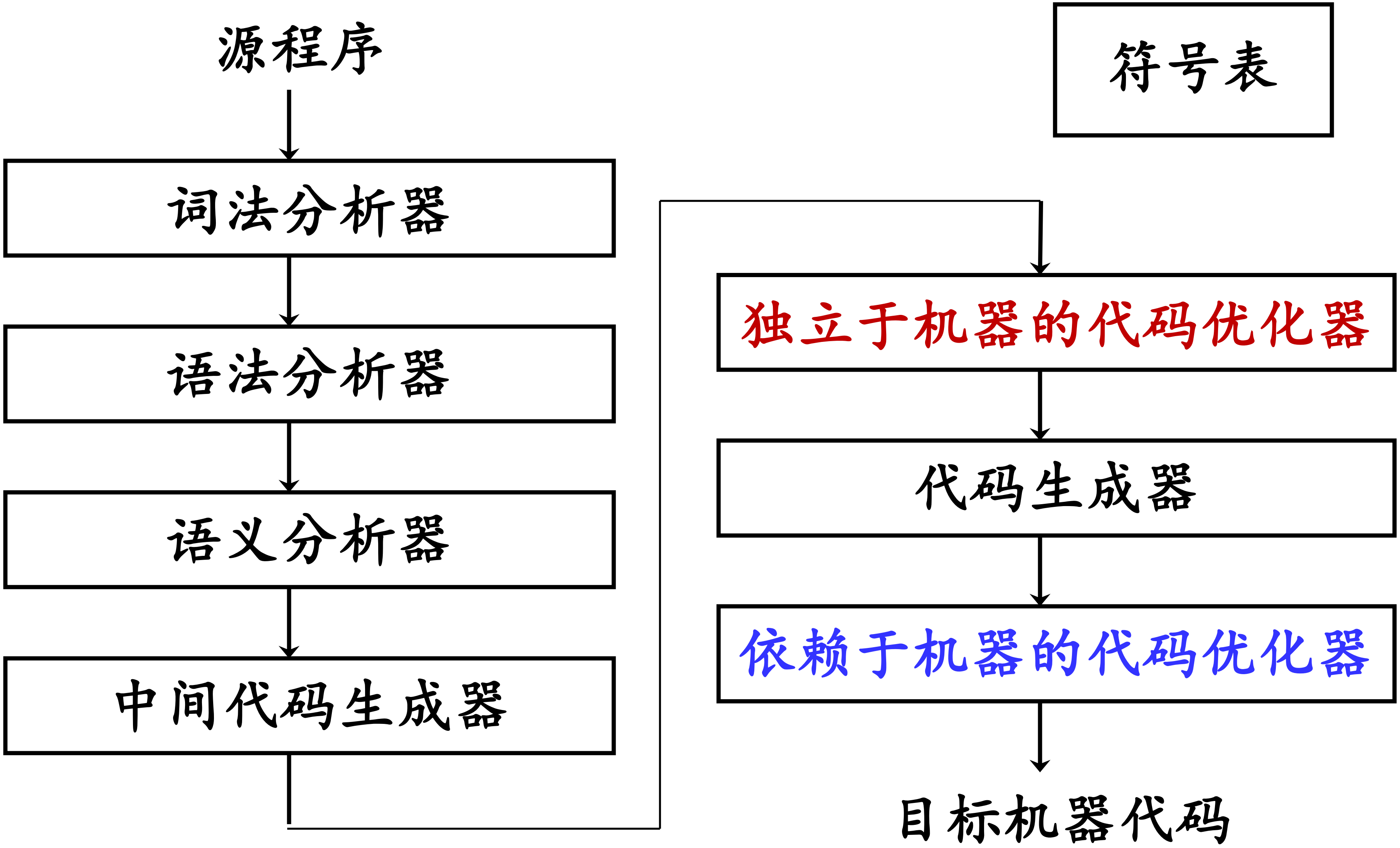
代码优化

《编译原理和技术》

张昱

0551-63603804，yuzhang@ustc.edu.cn
中国科学技术大学
计算机科学与技术学院

本章内容



源程序

词法分析器

语法分析器

语义分析器

中间代码生成器

符号表

独立于机器的代码优化器

代码生成器

依赖于机器的代码优化器

目标机器代码

本章内容

- □ 代码优化
  - ■ 通过**程序变换**（局部变换和全局变换）来改进程序
  - ■ **局部**：基本块内；**全局**：跨基本块
- □ 代码改进变换的标准
  - ■ 代码变换必须保程序的含义
  - ■ 采取安全稳妥的策略
  - ■ 通过变换减少程序的运行时间能平均达到某可度量的值
  - ■ 变换所作的努力是值得的
- □ 本章介绍独立于机器的优化
  - ■ **重点**：数据流分析及其一般框架

1. 优化的源头和种类

- 基本块内优化、全局优化
- 公共子表达式删除、复写传播、死代码删除
- 循环优化

优化的主要源头和主要种类

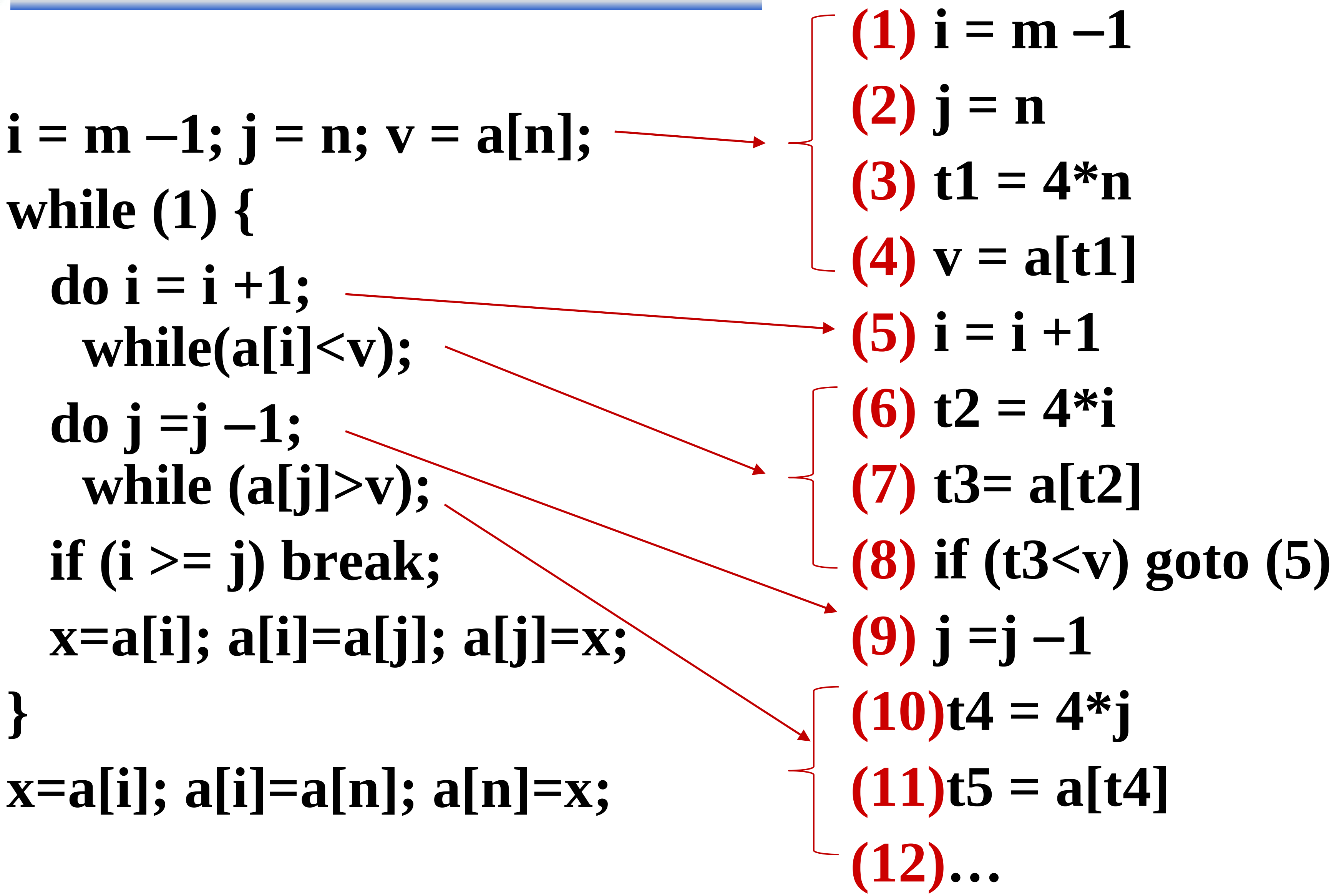
- 主要源头:程序中存在程序员无法避免的冗余计算

  如 A[i][j]、x.f1等数据访问操作

  - 编译后被展开成多步低级算术运算
  - 对同一数据结构的多次访问会产生许多公共的低级运算

- 主要种类
  - 公共子表达式删除(common subexpression elimination)
  - 复写传播(copy propogation)
  - 死代码删除(dead code elimination)
  - 代码外提(loop hoisting, code motion)
  - ……

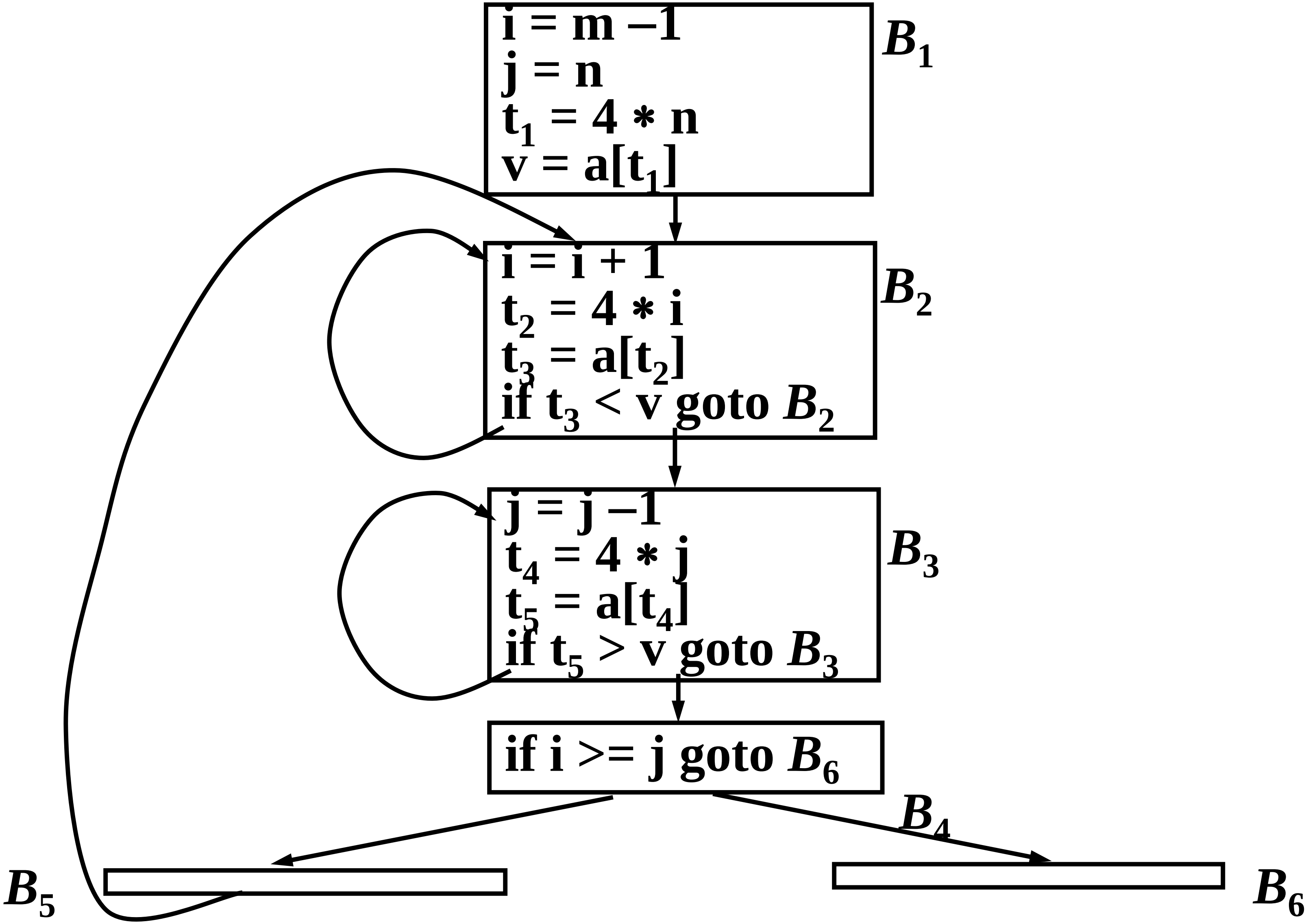
一个实例



```
i = m –1; j = n; v = a[n];
while (1) {
   do i = i +1;
     while(a[i]<v);
   do j =j –1;
     while (a[j]>v);
   if (i >= j) break;
   x=a[i]; a[i]=a[j]; a[j]=x;
}
x=a[i]; a[i]=a[n]; a[n]=x;
```

```
(1) i = m –1
(2) j = n
(3) t1 = 4*n
(4) v = a[t1]
(5) i = i +1
(6) t2 = 4*i
(7) t3= a[t2]
(8) if (t3<v) goto (5)
(9) j =j –1
(10)t4 = 4*j
(11)t5 = a[t4]
(12)…
```

一个实例：程序流图



$i = m - 1$
$j = n$
$t_1 = 4 * n$
$v = a[t_1]$

B_1

$i = i + 1$
$t_2 = 4 * i$
$t_3 = a[t_2]$
if $t_3 < v$ goto B_2

B_2

$j = j - 1$
$t_4 = 4 * j$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3

B_3

if $i >= j$ goto B_6

B_4

B_5

B_6

基本块内的优化

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

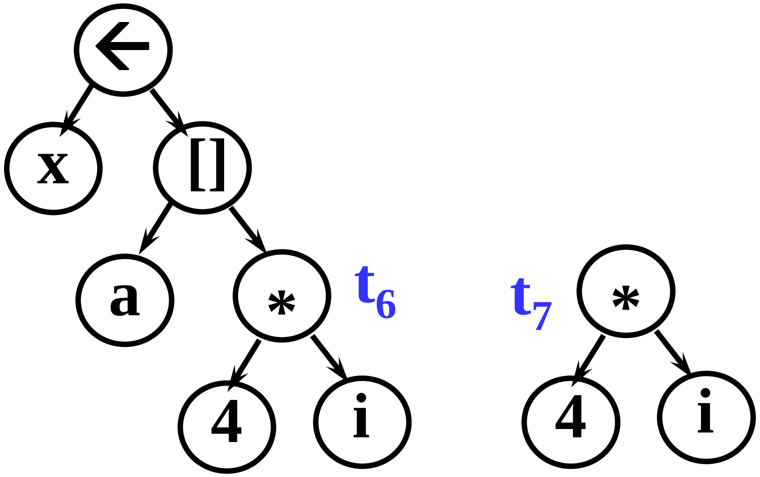
公共子表达式删除(CSE)

$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = 4 * i$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_7] = t_9$
$t_{10} = 4 * j$
$a[t_{10}] = x$
goto B_2

$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = t_6$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_7] = t_9$
$t_{10} = t_8$
$a[t_{10}] = x$
goto B_2

用DAG表示基本块或函数的一部分，如LLVM中的SelectionDAG

用于简化代码，方便实现优化



应用Value Numbering(值编码)决定两个计算是否等价，并去掉等价计算中的一个以进行保语义优化

基本块内的优化

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

公共子表达式删除(CSE)

复写传播

$t_6 = 4 * i$	$t_6 = 4 * i$	$t_6 = 4 * i$
$x = a[t_6]$	$x = a[t_6]$	$x = a[t_6]$
$t_7 = 4 * i$	$t_7 = t_6$	$t_7 = t_6$
$t_8 = 4 * j$	$t_8 = 4 * j$	$t_8 = 4 * j$
$t_9 = a[t_8]$	$t_9 = a[t_8]$	$t_9 = a[t_8]$
$a[t_7] = t_9$	$a[t_7] = t_9$	$a[t_6] = t_9$
$t_{10} = 4 * j$	$t_{10} = t_8$	$t_{10} = t_8$
$a[t_{10}] = x$	$a[t_{10}] = x$	$a[t_8] = x$
goto B_2	goto B_2	goto B_2

复写传播变换的做法：

在复写语句f = g后，尽可能用g代表f

基本块内的优化

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

复写传播本身不是优化，但给其他优化(常量合并、DCE等)带来机会

公共子表达式删除(CSE)

$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = 4 * i$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_7] = t_9$
$t_{10} = 4 * j$
$a[t_{10}] = x$
goto B_2

$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = t_6$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_7] = t_9$
$t_{10} = t_8$
$a[t_{10}] = x$
goto B_2

复写传播

$t_6 = 4 * i$
$x = a[t_6]$
~~$t_7 = t_6$~~
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_6] = t_9$
~~$t_{10} = t_8$~~
$a[t_8] = x$
goto B_2

死代码删除
(DCE)

$t_6 = 4 * i$
$x = a[t_6]$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_6] = t_9$
$a[t_8] = x$
goto B_2

死代码删除

□ 死代码

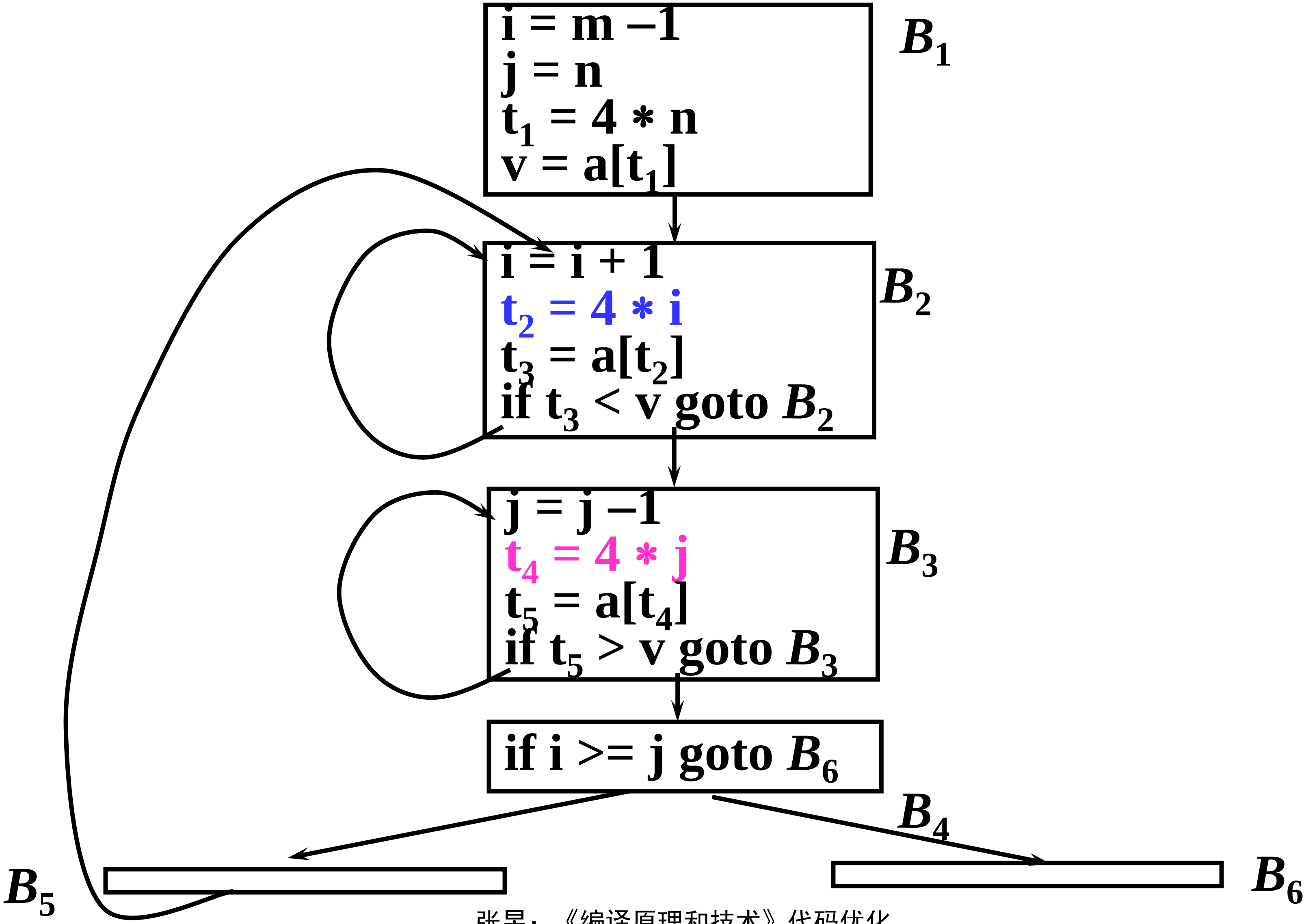
- 死代码指计算的结果绝不被引用的语句
- 一些优化变换可能会引起死代码

例：　为便于调试，可能在程序中加打印语句，测试后改成右边的形式

```
debug = true;             |        debug = false;
. . .                     |        . . .
if (debug) print …        |        if (debug) print …
```

靠优化来保证目标代码中没有该条件语句部分

基本块间的优化



B_1

$i = m - 1$

$j = n$

$t_1 = 4 * n$

$v = a[t_1]$

B_2

$i = i + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2

B_3

$j = j - 1$

$t_4 = 4 * j$

$t_5 = a[t_4]$

if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

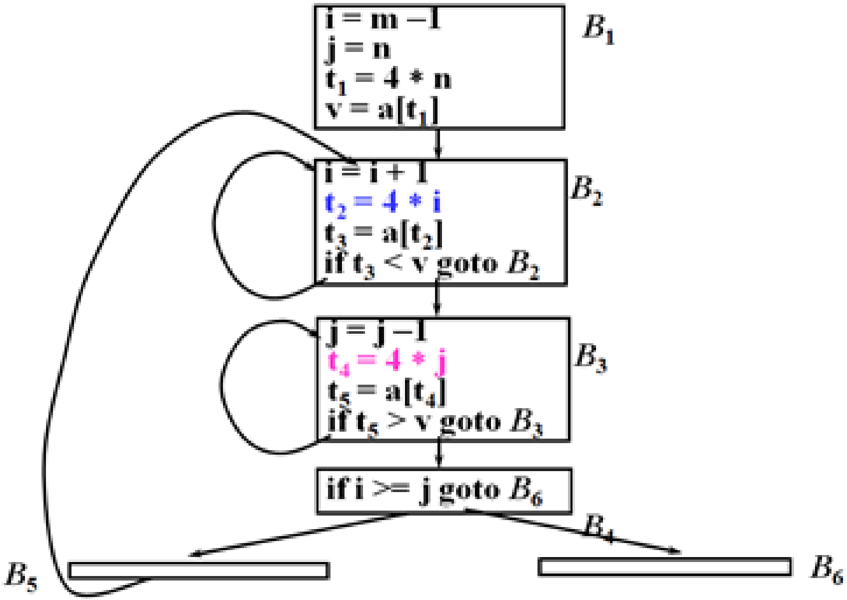
B_5

B_6

基本块间的优化

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

全局CSE、复写传播、DCE



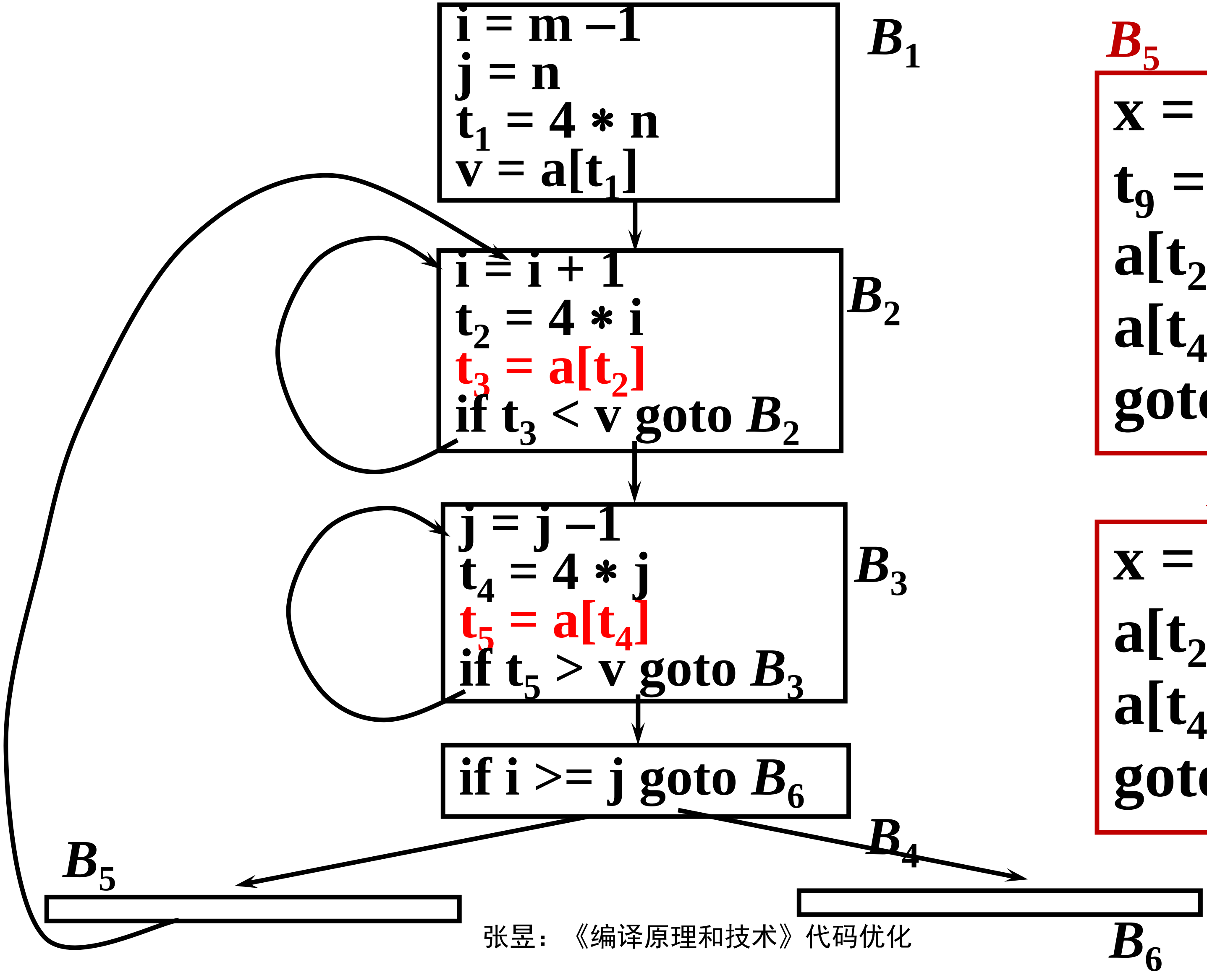
$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = 4 * i$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_7] = t_9$
$t_{10} = 4 * j$
$a[t_{10}] = x$
goto B_2

$t_6 = 4 * i$
$x = a[t_6]$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_6] = t_9$
$a[t_8] = x$
goto B_2

$x = a[t_2]$
$t_9 = a[t_4]$
$a[t_2] = t_9$
$a[t_4] = x$
goto B_2

$x = t_3$
~~$t_9 = a[t_4]$~~
$a[t_2] = t_5$
$a[t_4] = x$
goto B_2

基本块间的优化



B_1

$i = m - 1$
$j = n$
$t_1 = 4 * n$
$v = a[t_1]$

B_2

$i = i + 1$
$t_2 = 4 * i$
$t_3 = a[t_2]$
if $t_3 < v$ goto B_2

B_3

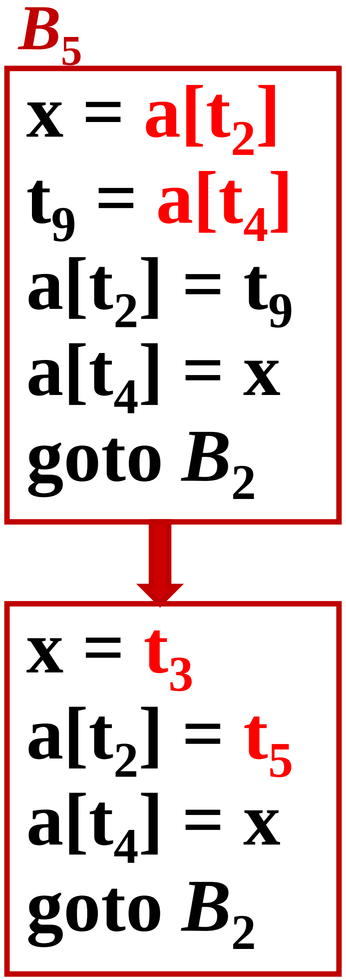
$j = j - 1$
$t_4 = 4 * j$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

B_6



B_5

$x = a[t_2]$
$t_9 = a[t_4]$
$a[t_2] = t_9$
$a[t_4] = x$
goto B_2

$x = t_3$
$a[t_2] = t_5$
$a[t_4] = x$
goto B_2

实例：B6的优化

B_6 $x = a[i]; a[i] = a[n]; a[n] = x;$

B_1 : $t_1 = 4 * n$

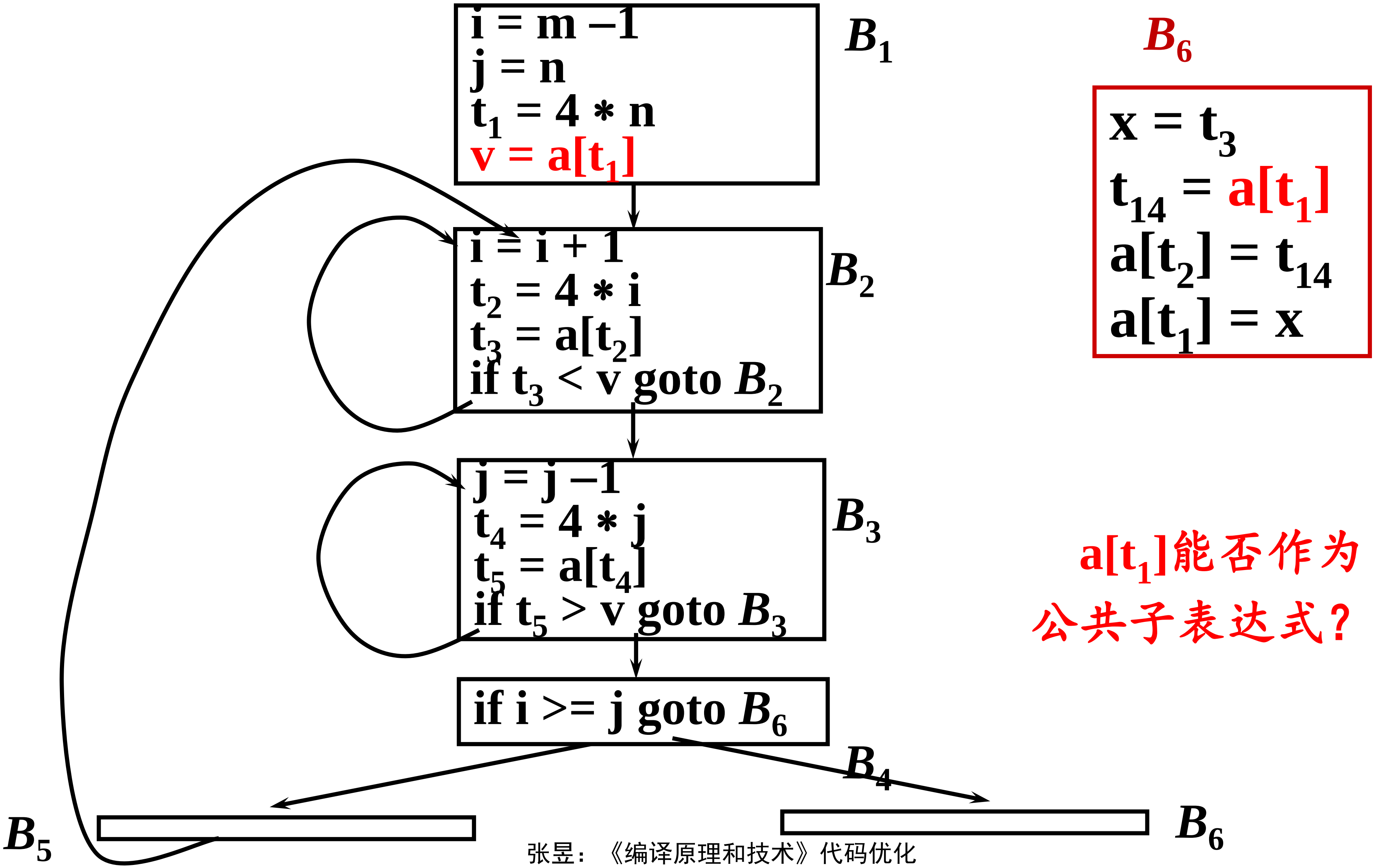
B_2 : $t_2 = 4 * i$, $t_3 = a[t_2]$

$t_{11} = 4 * i$
$x = a[t_{11}]$
$t_{12} = 4 * i$
$t_{13} = 4 * n$
$t_{14} = a[t_{13}]$
$a[t_{12}] = t_{14}$
$t_{15} = 4 * n$
$a[t_{15}] = x$

基本块内和
基本块间的优化

$x = t_3$
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = x$

控制流对优化的影响



B_1

$i = m - 1$
$j = n$
$t_1 = 4 * n$
$v = a[t_1]$

B_2

$i = i + 1$
$t_2 = 4 * i$
$t_3 = a[t_2]$
if $t_3 < v$ goto B_2

B_3

$j = j - 1$
$t_4 = 4 * j$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

B_6

B_6

$x = t_3$
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = x$

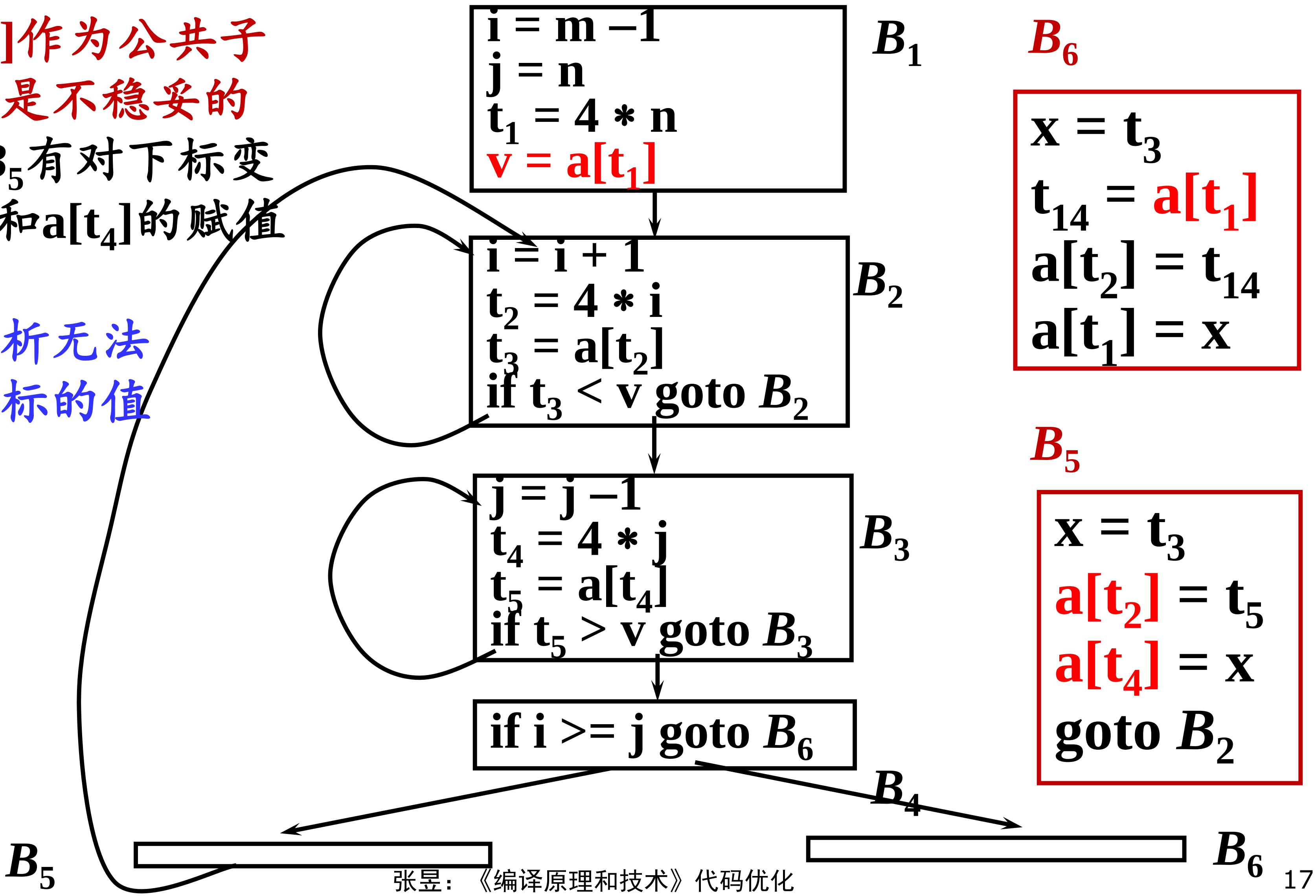
$a[t_1]$能否作为公共子表达式？

控制流对优化的影响

把$a[t_1]$作为公共子表达式是不稳妥的

因为B_5有对下标变量$a[t_2]$和$a[t_4]$的赋值

静态分析无法确定下标的值



B_1

$i = m - 1$
$j = n$
$t_1 = 4 * n$
$v = a[t_1]$

B_2

$i = i + 1$
$t_2 = 4 * i$
$t_3 = a[t_2]$
if $t_3 < v$ goto B_2

B_3

$j = j - 1$
$t_4 = 4 * j$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3

B_4

if $i >= j$ goto B_6

B_5

B_6

B_6

$x = t_3$
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = x$

B_5

$x = t_3$
$a[t_2] = t_5$
$a[t_4] = x$
goto B_2

循环优化

□ 循环优化的主要技术

- 归纳变量删除(induction variable elimination)
- 强度削弱(strength reduction)：如将乘法变换成加法
- 代码外提(loop hoisting, code motion)：将循环不变的运算外提

□ 代码外提

例：while (i <= limit – 2) … limit - 2是循环不变式

代码外提后变换成

```
t = limit – 2;
while (i <= t ) …
```

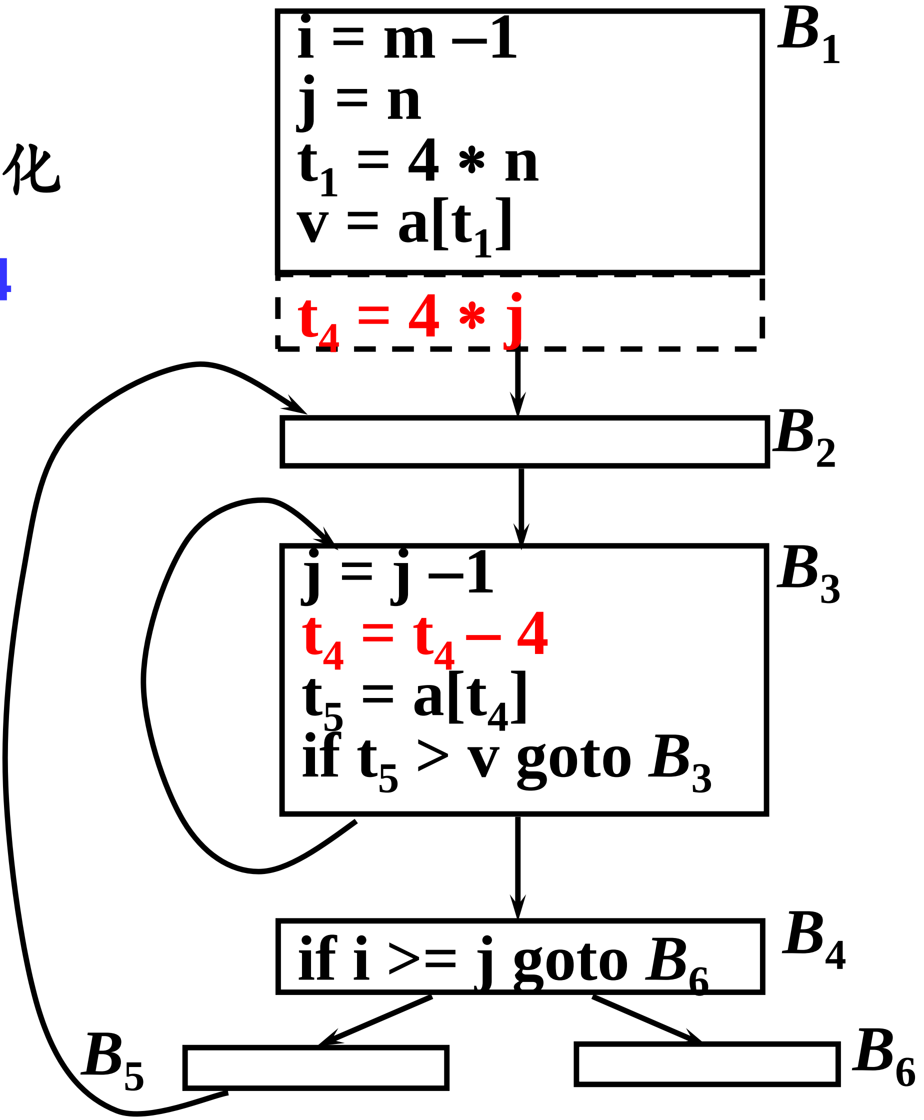
循环优化-强度削弱

□ 强度削弱

- j和t_4的值步调一致地变化
- $t_4 = 4 * j$ 变换为$t_4 = t_4 - 4$
- 在循环前增加$t_4 = 4 * j$

B_3

```
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
```



循环优化-强度削弱

- □ **强度削弱**
  - ■ B_2也可以类似地变换
- □ **归纳变量删除**
  - ■ 循环控制条件i >= j
    可以表示为t_2 >=t_4

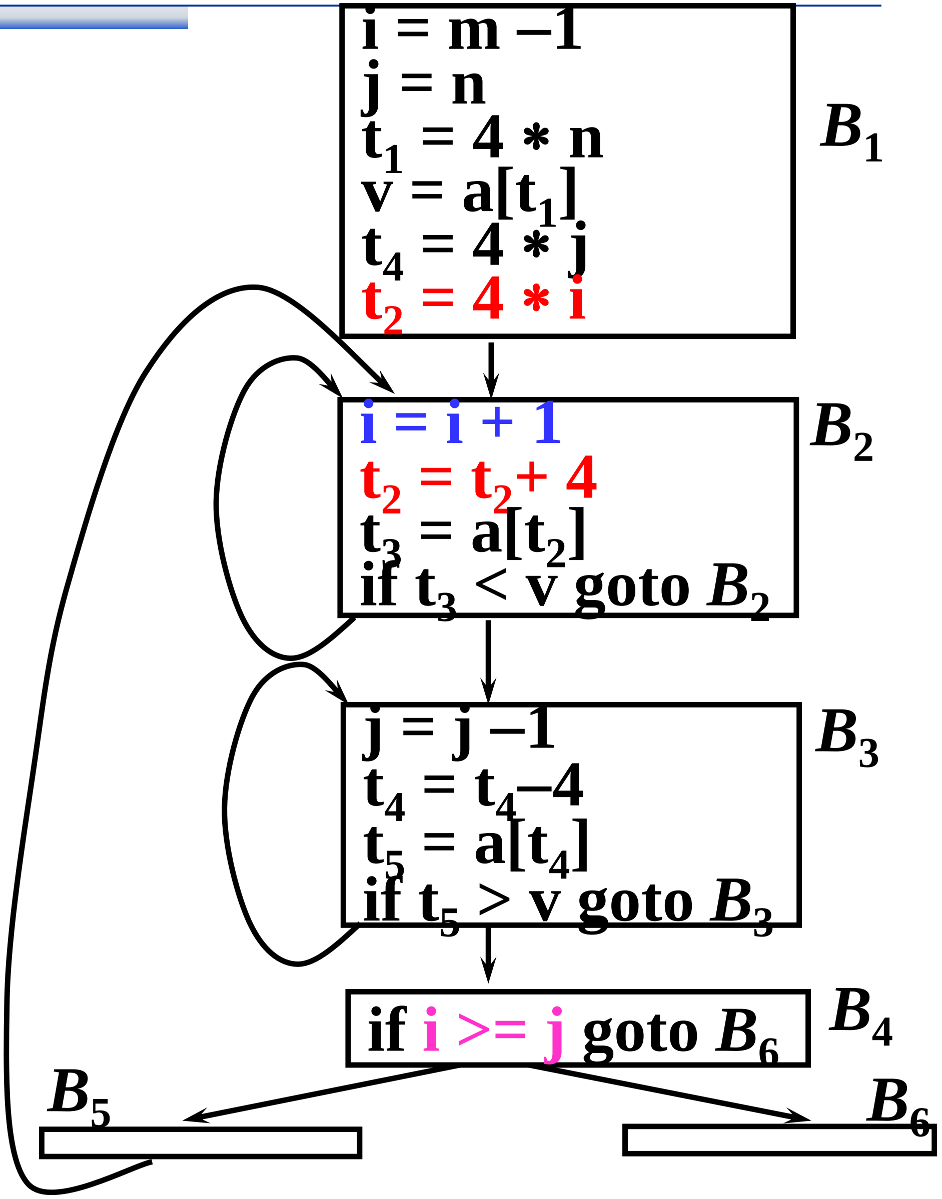
B_2

$i = i + 1$

$t_2 = 4 * i$

$t_3 = a[t_2]$

if $t_3 < v$ goto B_2



B_1:
$i = m - 1$
$j = n$
$t_1 = 4 * n$
$v = a[t_1]$
$t_4 = 4 * j$
$t_2 = 4 * i$

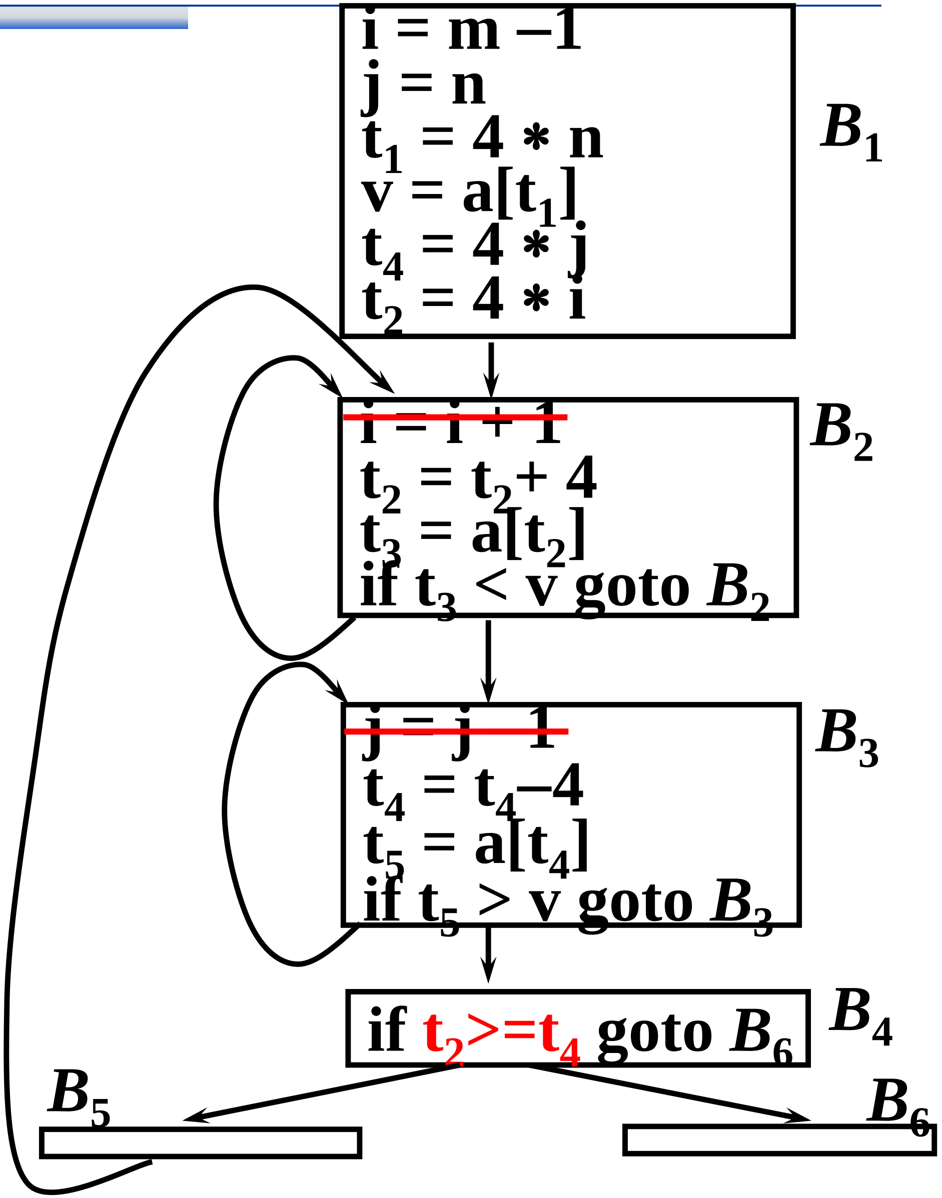
B_2:
$i = i + 1$
$t_2 = t_2 + 4$
$t_3 = a[t_2]$
if $t_3 < v$ goto B_2

B_3:
$j = j - 1$
$t_4 = t_4 - 4$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3

B_4:
if $i >= j$ goto B_6

循环优化-强度削弱

- □ 强度削弱
  - B_2也可以类似地变换
- □ 归纳变量删除
  - 循环控制条件i >= j 可以表示为t_2 >=t_4
  - 归纳变量 i和j 可删除



B_1

$i = m - 1$
$j = n$
$t_1 = 4 * n$
$v = a[t_1]$
$t_4 = 4 * j$
$t_2 = 4 * i$

B_2

~~i = i + 1~~
$t_2 = t_2 + 4$
$t_3 = a[t_2]$
if $t_3 < v$ goto B_2

B_3

~~j = j – 1~~
$t_4 = t_4 - 4$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3

B_4

if $t_2 >= t_4$ goto B_6

代码优化总结

□ 局部优化

- 基本块内的优化
- 窥孔(peephole)优化：仅分析一个滑动窗口内的指令。每次转换后可能还会暴露相邻窗口间的某些优化机会

✓ 冗余指令删除：如

```
mov r0, a        // r0 => a
mov a, r0        // a => r0, 可删除
```

✓ 删除死代码

```
goto L1
goto L2                    // 语句前无标号，死代码
```

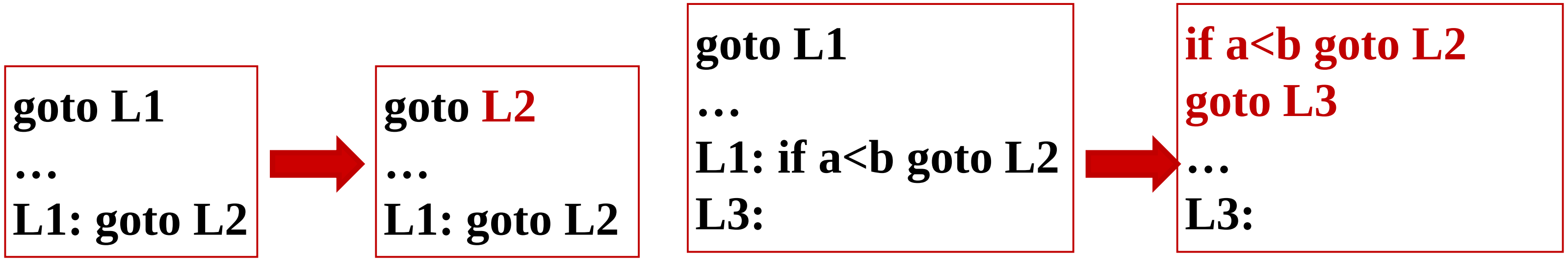
代码优化总结

□ 局部优化

- 基本块内的优化
- **窥孔(peephole)优化**：仅分析一个滑动窗口内的指令。每次转换后可能还会暴露相邻窗口间的某些优化机会

✓ 冗余指令删除、删除死代码

✓ 控制流优化



```
goto L1
…
L1: goto L2
```

```
goto L2
…
L1: goto L2
```

```
goto L1
…
L1: if a<b goto L2
L3:
```

```
if a<b goto L2
goto L3
…
L3:
```

代码优化总结

□ 局部优化

- 基本块内的优化
- 窥孔(peephole)优化：仅分析一个滑动窗口内的指令。每次转换后可能还会暴露相邻窗口间的某些优化机会

✓ 冗余指令删除、删除死代码、控制流优化

✓ 强度削弱、删除无用指令

mul $8, r0   => shiftleft $3, r0

add $0, r1   mul $1, r2   // 均可删除

✓ 利用目标机指令特点

如 **inc**、**enter**(建立栈帧)、**leave**(清除栈帧)、龙芯的乘加指令、向量扩展指令等等

代码优化总结

- □ 局部优化
  - 基本块内的优化
  - 窥孔(peephole)优化：仅分析一个滑动窗口内的指令。每次转换后可能还会暴露相邻窗口间的某些优化机会
- □ 全局优化
  - 基本块间优化（过程内）
  - 过程间优化：程序全局优化

2. 程序分析

□ 控制流分析

□ 数据流分析：到达-定值分析、数据流分析模式、活跃变量分析、可用表达式分析等

控制流分析与数据流分析

□ 控制流分析

- 发现每一个过程内的控制流层次结构
- 从构成过程的基本块开始，然后构造流图
- 使用必经结点来找出循环

□ 数据流分析

- 确定一个过程中与数据处理有关的全局信息
- 例如，常量传播分析是力求判定对一个特定变量的所有赋值在某个特定程序点是否总是给定相同的常数值。如果是这样，则在那一点可用一个常数来替代该变量

数据流分析举例

程序点(5)的所有程序状态

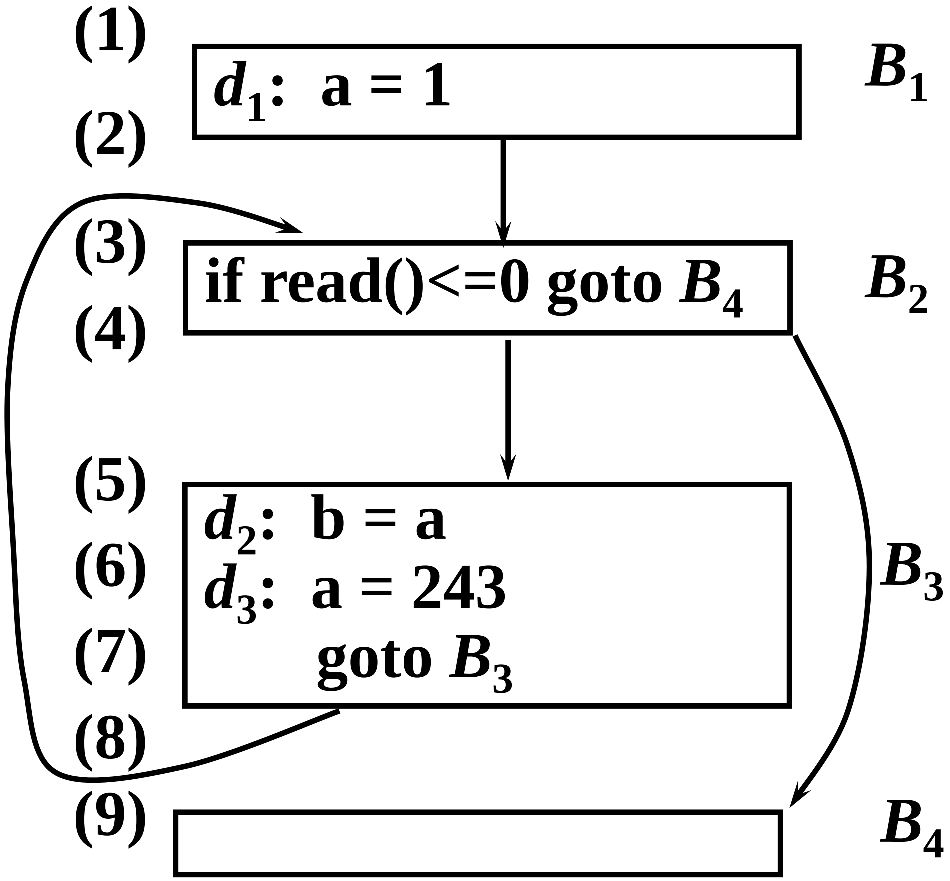
- a ∈ {1, 243}
- a由{d_1, d_3}定值

1. **到达-定值**

   {d_1, d_3}的定值到达点(5)

2. **常量合并**

   a在点(5)不是常量



(1)

B_1

d_1: a = 1

(2)

(3)

B_2

if read()<=0 goto B_4

(4)

(5)

B_3

d_2: b = a

d_3: a = 243

goto B_3

(6)

(7)

(8)

(9)

B_4

数据流分析

□ 数据流分析

- 分析程序行为时，必须在其流图上考虑所有的执行路径（在调用或返回语句被执行时，还需要考虑执行路径在多个流图之间的跳转）
- 由于存在循环，从流图得到的程序执行路径数无限，且执行路径长度没有有限的上界
- 每个程序点的不同状态数也可能无限

程序状态：存储单元到值的映射

不可能知晓所有执行路径上的所有程序状态

数据流抽象

- □ 数据流分析不打算
  - ■ 区分到达一个程序点的不同执行路径
  - ■ 掌握该程序点的每个完整的状态
- □ 数据流分析要点
  - ■ 从程序状态中**抽取**特定数据流分析所需信息➔**数据流值**
  - ■ 总结出用于该分析目的的一组**有限的事实**，且这组事实和到达这个程序点的路径无关，即从任何路径到达该程序点都有这样的事实

    保守的：得到的信息不会误解程序的行为
  - ■ 分析的目的不同，从程序状态提炼的信息也不同

到达-定值(reaching definitions)

- **数据流值：到达一个程序点的所有定值**
  - 每个定值语句有唯一编号，**数据流值：**定值编号的集合

    可用来判断一个变量在某程序点是否为常量、是否无初值
- **别名给到达-定值的计算带来困难**
  - 过程参数、数组访问、间接引用等都有可能引起别名

    例如：若**p = = q**，则**p->next**和**q->next**互为别名
  - 程序分析必须是稳妥的
  - 本章其余部分仅考虑变量无别名的情况
- **定值的注销(kill)**

  在一条执行路径上，对*x* 的赋值会注销先前对*x*的所有赋值

到达-定值的计算

□ 基本块向下暴露的定值

- *gen*[*B*]：基本块*B*生成的定值
- *kill*[*B*]：基本块*B*注销的定值

$gen\ [B_1] = \{d_1, d_2, d_3\}$

$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$

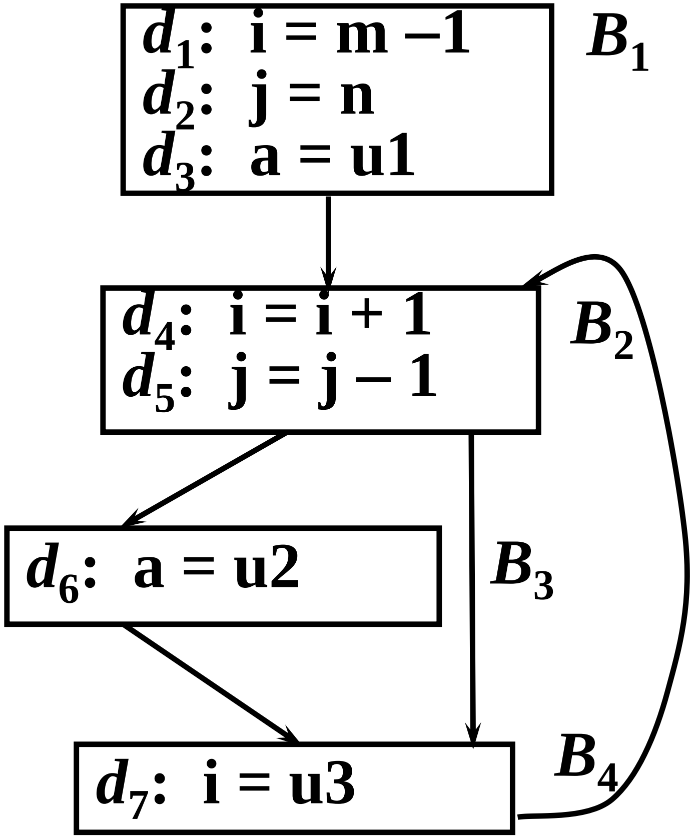
$kill\ [B_2]=\{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$

$kill\ [B_3]=\{d_3\}$

$gen\ [B_4] = \{d_7\}$

$kill\ [B_4]=\{d_1, d_4\}$



d_1: i = m –1
d_2: j = n
d_3: a = u1

B_1

d_4: i = i + 1
d_5: j = j – 1

B_2

d_6: a = u2

B_3

d_7: i = u3

B_4

基本块的*gen* 和 *kill* 的计算

- 对三地址指令 d: u = v + w，它的状态迁移函数是

  $f_d(x) = gen_d \cup (x - kill_d)$,

  x为到达指令入口点的定值

- 若$f_1(x) = gen_1 \cup (x - kill_1)$, $f_2(x) = gen_2 \cup (x - kill_2)$

  则：$f_2(f_1(x)) = gen_2 \cup (gen_1 \cup (x - kill_1) - kill_2)$

  $= (gen_2 \cup (gen_1 - kill_2)) \cup (x- (kill_1 \cup kill_2))$

- 若基本块B有n条三地址指令

  $kill_B = kill_1 \cup kill_2 \cup \ldots \cup kill_n$

  $gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \ldots \cup (gen_1 - kill_2 - kill_3 - \ldots - kill_n)$

到达-定值的数据流方程

□ 数据流方程/等式(flow equation)

- gen_B：B中生成的且能到达B的结束点的定值
- $kill_B$：B注销的定值
- IN[B]：能到达B的开始点的定值集合
- OUT[B]：能到达B的结束点的定值集合

两组等式（根据*gen*和*kill*定义IN和OUT）

- $\text{IN}[B] = \bigcup_{P是B的前驱} \text{OUT}[P]$
- $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$
- $\text{OUT}[\text{ENTRY}] = \varnothing$

到达-定值方程组的迭代求解，最终到达不动点(MFP)

到达-定值的迭代计算算法

```
// 正向数据流分析
OUT[ENTRY] = Ø;
for (除了ENTRY以外的每个块B)  OUT[B] = Ø;
while (任何一个OUT出现变化)
    for (除了ENTRY以外的每个块B) {
        IN[B] = $\bigcup_{P是B的前驱}$ OUT[P];
        OUT[B] = $f_B$(IN[B]);
        // $f_B(IN[B]) = gen_B \cup (IN[B] - kill_B)$
    }
```

这里采用**流不敏感**的算法，即没有按基本块的执行次序依次处理每个基本块

引用-定值(Use-Def)

- **引用-定值链(简称ud链)**
  - 对于变量的每一个引用，记录到达该引用的所有定值

- **通过到达-定值数据流分析计算并形成ud链**
  - 如果块B中变量a的引用之前有a的定值，那么只有a的最后一次定值会在该引用的ud链中
  - 如果块B中变量a的引用之前无a的定值，那么该引用的ud链就是IN[B]中a的定值的集合

…

d: a = …

…

… = …a…

…

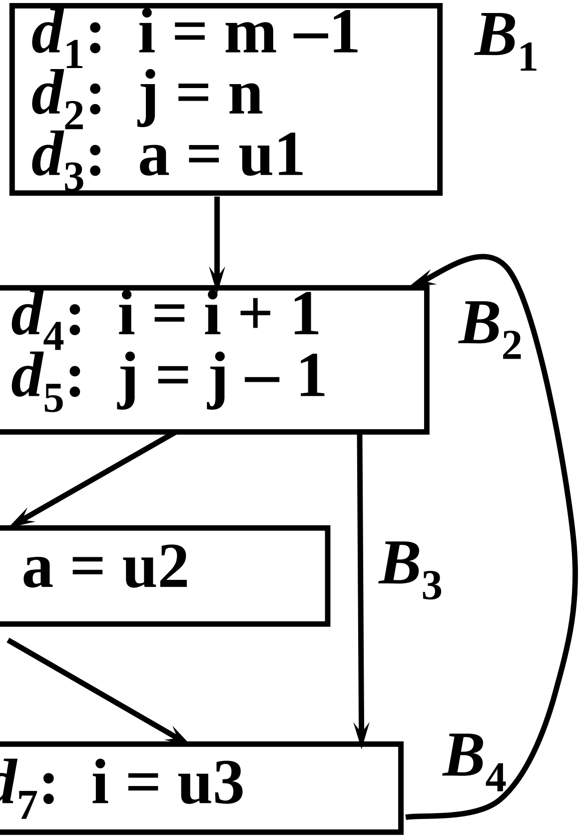
…

… = …a…

…

到达-定值计算示例

	IN [B]	OUT [B]
B_1		000 0000
B_2		000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

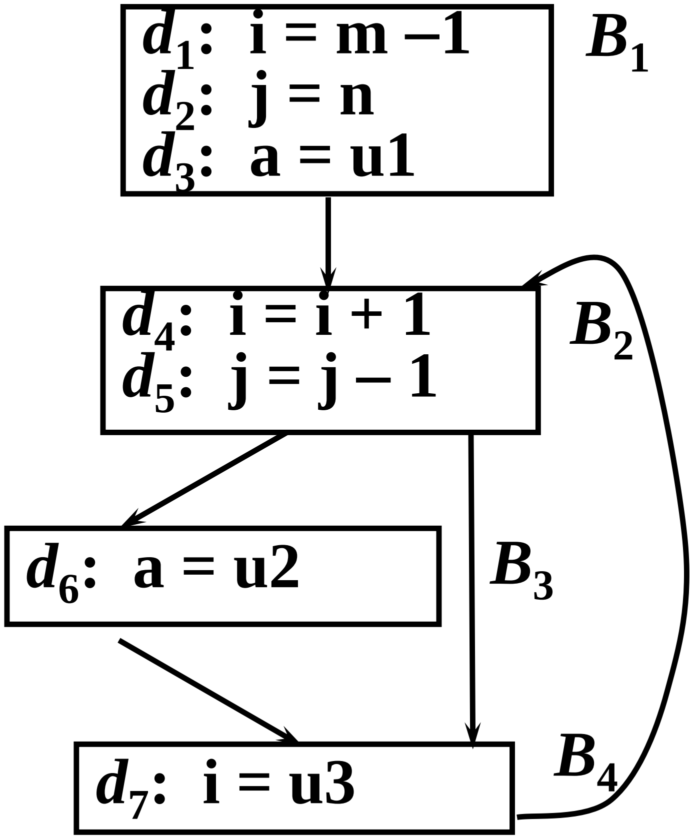
$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

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

$$\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B)$$

	IN [B]	OUT [B]
B_1	000 0000	000 0000
B_2		000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

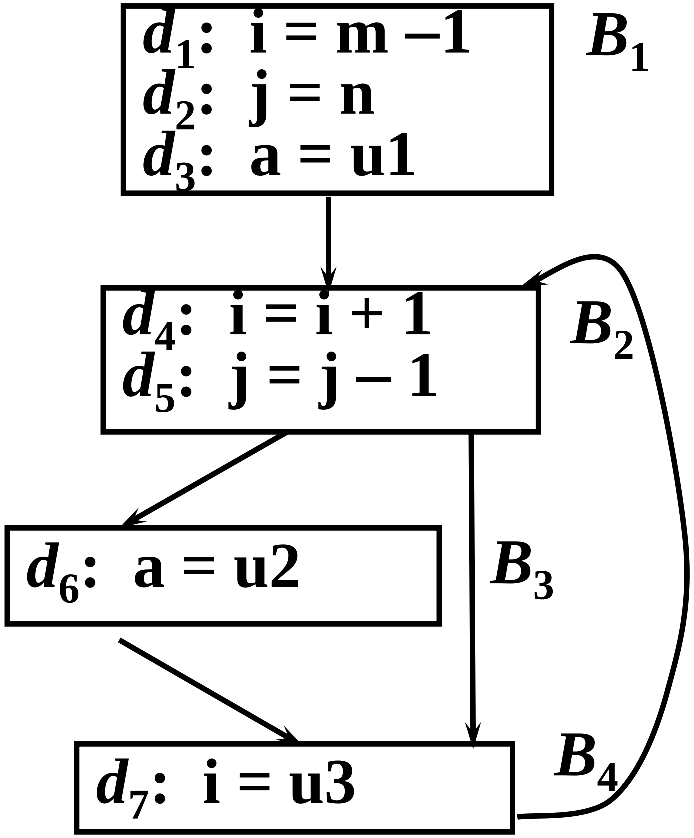
$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

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

$$\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B)$$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2		000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$

$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$

$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$

$kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$

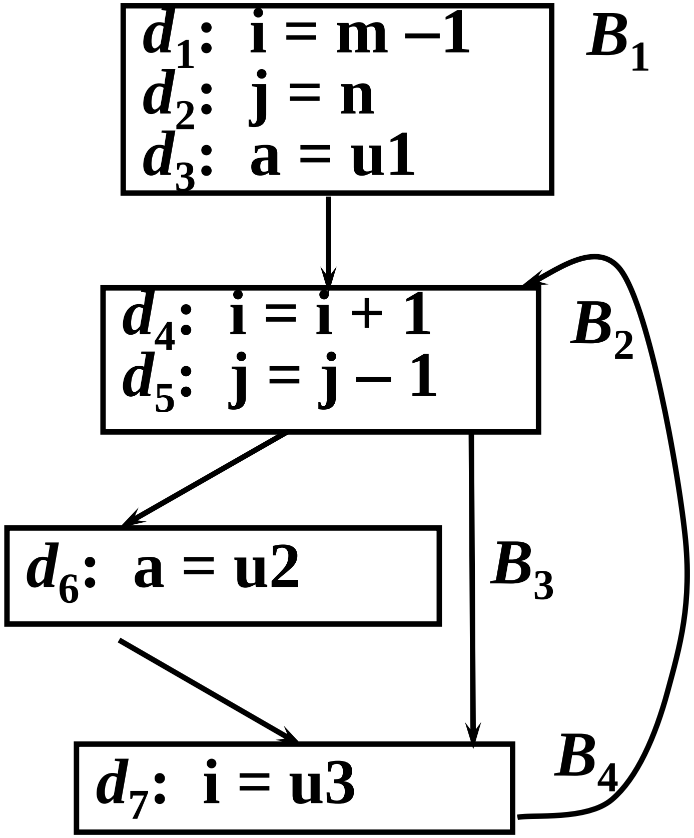
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

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

$$\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B)$$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$

$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$

$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$

$kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$

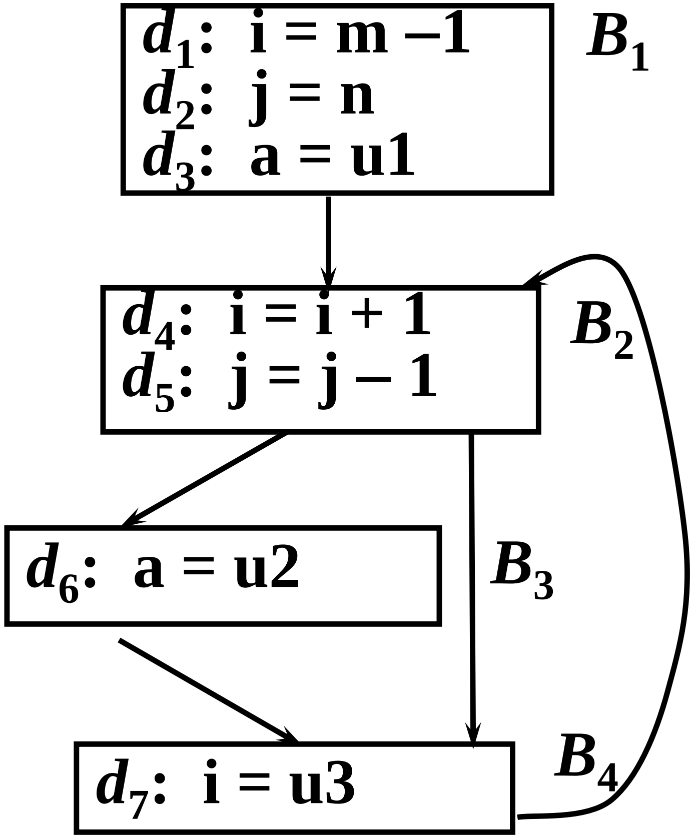
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

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

$$\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B)$$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3		000 0000
B_4		000 0000



B_1
d_1: i = m −1
d_2: j = n
d_3: a = u1

B_2
d_4: i = i + 1
d_5: j = j − 1

B_3
d_6: a = u2

B_4
d_7: i = u3

$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

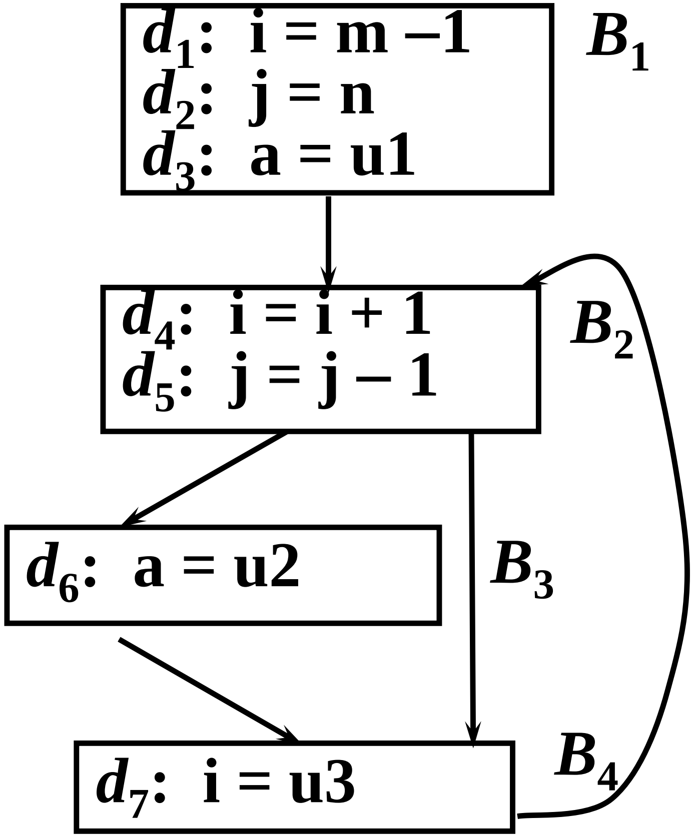
$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

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

$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 0000
B_4		000 0000



B_1

d_1: i = m −1

d_2: j = n

d_3: a = u1

B_2

d_4: i = i + 1

d_5: j = j – 1

B_3

d_6: a = u2

B_4

d_7: i = u3

$gen\ [B_1] = \{d_1, d_2, d_3\}$

$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$

$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$

$kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$

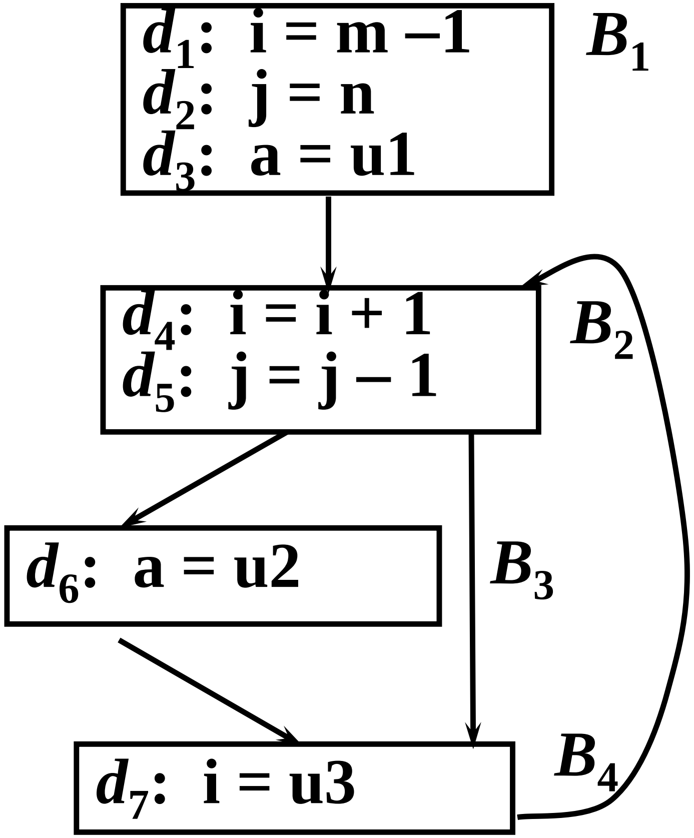
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

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

$$\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B)$$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 1110
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

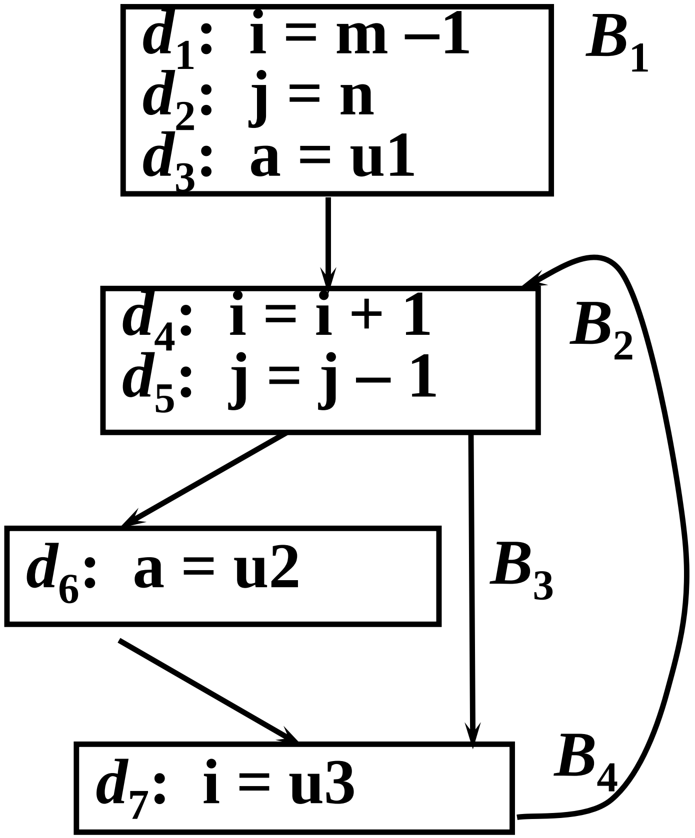
$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

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

$$\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B)$$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 1110
B_4	001 1110	000 0000



d_1: i = m −1
d_2: j = n
d_3: a = u1

d_4: i = i + 1
d_5: j = j – 1

d_6: a = u2

d_7: i = u3

$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

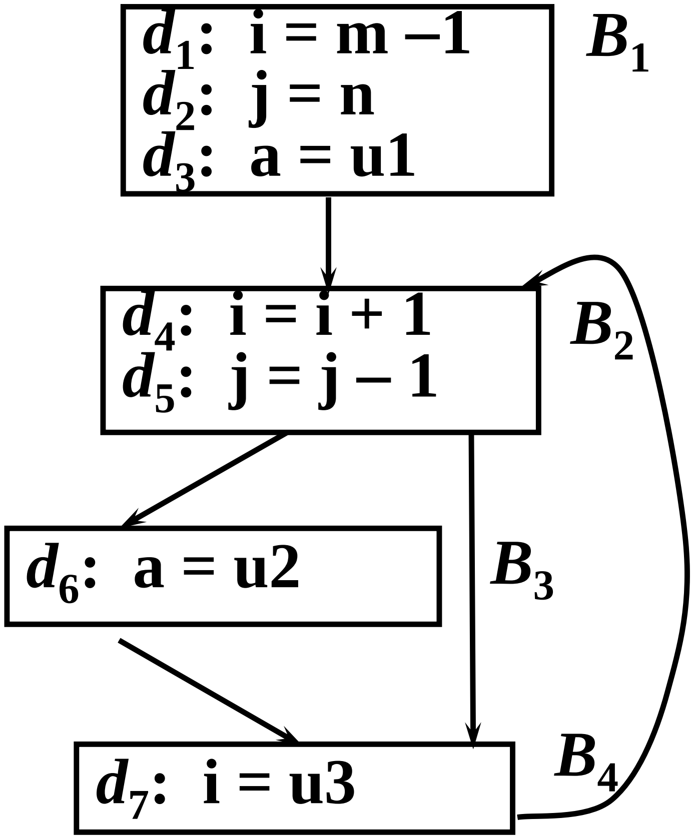
$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

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

$$\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B)$$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 1110
B_4	001 1110	001 0111



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

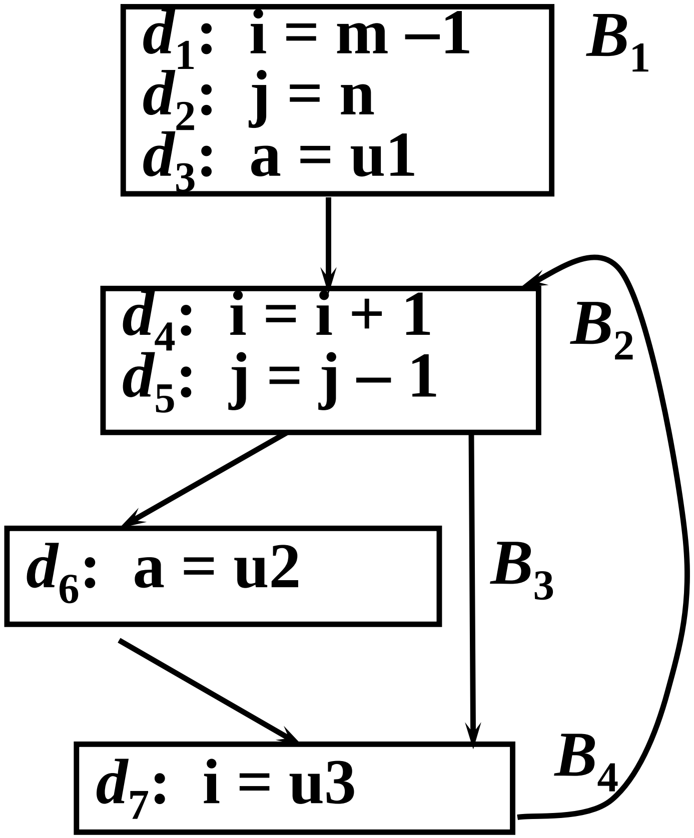
$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

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

$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0111	001 1100
B_3	001 1100	000 1110
B_4	001 1110	000 0000



d_1: i = m −1
d_2: j = n
d_3: a = u1
B_1

d_4: i = i + 1
d_5: j = j − 1
B_2

d_6: a = u2
B_3

d_7: i = u3
B_4

$gen[B_1] = \{d_1, d_2, d_3\}$
$kill[B_1]=\{d_4, d_5, d_6, d_7\}$

$gen[B_2] = \{d_4, d_5\}$
$kill[B_2] = \{d_1, d_2, d_7\}$

$gen[B_3] = \{d_6\}$
$kill[B_3] = \{d_3\}$

$gen[B_4] = \{d_7\}$
$kill[B_4] = \{d_1, d_4\}$

到达-定值计算示例

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

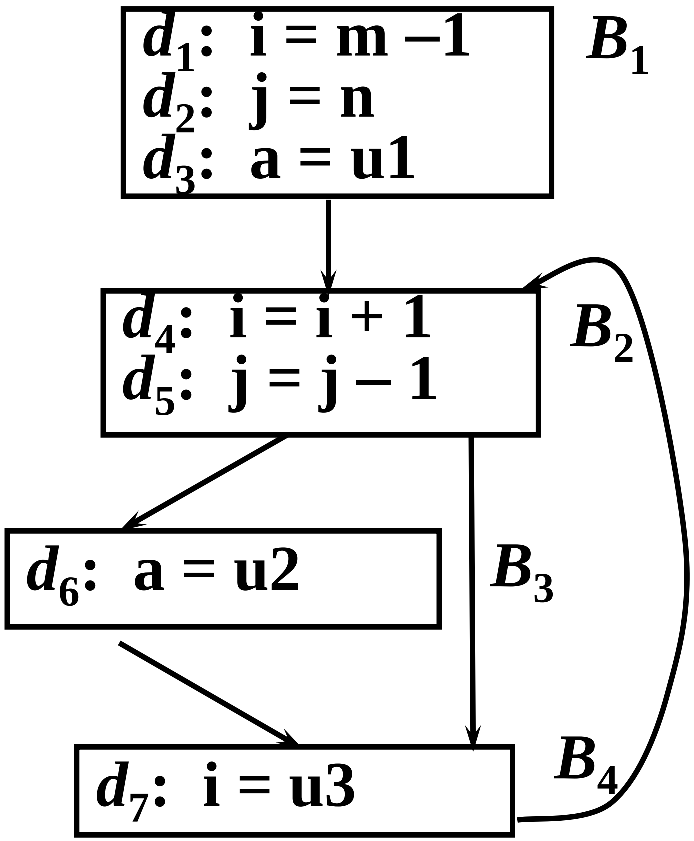
$$\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B)$$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0111	001 1110
B_3	001 1100	000 1110
B_4	001 1110	001 0111

不再继续演示迭代计算

$gen\ [B_1] = \{d_1, d_2, d_3\}$

$kill\ [B_1]=\{d_4, d_5, d_6, d_7\}$



B_1

d_1: i = m –1

d_2: j = n

d_3: a = u1

B_2

d_4: i = i + 1

d_5: j = j – 1

B_3

d_6: a = u2

B_4

d_7: i = u3

$gen\ [B_2] = \{d_4, d_5\}$

$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$

$kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$

$kill\ [B_4] = \{d_1, d_4\}$

2. 程序分析

- 控制流分析
- 数据流分析：到达-定值分析、数据流分析模式、活跃变量分析、可用表达式分析等

到达-定值计算

- 到达-定值数据流等式（方程）是**正向**的方程
  - $\text{OUT}[B] = gen[B] \cup (\text{IN}[B] - kill[B])$
  - $\text{IN}[B] = \bigcup_{P\text{是}B\text{的前驱}} \text{OUT}[P]$

  某些数据流等式是反向的
- 到达-定值数据流等式的**合流**运算是**求并集**
  - $\text{IN}[B] = \bigcup_{P\text{是}B\text{的前驱}} \text{OUT}[P]$

  某些数据流等式的合流运算是求交集
- 对到达-定值数据流等式，迭代求它的**最小解**

  某些数据流方程可能需要求最大解

数据流分析模式

□ 数据流值

- 数据流分析总把程序点和数据流值联系起来
- 数据流值代表在程序点能观测到的所有可能程序状态集合的一个抽象

- 语句*s*前后两点数据流值用IN[*s*]和OUT[*s*]来表示
- 数据流问题就是通过基于语句语义的约束（迁移函数）和基于控制流的约束来寻找所有语句*s*的IN[*s*]和OUT[*s*]的一个解

数据流分析模式

□ 迁移函数(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

- 正向: $\mathrm{IN}[s_{i+1}] = \mathrm{OUT}[s_i],\ i = 1, 2, \ldots, n-1$
  逆向: $\mathrm{OUT}[s_{i-1}] = \mathrm{IN}[s_i]$，$i = 2, 3, \ldots, n$
- 正向: $f_B = f_n \circ \ldots \circ f_2 \circ f_1$
  逆向： $f_B = f_1 \circ \ldots \circ f_{n-1} \circ f_n$
- 正向: $\mathrm{OUT}[B] = f_B\,(\mathrm{IN}[B])$
  逆向： $\mathrm{IN}[B] = f_B\,(\mathrm{OUT}[B])$

数据流分析模式

- 控制流约束
  - 正向传播

$$\mathrm{IN}[B] = \mathrm{Y}_{P是B的前驱}\mathrm{OUT}[P]$$

  - 逆向传播

$$\mathrm{OUT}[B] = \mathrm{Y}_{S是B的后继}\mathrm{IN}[S]$$

- 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

2. 程序分析

- □ 控制流分析
- □ 数据流分析：到达-定值分析、数据流分析模式、活跃变量分析、可用表达式分析等

活跃变量(live-variable)

□ 定义

- 变量 x的值在*p*点开始的某条执行路径上被引用，则说x在*p*点活跃，否则称x在*p*点已经死亡
- IN[*B*]：块*B*开始点的活跃变量集合
- OUT[*B*]：块*B*结束点的活跃变量集合
- use_B：块*B*中有引用且在引用前无定值的变量集
- def_B：块*B*中有定值的变量集

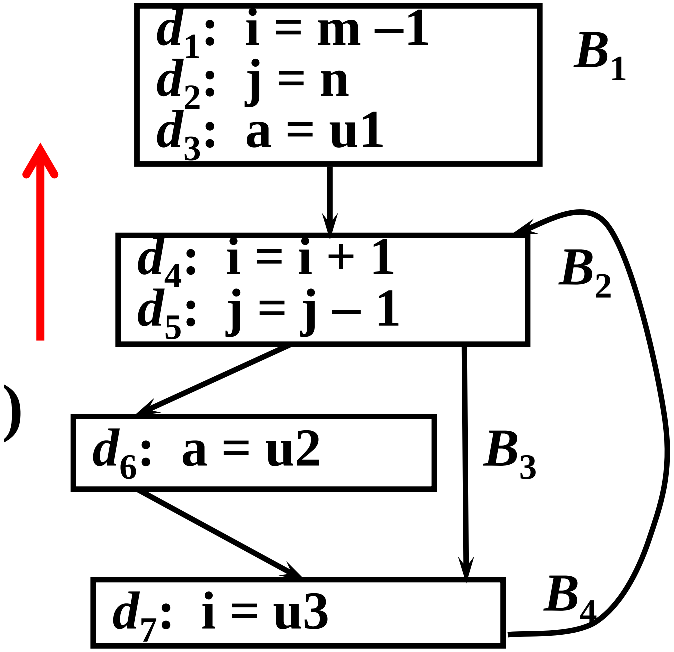
□ 应用

- 一种重要应用就是基本块的寄存器分配

活跃变量

- □ 例

  $use[B_2] = \{ \text{i, j} \}, def[B_2] = \{ \text{i, j} \}$



- □ 活跃变量数据流等式
  - ■ $\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$
  - ■ $\text{OUT}[B] = \text{Y}_{S\text{是}B\text{的后继}} \text{IN}[S]$
  - ■ $\text{IN}[\text{EXIT}] = \varnothing$
- □ 和到达-定值等式之间的联系与区别
  - ■ 汇合算符：都是集合并算符
  - ■ 信息流动方向相反，IN和OUT的作用相互交换
  - ■ *use*和*def*分别取代*gen*和*kill*
  - ■ 仍然需要最小解

定值-引用(Def-Use)

- **定值-引用链(du链)**
  - 设变量x有某定值d，该定值能到达的所有引用u的集合称为x在d处的定值-引用链，简称du链
- **通过活跃变量分析来计算du链**

  OUT[B]: 从B的末尾处能到达的引用集合

  - 如果B中x的定值d之后有x的第一个定值d′，则d和d′之间x的所有引用构成d的du链

$\cdots$

$d\colon x = \cdots$

$\cdots$

$\cdots = \cdots x \cdots$

$\cdots$

$d'\colon x = \cdots$

$\cdots$

  - 如果B中x的定值d之后无x的新的定值，则B中d之后x的所有引用以及OUT[B]中x的所有引用构成d的du链

$\cdots$

$d\colon x = \cdots$

$\cdots$

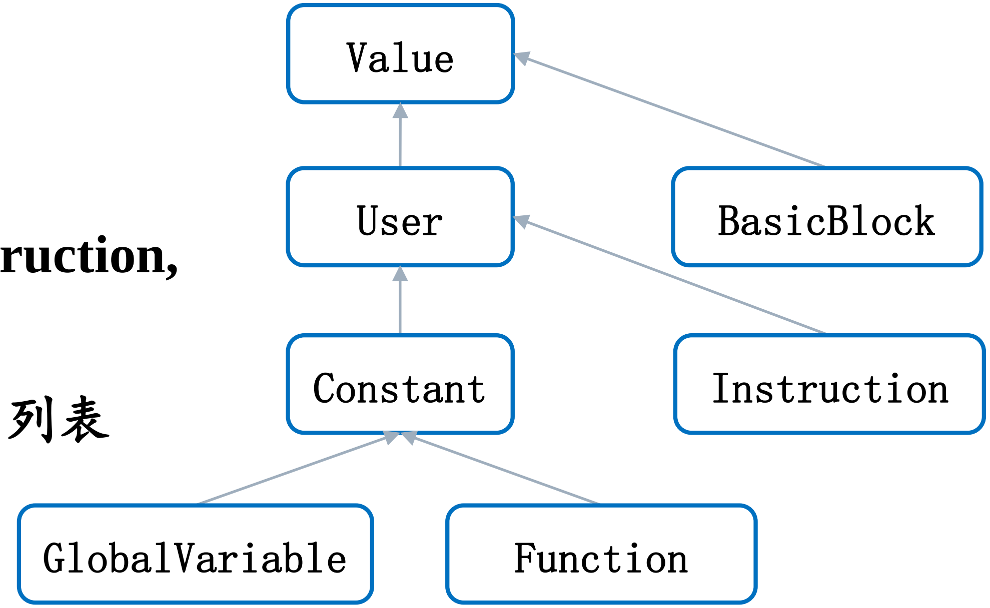
$\cdots = \cdots x \cdots$

$\cdots$

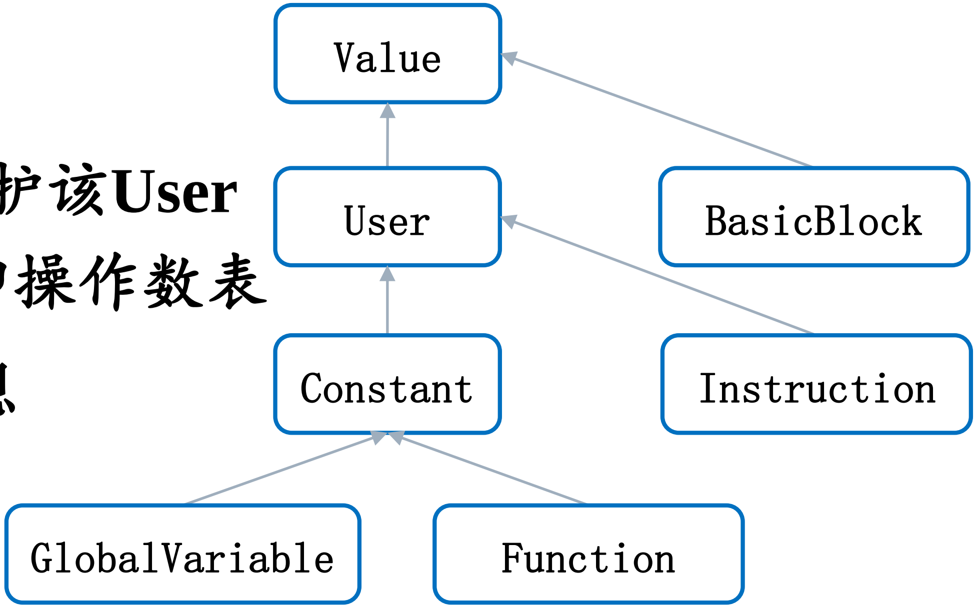
LLVM IR的UD和DU链

- Value类
  - 代表一个带类型的数据

    Constant, Argument, Instruction, Function 等
  - 维护一个该数据使用者的列表
    - 该表描述def-use信息
  - 主要方法
    - Type* getType(): 获取 Value 的类型
    - use 迭代器
      - Value::use_iterator/Value::const_use_iterator
      - use_size(), use_empty(), use_begin(), use_end()
    - use 替换： void replaceAllUsesWith(Value *V)



LLVM IR的UD和DU链

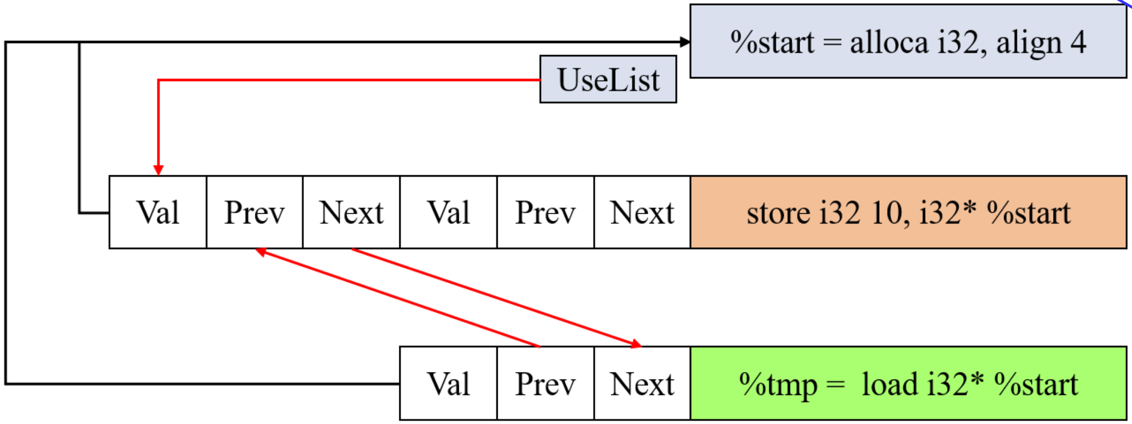


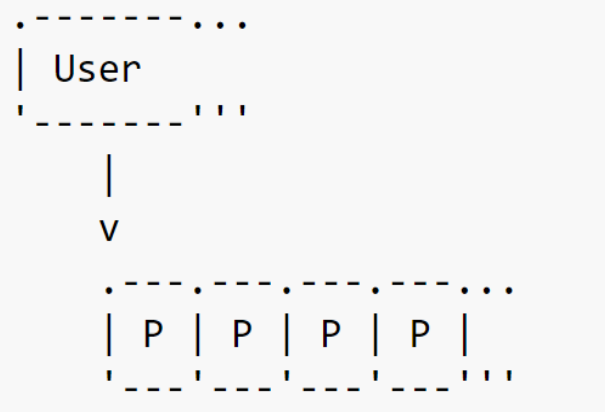
- □ **User类**
  - ■ **代表一个使用者，维护该User使用的所有Value，即操作数表**
    - □ **提供了use-def信息**
  - ■ **主要方法**
    - □ 获取操作数
      - ■ Value *getOperand(unsigned i)，获取第i个操作数
      - ■ unsigned getNumOperands()，获取操作数的个数
    - □ 操作数迭代器，User::op_iterator
      - ■ op_begin(), op_end()

LLVM IR的UD和DU链

- **Value、Use、User组成的双向链保存DU、UD关系**
  - □ 每个Value对象有一个Use ** UseList双向链表，维护所有的使用者
  - □ 每个Use对象代表一条从User→Value的边
  - □ 每个User对象维护一个Value列表
    - 固定数目操作数的User
    - 可变数目操作数的User

```
...---.---.---.---.-------...
  | P | P | P | P | User
'''---'---'---'---'-------'''
```





```
.-------...
| User
'-------'''
    |
    v
    .---.---.---.---...
    | P | P | P | P |
    '---'---'---'---'''
```

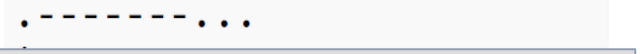
LLVM IR的UD和DU链

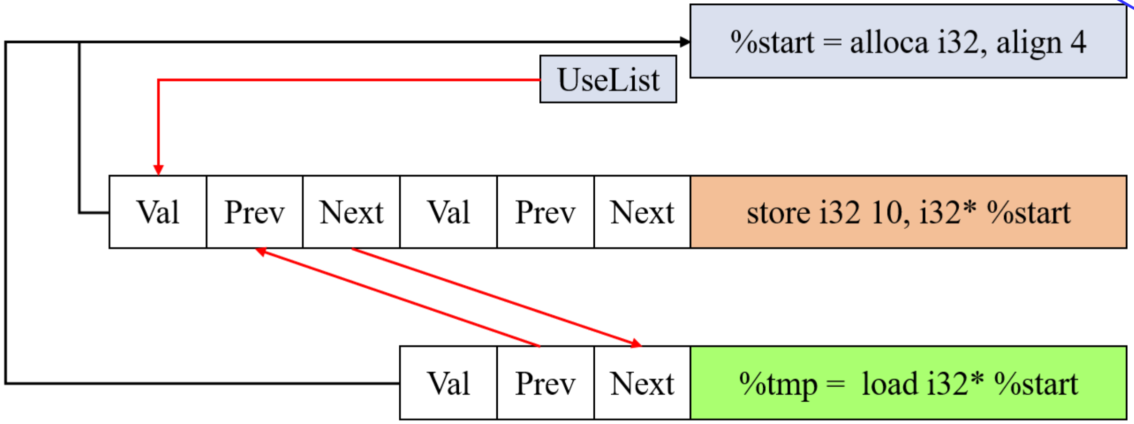
- Value、Use、User组成的双向链保存DU、UD关系
  - □ 每个Value对象有一个Use ** UseList双向链表，维护所有的使用者
  - □ 每个Use对象代表一条从User→
  - □ 每个User对象维护一个Value列
    - 固定数目操作数的User
    - 可变数目操作数的User

遍历使用者

```
Function *F = ...;

for (User *U : F->users())
  if (Instruction *Inst =
dyn_cast<Instruction>(U)) {...}
```





遍历操作数

```
Instruction *pi = ...;

for (Use &U : pi->operands()) {
  Value *v = U.get();
  // ...
}
```

2. 程序分析

- 控制流分析
- 数据流分析：到达-定值分析、数据流分析模式、活跃变量分析、可用表达式分析等

可用表达式

□ 可用表达式(available expressions)

$x = y + z$

…

…

…

p

没有对y和z的定值

$y + z$ 在p点可用

$x = y + z$

…

$y = …$

…

p

$y + z$ 在p点不可用

$x = y + z$

…

$z = …$

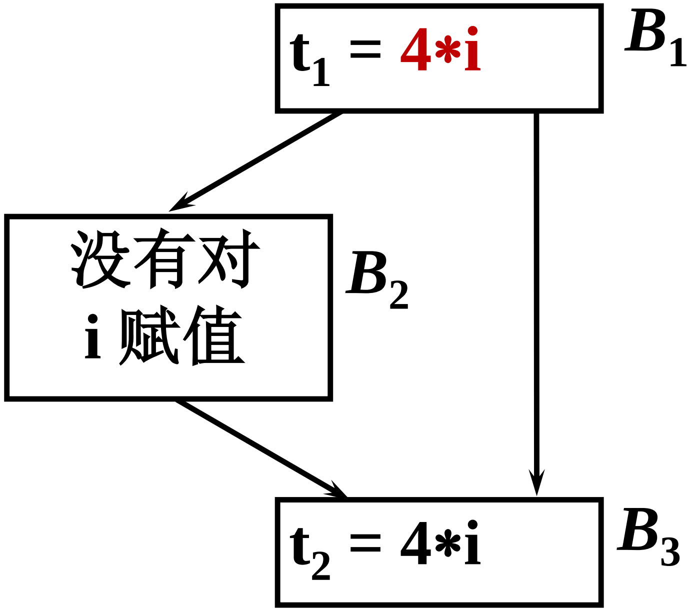
…

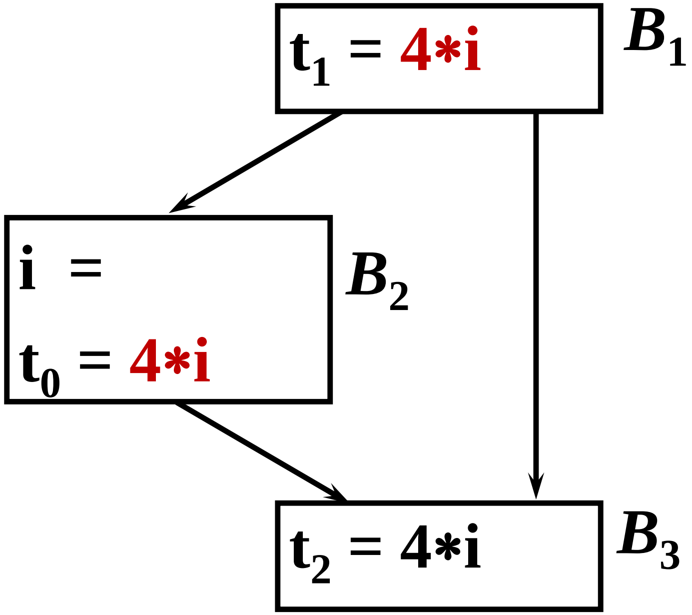
p

$y + z$ 在p点不可用

可用表达式

下面两种情况下， 4*i 在B_3的入口都可用





可用表达式

□ 定义

- 若到点*p*的每条执行路径都计算x + y，并且计算后没有对x或y赋值，那么称x + y在点*p*可用
- e_gen_B：块*B*产生的可用表达式集合
- e_kill_B：块*B*注销的可用表达式集合
- IN [*B*]：块*B*入口的可用表达式集合
- OUT [*B*]：块*B*出口的可用表达式集合

□ 应用

- 公共子表达式删除

可用表达式

- 数据流等式
  - $\mathrm{OUT}\,[B] = e\_gen_B \cup (\mathrm{IN}\,[B] - e\_kill_B\,)$
  - $\mathrm{IN}\,[B] = \mathrm{Y}_{P是B的前驱}\ \mathrm{OUT}\,[P]$
  - $\mathrm{IN}\,[\mathrm{ENTRY}] = \varnothing$
- 同先前的主要区别
  - 汇合算符：使用∩而不是∪
  - 求最大解而不是最小解

值编码Value Numbering

- □ Value Numbering
  - Balke 1968 or Ershov 1954
  - 早期面向低级的线性IR（如三地址码）
  - 现在也存在面向树或图的等价方法
- □ 优化的范围
  - 基本块：Local value numbering
  - 扩展的基本块：Superlocal value numbering (SVN)
  - 支配关系：基于支配的值编码 (DVN)

Local Value Numbering

□ 每个值不同的表达式有一个值编号

Original Code	With VNs	Hash Table for VN
a ← x + y	$a^3 \leftarrow x^1 + y^2$	{<x,1>, <y,2>, <<+,1,2>,3>, <a,3>}
z ← y	$z^2 \leftarrow y^2$	{..., <z,2>}
d ← 17	$d^4 \leftarrow 17$	{..., <z,2>, <17,4>, <d,4>}
c ← x + z	$c^3 \leftarrow x^1 + z^2$	{..., <z,2>, <17,4>, <d,4>, <c,3>}

□ 冗余消除（Redundancy Elimination）

Rewritten	Hash Table for Rewritten
a ← x + y	{<1,x>, <2,y>, <3,a>}
z ← y	{<1,x>, <2,y>, <3,a>}
d ← 17	{<1,x>, <2,y>, <3,a>, <4,17>}
c ← a	{<1,x>, <2,y>, <3,a>, <4,17>}

值编号作为key，value为存放值的变量或者常量

LVN：解决重写问题

Original Code	With VNs	Rewritten
$a \leftarrow x + y$	$a^3 \leftarrow x^1 + y^2$	$a \leftarrow x + y$
$z \leftarrow y$	$z^2 \leftarrow y^2$	$z \leftarrow y$
* $b \leftarrow x + y$	* $b^3 \leftarrow x^1 + y^2$	* $b \leftarrow a$
$a \leftarrow 17$	$a^4 \leftarrow 17$	a ~~← 17~~
* $c \leftarrow x + y$	* $c^3 \leftarrow x^1 + y^2$	* c ~~← a~~

Two redundancies marked by *

Hash Table for Rewritten

{<1,x>, <2,y>, <3,a>}
{<1,x>, <2,y>, <3,a>}
{<1,x>, <2,y>, <3,a>, <4,17>}
{<1,x>, <2,y>, <3,a>, <4,17>}

Optional Solutions:
- Use $c^3 \leftarrow b^3$
- Save a^3 in t^3
- Rename around it (best)

根据引用的值编号来查找值并计算，记录当前定值所关联的值编号到值的映射
对a重写时，并不去查询value为a的pair

LVN：值编码的重命名

- 重命名：给每个值唯一的名字 => SSA

Original Code	**With VNs**	**Rewritten**
$a_0 \leftarrow x_0 + y_0$	$a_0^3 \leftarrow x_0^1 + y_0^2$	$a_0 \leftarrow x_0 + y_0$
$z_0 \leftarrow y_0$	$z^2_0 \leftarrow y_0^2$	$z_0 \leftarrow y_0$
* $b_0 \leftarrow x_0 + y_0$	* $b_0^3 \leftarrow x_0^1 + y_0^2$	* $b_0 \leftarrow a_0$
$a_1 \leftarrow 17$	$a_1^4 \leftarrow 17$	$a_1 \leftarrow 17$
* $c_0 \leftarrow x_0 + y_0$	* $c_0^3 \leftarrow x_0^1 + y_0^2$	* $c_0 \leftarrow a_0$

Hash Table for Rewritten

$\{<1,x_0>, <2,y_0>, <3,a_0>\}$

$\{<1,x_0>, <2,y_0>, <3,a_0>\}$

$\{<1,x_0>, <2,y_0>, <3,a_0>, <4,17>\}$

$\{<1,x_0>, <2,y_0>, <3,a_0>, <4,17>\}$

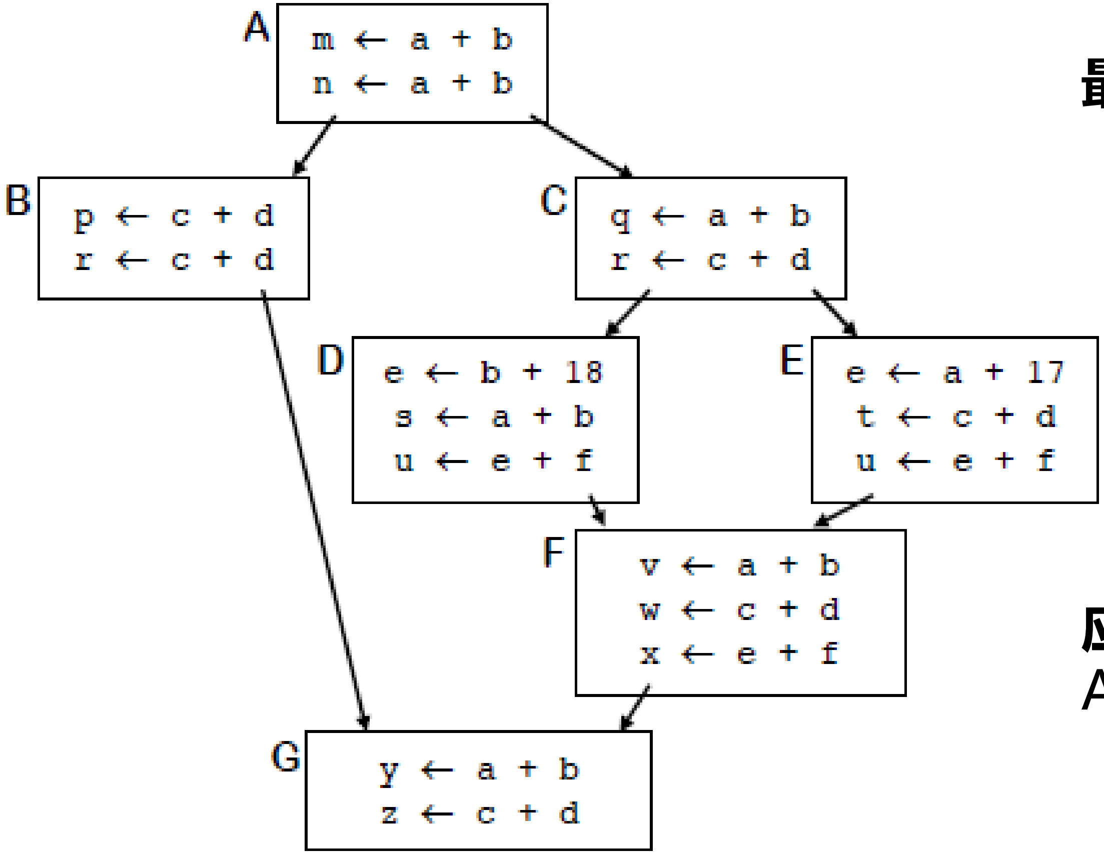
Result:

- a_0 is available
- Rewriting just works

Superlocal Value Numbering

□ 扩展的基本块EBB

- B1, B2,…Bn，2<= i <= n, Bi有唯一的前驱



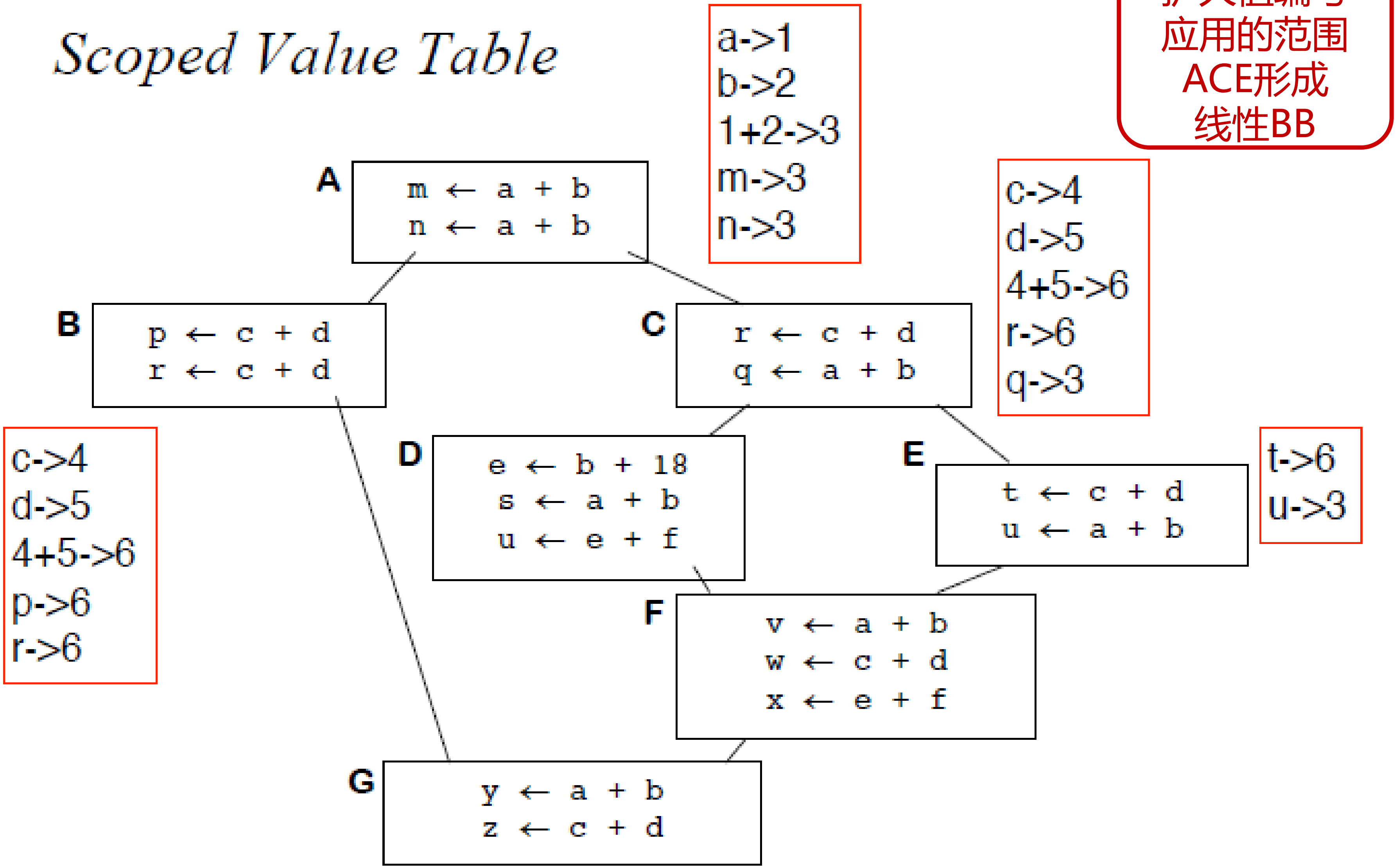
最大的EBB：ABCDE、F、G

应用局部化到EBB的路径：
AB、ACD、ACE、F、G

Superlocal Value Numbering

扩大值编号
应用的范围
ACE形成
线性BB

Scoped Value Table



A

```
m ← a + b
n ← a + b
```

a->1
b->2
1+2->3
m->3
n->3

B

```
p ← c + d
r ← c + d
```

c->4
d->5
4+5->6
p->6
r->6

C

```
r ← c + d
q ← a + b
```

c->4
d->5
4+5->6
r->6
q->3

D

```
e ← b + 18
s ← a + b
u ← e + f
```

E

```
t ← c + d
u ← a + b
```

t->6
u->3

F

```
v ← a + b
w ← c + d
x ← e + f
```

G

```
y ← a + b
z ← c + d
```

数据流问题小结

见书上表9.2

□ 数据流问题的组成

- 数据流值的论域、数据流的方向、迁移函数、边界条件、汇合算符、数据流等式

	数据流值的定义域	方向	应用
到达-定值	定值点集合	正向	用于DCE的def-use链
活跃变量	变量集合	反向	寄存器分配、未初始化检测、SSA构造、无用存储消除等
可用表达式	表达式集合	正向	CSE
非常忙表达式	表达式集合	反向	循环外提
常量传播	<v,c>集合	正向	常量折叠

流敏感(flow-sensitivity)

- **流不敏感分析（flow-insensitive analysis）**
  - 不考虑程序中语句执行的顺序.
  - 把程序中语句随意交换位置（即：改变控制流），如果分析结果始终不变，则该分析为流非敏感分析。

前面的数据流分析算法中(4)没有规定对基本块的操作次序

- **流敏感分析（flow-sensitive analysis）**

考虑程序中过程内的控制流情况（顺序、分支、循环）

上下文敏感、域敏感、路径敏感

- **上下文不敏感分析Context-insensitive analysis**
  - 在过程调用的时候忽略调用的上下文
- **上下文敏感分析Context-sensitive analysis**
  - 考虑过程调用的上下文(调用点的实参、返回值的使用)
- **域(不)敏感分析field-sensitive analysis**
  - 分析中是否考虑结构体中的不同域、数组中的不同下标元素
- **路径(不)敏感分析path-sensitive analysis**
  - 是否依据分支语句的不同谓词来计算不同的分析信息

3. 符号执行

静态分析

□ 能分析程序中所有可能的运行

- 有许多有趣的想法和工具
- 很多只是在论文上展示有好的效果
- 学术界推出的被企业认可的商用工具寥寥无几

Dawson Engler: coverity [CACM2010]

但是，开发者使用起来….

- 不容易，论文中的结果描述的是静态分析专家所用的
- 有生命力的商用工具：要能处理误报(**false positives**)、开发者的困惑、错误管理、…

[CACM2010] A few billion lines of code later: using static analysis to find bugs in the real world, Communication of the ACM, 53(2):66-75, 2010.

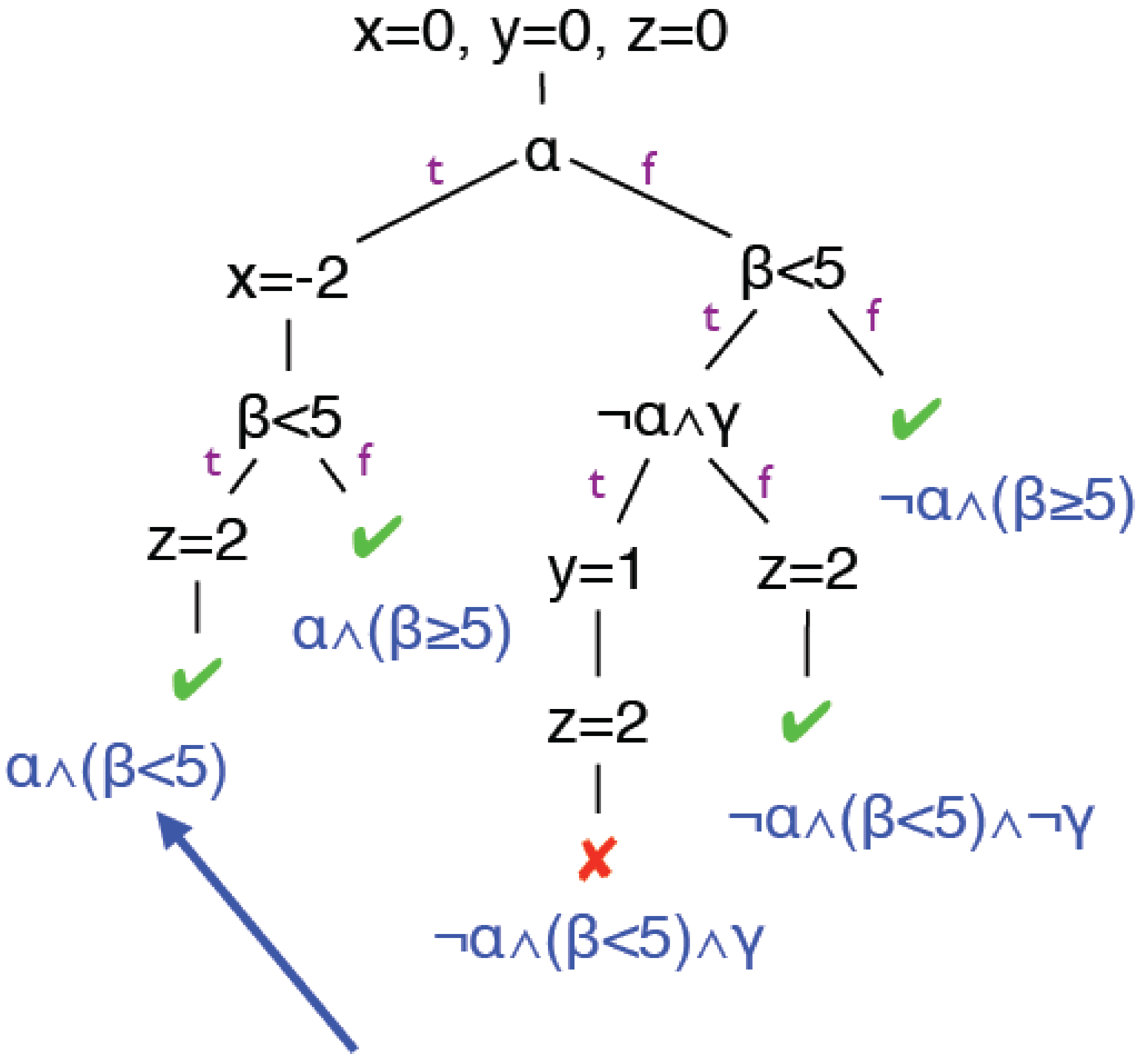
符号执行的引入

- □ 抽象的作用
  - ■ 让静态分析对所有可能的运行进行建模，但引入保守性
  - ■ *-敏感性方法试图改进之，但是远远不够
- □ 静态分析的抽象 ≠ 开发者的抽象
- □ 测试
  - ■ 每个测试只能考察一种可能的执行
  - ■ 希望测试用例更具有一般性，但是却没有保障
- □ 符号执行：对测试的泛化，引入符号值来计算

[计算机学报2015] 软件安全漏洞检测技术，38(4):717-732, 2015.

符号执行举例

```
int a = α, b = β, c = γ;
// symbolic
int x = 0, y = 0, z = 0;
if (a) {
    x = -2;
}
if (b < 5) {
     if (!a && c) { y = 1; }
     z = 2;
}
assert(x+y+z!=3)
```



path condition

符号执行

- □ 每个符号执行路径代表多个实际的程序运行
  - ■ 实际运行的具体值满足符号执行的路径条件

相比测试，符号执行可以覆盖更多的程序执行空间

- □ 符号执行的发展
  - ■ 早期的工作：**James C. King. Symbolic execution and program testing. CACM, 19(7):385–394, 1976. (高引)**
  - ■ 问题：**1980’s** 当时的机器内存小且慢
  - ■ 符号执行代价极高：大量可能的执行路径，需要求解器判断哪些路径是可行的，程序状态有许多位

符号执行现状

- 计算机：速度快、内存便宜
- 有非常强大的SMT/SAT求解器
  - SMT= Satisfiability Modulo Theories =SAT++
  - 快速解决非常多的问题：检查断言、削减可行路径
  - 求解器：Z3(已集成到LLVM中)、STP、Yices等
- 近10年来的有代表性论文
  - KLEE（OSDI2008, C. Cadar、D.Engler等,用于LLVM）
  - [CACM2013] Symbolic execution for software testing(30y+)
  - [CSUR2018] A Survey of Symbolic Execution Techniques

SMT求解器的性能

- SAT求解器是SMT求解器的核心
  - SMT(可满足性模理论)：接受不同格式的等式系统
  - SAT：必须是合取CNF范式的布尔等式
  - 理论上，所有SMT查询可以约减到SAT查询
  - 实践上，SMT和高级优化是关键的

  - 简单的等式 x+0=x
  - 数组理论： read(x, write(42, x, A)) =42
    - x是下标，A是数组，write(42, x, A)表示A[x]=42后的数组
  - 缓存：记住求解器的查询； 删除无用变量; …

符号执行重生

- 两个关键的系统
  - DART (Godefroid and Sen, PLDI 2005)
  - EXE (Cadar, Ganesh, Pawlowski, Dill, and Engler， CCS2006)
- SAGE：Microsoft的concolic执行器(动态符号执行)
- Mayhem(CMU), Angr(UCSB), Triton
- Java Symbolic PathFinder
- KLEE

简单命令式语言IMP的符号执行

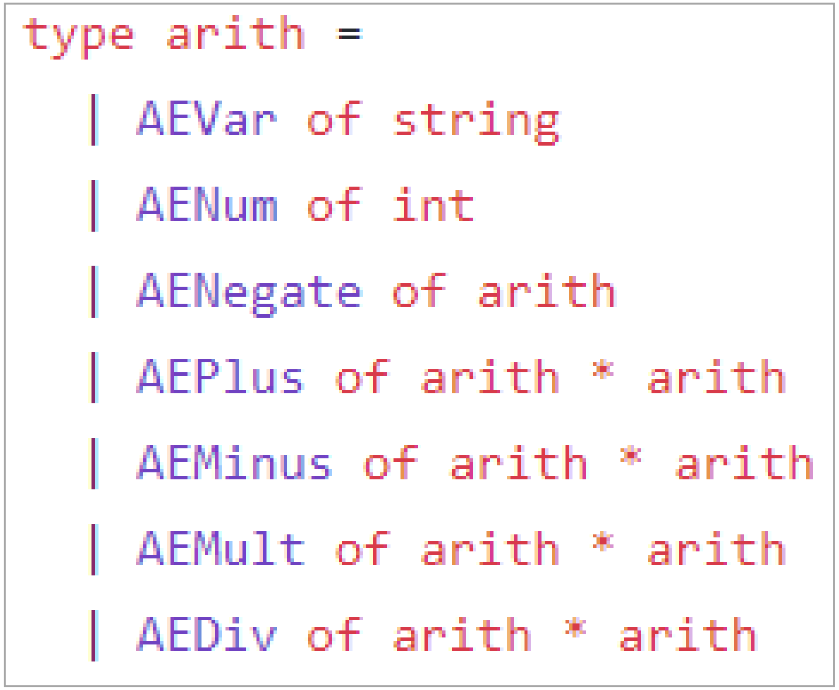
```
a ::= n | X | -a | a0+a1 | a0-a1 | a0×a1 | a0/a1
b ::= bv | ¬b | b0∧b1 | b0∨b1 | a0=a1 | a0<a1 | a0>a1
c ::= skip | input(s) | X:=a | if b then c else c
      | c0; c1 | while b do c | assert b
```

- n是整数、X是变量、bv是布尔值
- c是命令、a和b分别是整型和布尔型表达式
- Sym-while的OCaml实现：https://github.com/Isweet/sym-while
  - syswhile.ml 包含总控程序，即main，具体执行或符号执行
  - lexer.mll(词法描述)、parser.mly(语法描述)、ast.ml(AST)
  - concrete.ml: 解释执行，状态是变量到整数的映射，Imp.run s
  - symbol.ml, symbolic.ml: 符号执行，状态时变量到符号表达式和路径条件的映射

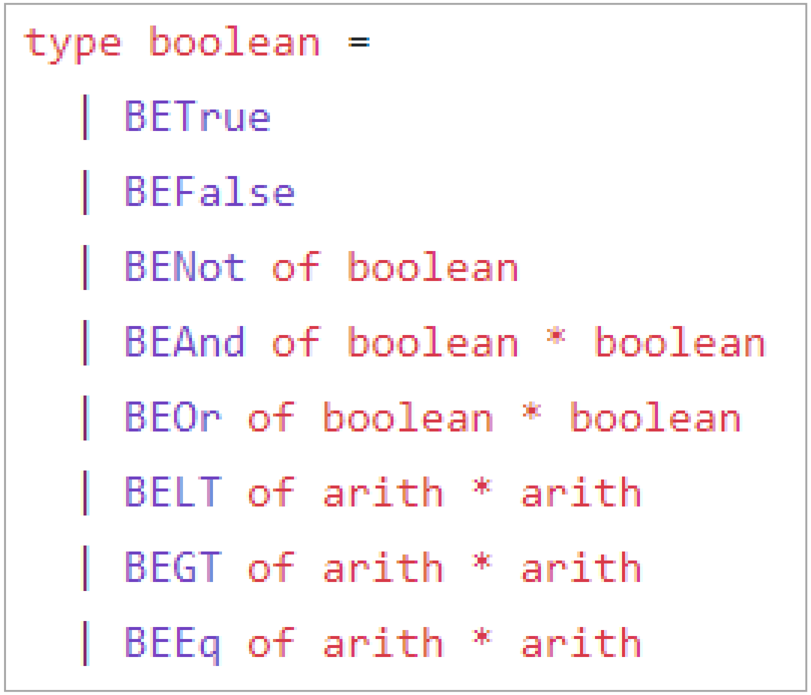
符号表达式

□ 符号表达式可能包含变量

ast.ml



```
type arith =
    | AEVar of string
    | AENum of int
    | AENegate of arith
    | AEPlus of arith * arith
    | AEMinus of arith * arith
    | AEMult of arith * arith
    | AEDiv of arith * arith
```



```
type boolean =
    | BETrue
    | BEFalse
    | BENot of boolean
    | BEAnd of boolean * boolean
    | BEOr of boolean * boolean
    | BELT of arith * arith
    | BEGT of arith * arith
    | BEEq of arith * arith
```

符号状态

- **具体状态：** 变量到整数

[concrete.ml]

```
type conc_state = int StringMap.t
```

- **符号状态：** 变量到符号表达式和路径条件 [symbolic.ml]

```
type sym_state = (Symbol.int_t StringMap.t) * Symbol.t
```

  - **Symbol.t:**布尔型符号表达式类型
  - **Symbol.int_t:**整型符号表达式类型

基于fork的符号执行

- 如何判断哪个分支可行(feasible)？

```
let rec eval (s : stmt) (s_st : sym_state) : answer =
  let (env, pc) = s_st in
```

```
| SIf (b, s1, s2) ->
  let l = t_of_boolean b env in           (* branch cond *)
  let cond_true = LAnd (l, pc) in         (* ... and path cond *)
  let cond_false = LAnd ((LNot l), pc) in
  let sat_true = check (z3_of_t cond_true) in
  let sat_false = check (z3_of_t cond_false) in
(match sat_true with                      (* might do both branches *)
   | Some _ -> ...(eval s1 (env, cond_true))
   | None -> ...);
  (match sat_false with
   | Some _ -> ...(eval s2 (env, cond_false))
   | None -> ...);
```

符号执行策略

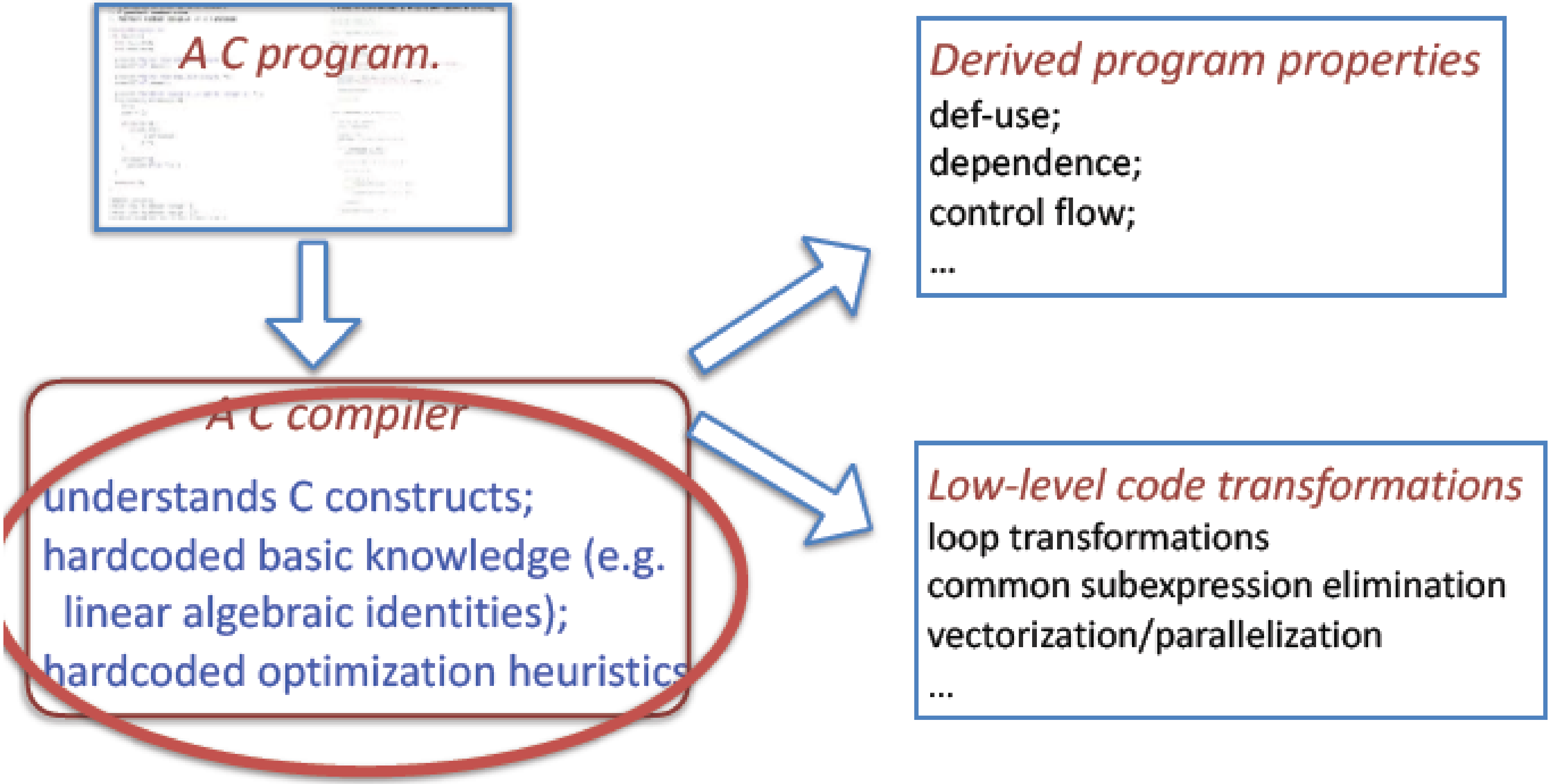
- □ **顶层策略**
  - 初始化状态：pc=0, 路径条件为true，状态为empty
  - 对每条语句符号求值
  - 一旦执行分叉，则对两个分支都分别求值（DepthFS）
  - 执行完后，返回多个符号状态
- □ **路径爆炸(path explosion): 分支、循环**
- □ **搜索策略**

DFS（容易在某部分stuck）、BFS; 树→图

优先权搜索、随机搜索、覆盖引导的启发式搜索、分代…

4. 面向AI的优化

传统的优化编译器



A C program.

A C compiler

understands C constructs;

hardcoded basic knowledge (e.g. linear algebraic identities);

hardcoded optimization heuristics

Derived program properties

def-use;

dependence;

control flow;

...

Low-level code transformations

loop transformations

common subexpression elimination

vectorization/parallelization

...

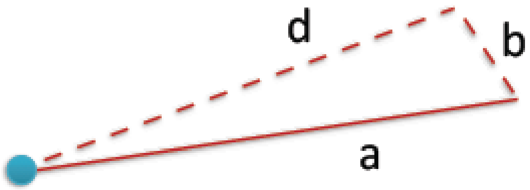
高级语义驱动的编译

- 高级语义
  - DSL：如矩阵运算 b= a+c
  - 泛化的编译技术
    - 强度削弱
    - 循环冗余消除

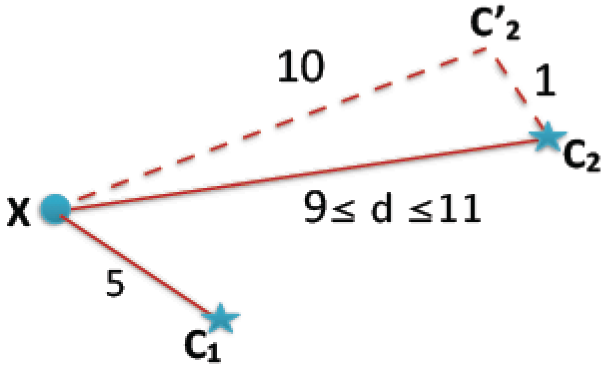
高级语义驱动的编译

- 强度削弱

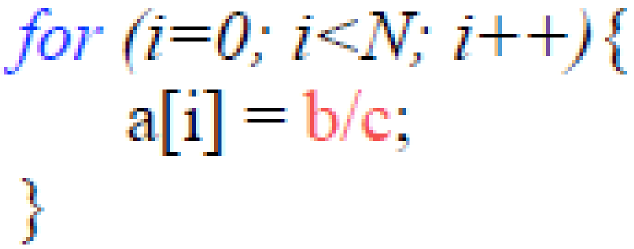
传统：b/2 ➔ b>>1



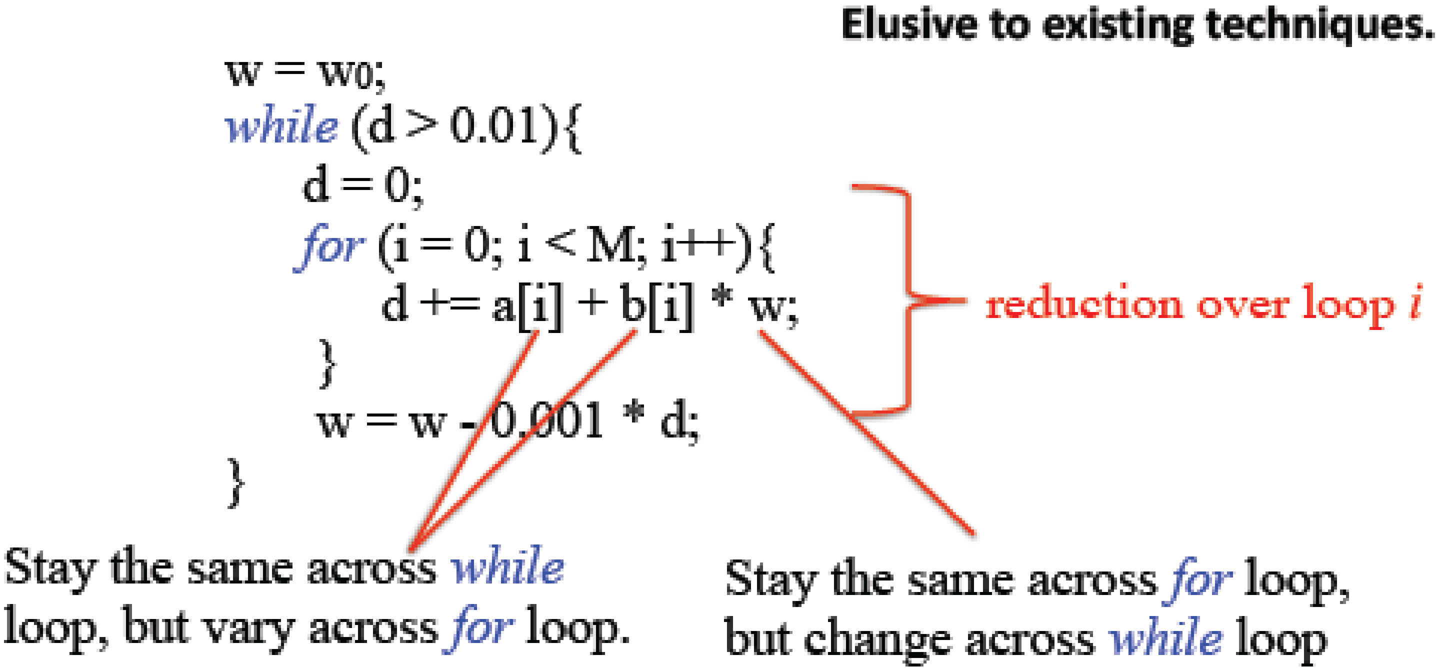
$$a\text{-}b \le d \le a+b$$



循环冗余



```
for (i=0; i<N; i++){
      a[i] = b/c;
}
```



Elusive to existing techniques.

```
w = w0;
while (d > 0.01){
    d = 0;
    for (i = 0; i < M; i++){
        d += a[i] + b[i] * w;
    }
    w = w - 0.001 * d;
}
```

reduction over loop *i*

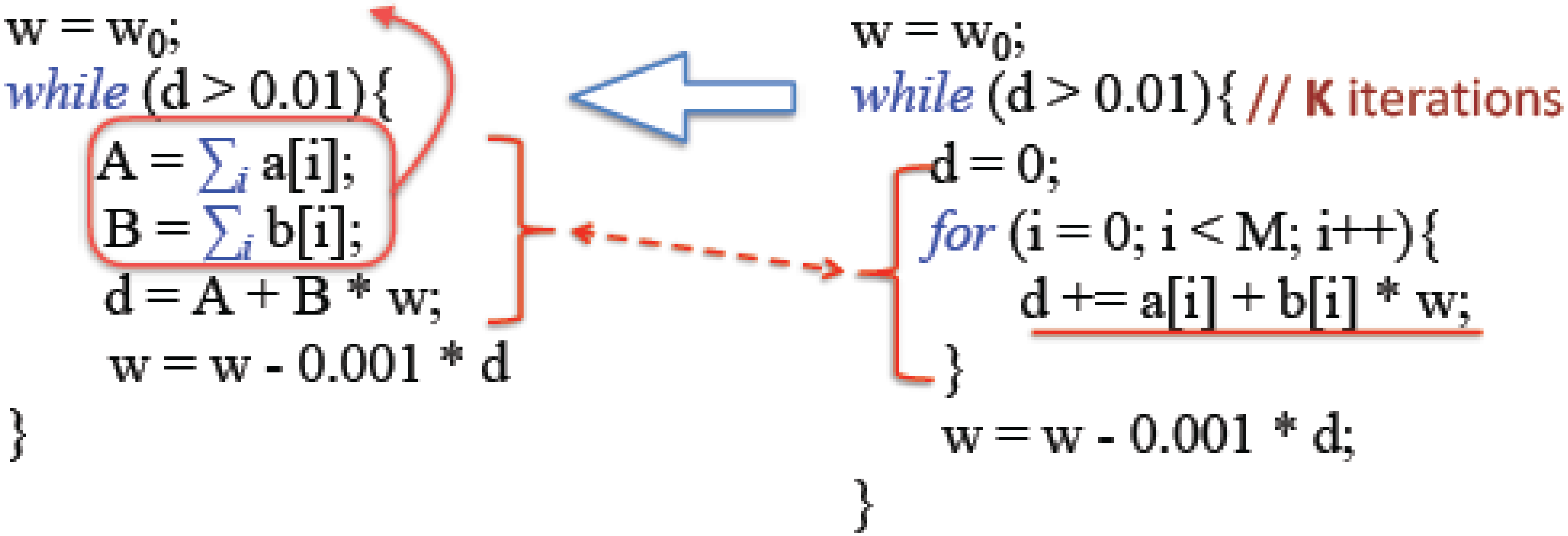
Stay the same across *while* loop, but vary across *for* loop.

Stay the same across *for* loop, but change across *while* loop

循环冗余

- An equivalent form of the example.

O(M*K) → O(M+K)



```
$w = w_0$;
while (d > 0.01){
    A = $\sum_i$ a[i];
    B = $\sum_i$ b[i];
    d = A + B * w;
    w = w - 0.001 * d
}
```

```
$w = w_0$;
while (d > 0.01){ // K iterations
    d = 0;
    for (i = 0; i < M; i++){
        d += a[i] + b[i] * w;
    }
    w = w - 0.001 * d;
}
```

5. 数据流分析的基础

数据流分析框架

- □ 数据流分析框架 $(D, V, \wedge, F)$包括
  - 数据流分析的**方向**D，它可以是正向或逆向
  - **数据流值的论域**：半格V、**汇合算子**$\wedge$
  - V到V的**迁移函数族**F,包括适用于边界条件（ENTRY和EXIT结点）的常函数
- □ 半格$(V, \wedge)$
  - 是一个集合V和一个二元交运算(汇合运算) $\wedge$，满足：
  - 幂等性：对所有的x，$x \wedge x = x$
  - 交换性：对所有的x和y，$x \wedge y = y \wedge x$
  - 结合性：对所有的x, y和z，$x \wedge (y \wedge z) = (x \wedge y) \wedge z$

半格(semilattices)

- □ **半格有顶元⊤ (最大元素),可以还有底元⊥(最小元素)**
  - ■ 对V中的所有x， $\top \wedge x = x$
  - ■ 对V中的所有x，$\bot \wedge x = \bot$
- □ **偏序关系：集合V上的关系$\leqslant$**
  - ■ 自反性：对所有的x，$x \leqslant x$
  - ■ 反对称性：对所有的x和y，如果$x \leqslant y$且$y \leqslant x$,那么$x = y$
  - ■ 传递性：对所有的x, y和z，如果$x \leqslant y$且$y \leqslant z$,那么$x \leqslant z$

此外，关系$<$的定义

$$x < y \text{当且仅当} (x \leqslant y) \text{并且} (x \neq y)$$

半格

□ 半格和偏序关系之间的联系

- 半格$(V, \wedge)$的汇合运算$\wedge$确定了半格值集V上一种偏序$\leqslant$：

  对V中所有的x和y，$x \leqslant y$当且仅当$x \wedge y = x$
- 若$x \wedge y$等于g，则g就是x和y的最大下界

例 半格的论域V是先前全域U的幂集

- 汇合运算为集合并：$\varnothing$是顶元, U是底元, 偏序关系是$\supseteq$
- 汇合运算为集合交：U是顶元, $\varnothing$是底元, 偏序关系是$\subseteq$
- 按偏序$\leqslant$意义上的最大解是最精确的

  (1) 到达-定值：最精确的解含最少定值

  (2) 可用表达式：最精确的解含最多表达式

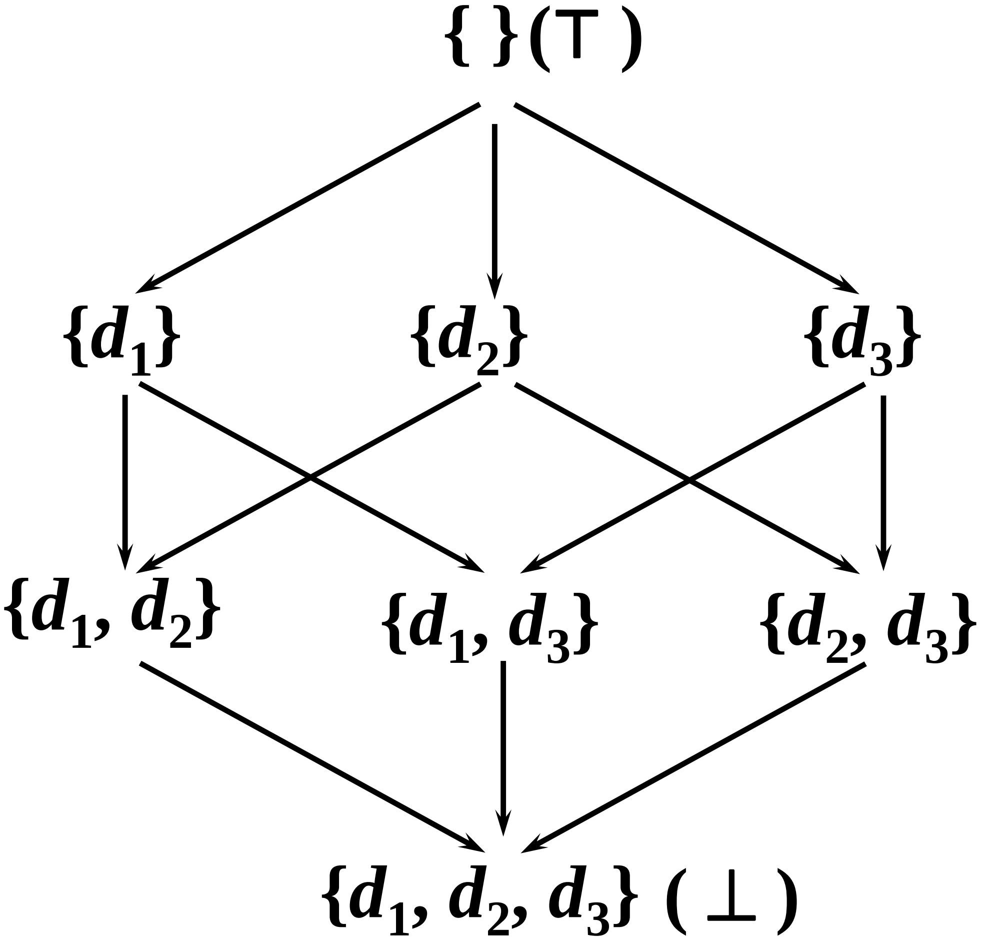
格图

□ 格图

- 结点是V中的元素
- 如果y ≤ x, 则有从x朝下到y的有向边

右图是定值子集之间形成的格:

- 到达-定值的≤是⊇
- x ∧ y的最大下界是x ∪ y



如何降低格图的规模?

- □ **到达-定值格图存在的问题**
  - 数据流值的集合是定值集合的幂集
    =>格图结点数随变量数呈指数级增长
  - 每个变量的定值可达性独立于其他变量的定值可达性

定值半格表示为从每个变量的简单定值半格构造出的积半格

- □ **积半格(假定$(A, \wedge_A)$和$(B, \wedge_B)$是半格)**
  - 论域是$A \times B$
  - 汇合运算∧: $(a, b) \wedge (a', b') = (a \wedge_A a', b \wedge_B b')$

数据流分析算法的收敛速度

□ 半格的高度

- 偏序集合$(V, \leq)$中的一个上升链是序列$x_1 < x_2 < \ldots < x_n$
- **半格的高度**就是其中最长上升链中$<$的个数

例，在一个有n个定值的程序中，到达-定值的高度是n

□ 数据流分析算法的收敛

- 半格的高度有限 => 数据流分析迭代算法收敛
- 半格的值论域有限 => 半格的高度有限
- 半格的值论域无限 => 半格的高度可能有限

  如，常量传播算法中使用的半格

迁移函数

□ 迁移函数族$F: V \to V$有下列性质

- F包括恒等函数 I，即对V中所有的x，有$I(x) = x$
- F封闭于复合，即对F中任意两个函数f和g，$g \circ f \in F$
- 若F中所有函数f都有单调性，即

  $x \leqslant y$蕴涵 $f(x) \leqslant f(y)$，或 $f(x \wedge y) \leqslant f(x) \wedge f(y)$

  则称框架$(D, V, \wedge, F)$是单调的
- 框架$(D, V, \wedge, F)$的分配性

  对F中所有的f，$f(x \wedge y) = f(x) \wedge f(y)$

框架单调=> 所求得的解是数据流方程组的最大不动点

迁移函数

□ 例 到达-定值分析

若$f_1(x) = G_1 \cup (x - K_1)$，$f_2(x) = G_2 \cup (x - K_2)$

- 若G和K是空集，则f是恒等函数
- $f_2(f_1(x)) = G_2 \cup ((G_1 \cup (x - K_1)) - K_2)$

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

  因此f_1和f_2的复合f为$f = G \cup (x - K)$的形式
- 分配性可以由检查下面的条件得到

$$G \cup ((y \cup z) - K) = (G \cup (y - K)) \cup (G \cup (z - K))$$

分配性： $f(y \wedge z) = f(y) \wedge f(z)$

一般框架的迭代算法

- 以正向数据流分析为例

```
OUT[ENTRY] = $v_{ENTRY}$;
for (除了ENTRY以外的每个块B)  OUT[B] = $\top$;
while (任何一个OUT出现变化)
    for (除了ENTRY以外的每个块B) {
        IN[B] = $\wedge_{P是B的前驱}$ OUT[P];
        OUT[B] = $f_B$(IN[B]);
    }
```

数据流解的含义

- 结论：算法所得解是理想解的稳妥近似
- 理想解所考虑的路径
  - 执行路径集：流图上每一条路径都属于该集合

    若流图有环，则执行路径数是无限的
  - 程序可能的执行路径集：程序执行所走的路径属于该集合 —— 这是理想解所考虑的路径集
  - 可能的执行路径集 $\subseteq$ 执行路径集
  - 寻找所有可能执行路径是不可判定的
- 以下讨论以正向数据流分析为例

理想解

理想解

若路径$P = \text{ENTRY} \to B_1 \to B_2 \to \ \dots \to B_k$，定义

- $f_P = f_{k-1} \circ \dots \circ f_2 \circ f_1$
- $\text{IDEAL}[B] = \wedge_{P\text{是从ENTRY到}B\text{的一条可能路径}} f_P(v_{\text{ENTRY}})$

有关理解解的结论

- 任何大于理想解IDEAL的回答一定是不对的
- 任何小于或等于IDEAL的值是稳妥的
- 在稳妥的值中，越接近IDEAL的值越精确

MFP和MOP

- □ **MFP最大不动点解**

  maximal fixed point

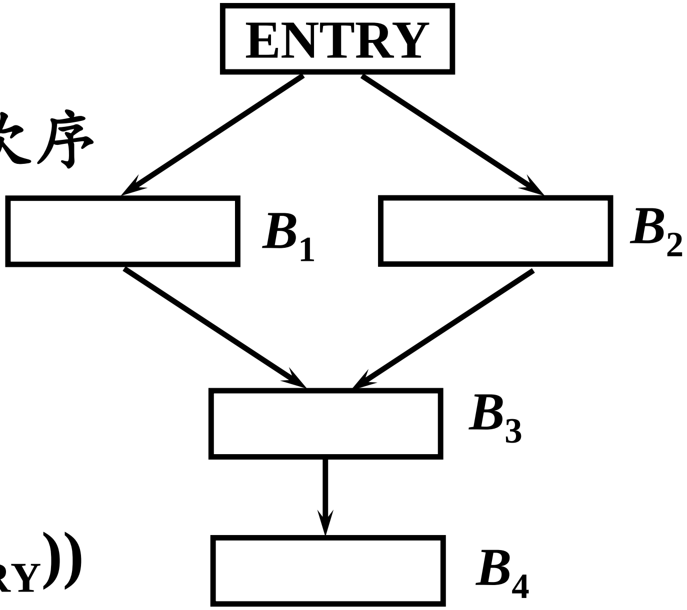
  - 访问每个基本块（不一定按照程序执行时的次序）
  - 在每个汇合点，把汇合运算作用到当前得到的数据流值，所用的一些初值是人工引入的

- □ **MOP执行路径上的解**

  meet over paths

  - $\mathrm{MOP}[B] = \wedge_{P\text{是从ENTRY到}B\text{的一条路径}} f_P(v_{\mathrm{ENTRY}})$
  - MOP解汇集了所有可能路径的数据流值，包括那些不可能被执行路径的数据流值
  - 对所有的块*B*，$\mathrm{MOP}[B] \preccurlyeq \mathrm{IDEAL}[B]$

MFP与MOP的联系



- MFP与MOP的联系
  - MFP访问基本块未必遵循执行次序

    由各块的初值和迁移函数的

    单调性保证结果一致
  - MFP较早地使用汇合运算

$$\mathrm{IN}[B_4] = f_3(f_1(v_{\mathrm{ENTRY}}) \wedge f_2(v_{\mathrm{ENTRY}}))$$

而 $\mathrm{MOP}[B_4] = (f_3 \circ f_1)\,(v_{\mathrm{ENTRY}}) \wedge (f_3 \circ f_2)\,(v_{\mathrm{ENTRY}})$

在数据流分析框架具有分配性时，二者的结果是一样的

- $\mathrm{MFP} \preccurlyeq \mathrm{MOP} \preccurlyeq \mathrm{IDEAL}$

例题 1

一个C语言程序如下，右边是优化后的目标代码

```
main()                    pushl %ebp
{                               movl %esp,%ebp
    int i,j,k;                  movl $1,%eax        -- j=1
    i=5;                        movl $6,%edx        -- k=6
    j=1;                      L4:
    while(j<100){               addl %edx,%eax      -- j=j+6
        k=i+1;                  cmpl $99,%eax
        j=j+k;                  jle .L4             -- while(j≤99)
    }
}
```

完成了哪些优化？

例题 1

一个C语言程序如下，右边是优化后的目标代码

```
main()
{
    int i,j,k;
    i=5;
    j=1;
    while(j<100){
        k=i+1;
        j=j+k;
    }
}
```

```
pushl %ebp
        movl %esp,%ebp
        movl $1,%eax          -- j=1
        movl $6,%edx          -- k=6
    L4:
        addl %edx,%eax        -- j=j+6
        cmpl $99,%eax
        jle .L4               -- while(j≤99)
```

复写传播、常量合并、代码外提、删除无用赋值

对i，j和k分配内存单元也成为多余，从而被取消

例题 2

一个C语言程序

```
main()
{
    long i,j;

    while (i) {
        if (j) { i = j; }
    }
}
```

生成的汇编码见右边

为什么会有连续跳转？

```
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
.L2:
        cmpl $0,-4(%ebp)
        jne .L4
        jmp .L3
.L4:
        cmpl $0,-8(%ebp)
        je .L5
        movl -8(%ebp),%eax
        movl %eax,-4(%ebp)
.L5:
        jmp .L2
.L3:
```

```
while E1 do S1
L2:     E1的代码
        真转 L4
        无条件转 L3
L4:     S1的代码
        JMP L2
L3:
if E2 then S2
        E2的代码
        假转 L5
        S2的代码
L5:
```

例题 2

一个C语言程序

```
main()
{
    long i,j;

    while (i) {
        if (j) { i = j; }
    }
}
```

生成的汇编码见右边

为什么会有连续跳转？

```
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
.L2:
        cmpl $0,-4(%ebp)
        jne .L4
        jmp .L3
.L4:
        cmpl $0,-8(%ebp)
        je .L5
        movl -8(%ebp),%eax
        movl %eax,-4(%ebp)
.L5:
        jmp .L2
.L3:
```

嵌套时代码结构变成

L2:　E1的代码
　　真转 L4
　　无条件转 L3
L4:　~~S1的代码~~
　　E2的代码
　　假转 L5
　　S2的代码
L5:　JMP L2
L3:

if E2 then S2
　　E2的代码
　　假转 L5
　　S2的代码
L5:

例题 2

一个C语言程序

```
main()
{
    long i,j;

    while (i) {
        if (j) { i = j; }
    }
}
```

优化编译的汇编码见右边

```
    pushl %ebp
    movl %esp,%ebp
.L7:
    testl %eax,%eax
    je .L3
    testl %edx,%edx
    je .L7
    movl %edx,%eax
    jmp .L7
.L3:
```

例题 3 尾递归

求最大公约数的函数

```
long gcd(p,q)
long p,q;
{
    if (p%q == 0)
        return q;
    else
        return gcd(q, p%q);
}
```

- 其中的递归调用称为尾递归
- 对于尾递归，编译器应怎样产生代码，使得这种递归调用所需的时空开销大大减少？
- 计算实在参数q和p%q，存放在不同的寄存器中
- 将上述寄存器中实在参数的值存入当前活动记录中形式参数p和q的存储单元
- 转到本函数第一条语句的起始地址继续执行

例题 3 尾递归

求最大公约数的函数

```
long gcd(p,q)
long p,q;
{
    if (p%q == 0)
        return q;
    else
        return gcd(q, p%q);
}
```

```
    movl  8(%ebp),%esi     p
    movl  12(%ebp),%ebx  q
.L4:
    movl  %esi,%eax
    cltd                          扩展为64位
    idivl  %ebx
    movl %edx,%ecx        p%q
    testl %ecx,%ecx        p%q
    je .L2
    movl %ebx,%esi        q⇒p
    movl %ecx,%ebx        p%q⇒q
    jmp .L4
.L2:
```

例题 4

Program → Stmt

Stmt → id := Exp | read (id) | write (Exp) |
Stmt ; Stmt |
while (Exp) do begin Stmt end |
if (Exp) then begin Stmt end
else begin Stmt end

Exp → id | lit | Exp OP Exp

定义**Stmt**的两个属性

- ***MayDef***表示它可能定值的变量集合
- ***MayUse***表示它可能引用的变量集合
- 写一个语法制导定义或翻译方案，它计算**Stmt**的上述***MayDef***和***MayUse***属性

例题 4

Stmt → id := Exp

{ Stmt.*MayDef* = {id.*name*} ;

Stmt.*MayUse* = Exp.*MayUse* }

Stmt → read (id)

{ Stmt.*MayUse* = ∅ ; Stmt.*MayDef* = {id.*name*}}

Stmt → write (Exp)

{ Stmt.*MayDef* = ∅ ; Stmt.*MayUse* = Exp.*MayUse* }

Stmt → Stmt_1 ; Stmt_2

{ Stmt.*MayUse* = Stmt_1.*MayUse* ∪ Stmt_2.*MayUse* ;

Stmt.*MayDef* = Stmt_1.*MayDef* ∪ Stmt_2.*MayDef* }

例题 4

Stmt → if (Exp) then begin Stmt$_1$ end

else begin Stmt$_2$ end

{ Stmt.*MayUse* = Stmt$_1$.*MayUse* ∪

Stmt$_2$.*MayUse* ∪ Exp.*MayUse*;

Stmt.*MayDef* = Stmt$_1$.*MayDef* ∪ Stmt$_2$.*MayDef* }

Stmt → while (Exp) do begin Stmt$_1$ end

{ Stmt.*MayUse* = Stmt$_1$.*MayUse* ∪ Exp.*MayUse*;

Stmt.*MayDef* = Stmt$_1$.*MayDef* }

Exp → id { Exp.*MayUse* = {id.*name*} }

Exp → lit { Exp.*MayUse* = ∅ }

Exp → Exp$_1$ OP Exp$_2$

{ Exp.*MayUse* = Exp$_1$.*MayUse* ∪Exp$_2$.*MayUse* }

例题 4

基于*MayDef*和*MayUse*属性, 说明$\text{Stmt}_1;\text{Stmt}_2$和$\text{Stmt}_2;\text{Stmt}_1$在什么情况下有同样的语义

$\text{Stmt}_1.\mathit{MayDef} \cap \text{Stmt}_2.\mathit{MayUse} = \varnothing$ and

$\text{Stmt}_2.\mathit{MayDef} \cap \text{Stmt}_1.\mathit{MayUse} = \varnothing$ and

$\text{Stmt}_1.\mathit{MayDef} \cap \text{Stmt}_2. \mathit{MayDef} = \varnothing$