北京大学信息科学技术学院

2022年春季学期《编译技术》

第九章 代码优化

Code Optimization

内容提要

- 代码优化概述
- 代码优化的主要方法
- 流图中的循环及其查找
- 数据流分析简介
  - 可达定义
  - 可用表达式
  - 活跃变量

代码优化概述

- 为了设计一个好的代码优化程序，首先考虑如下几个方面：
  - 代码优化应遵循的准则
  - 进行代码优化的阶段
  - 进行代码优化的范围
  - 代码优化程序的结构

The 80/20 Rule

- 通常来说，**一个程序80% 的执行时间花费在20%的代码之上；**对于某些程序可能会达到90%/10%
- 程序优化的主要精力应当关注的是程序中最重要的10/20%
- 在优化常用代码的时候，甚至可以选择使不常用代码部分速度变慢

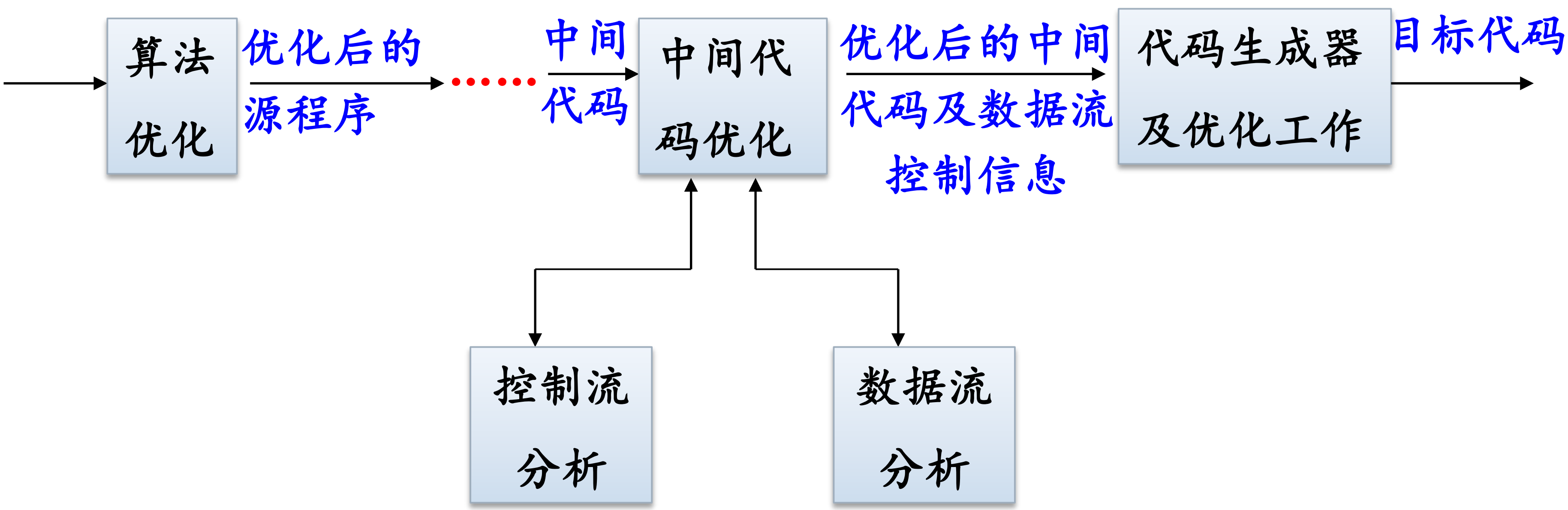
代码优化的阶段

- 算法设计阶段的优化
- 编译程序产生中间代码后的优化
- 目标代码生成阶段的优化

代码优化的范围

- 局部优化（local optimization）
  - 只对基本块内的语句进行分析，在基本块内实施的优化。
- 全局优化（global optimization）
  - 对整个过程所有基本块的信息及它们之间的关系进行分析，在此基础上在整个过程范围内实施优化。
- 过程间优化（inter-procedural optimization）
  - 对整个程序所有过程及其调用信息进行分析，对整个程序进行整体优化。

优化程序的一般结构



算法优化
优化后的源程序
中间代码
中间代码优化
优化后的中间代码及数据流控制信息
代码生成器及优化工作
目标代码
控制流分析
数据流分析

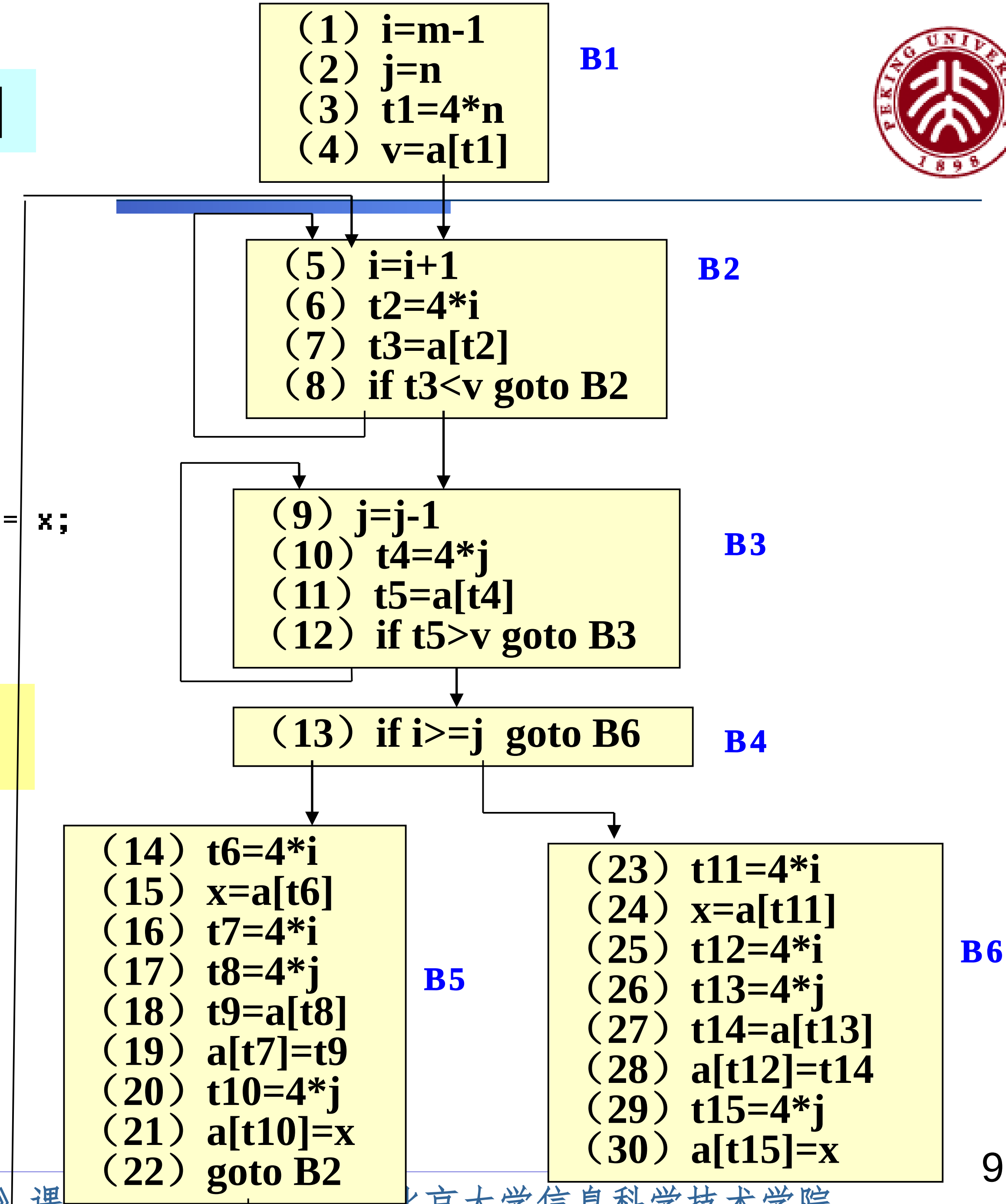
代码优化的主要方法

- 删除公共子表达式
- 复写传播
- 删除无用赋值
- 合并已知量
- 循环优化
  - 代码外提
  - 消减运算强度
  - 删除归纳变量

快速排序的程序流图

```c
void quicksort (int m, int n)
{
    int i,j,v,x;
    if(n<=m) return;
    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[j]; a[j]= x;
    quicksort(m,j);quicksort(i+1,n);
}
```

图：用c语言编写的快速排序函数



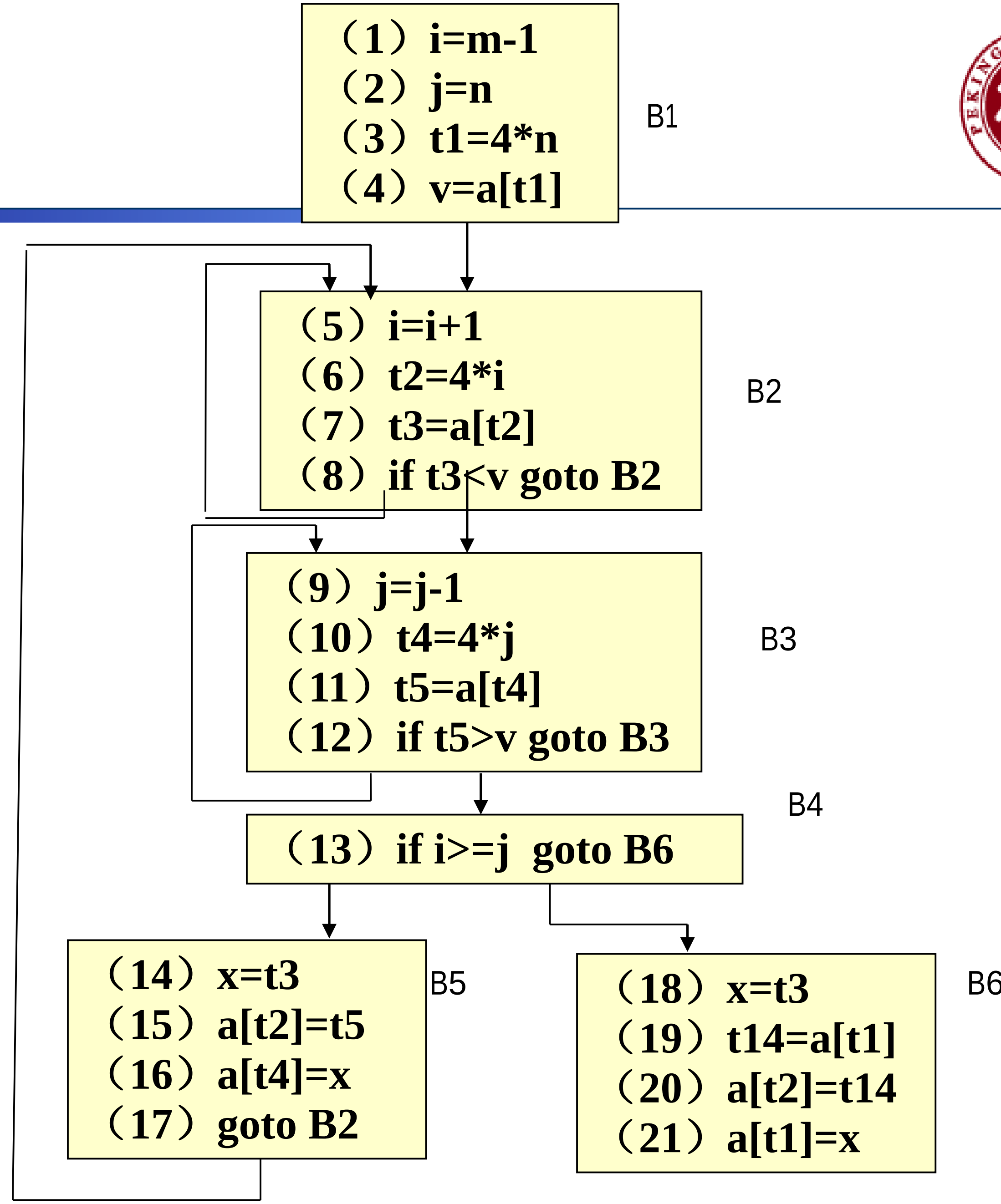
删除公共子表达式

（14） t6=4*i
（15） x=a[t6]
（16） t7=4*i
（17） t8=4*j
（18） t9=a[t8]
（19） a[t7]=t9
（20） t10=4*j
（21） a[t10]=x
（22） goto B2

（14） t6=4*i
（15） x=a[t6]
（17） t8=4*j
（18） t9=a[t8]
（19） a[t6]=t9
（21） a[t8]=x
（22） goto B2

删除局部公共子表达式后的B5

删除B5和B6中公共子表达式后的快速排序的程序流程图



复写传播(Copy Propagation)

- 形如x=y的赋值语句称为**复写语句**。
- 复写传播就是把用到x的地方换成y，从而最终达到可以删除x=y的目的。

（14）x=t3
（15）a[t2]=t5
（16） a[t4]=x
（17）goto B2

（14）x=t3
（15）a[t2]=t5
（16）a[t4]=t3
（17）goto B2

B5

删除无用赋值

- 无用赋值是指计算的结果决不被引用的语句
- 一些优化变换可能会引起无用赋值
  - 例：复写传播可能会引起死代码删除

（14） x=t3
（15） a[t2]=t5
（16） a[t4]=t3
（17） goto B2

~~（14） x=t3~~
（15） a[t2]=t5
（16） a[t4]=t3
（17） goto B2

B5

合并已知量

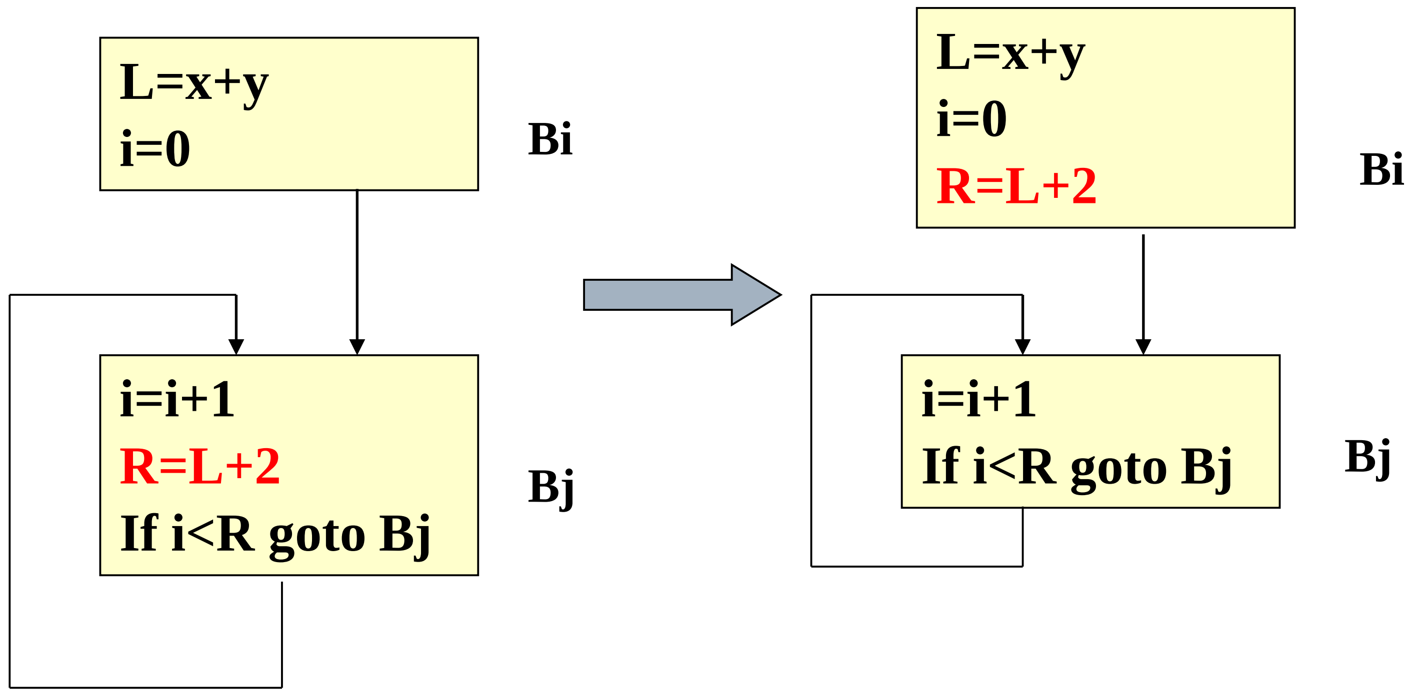
- 编译时可以进行的求值运算。
- 例如：

```
t = 2*3.14
c = t*R
```

→

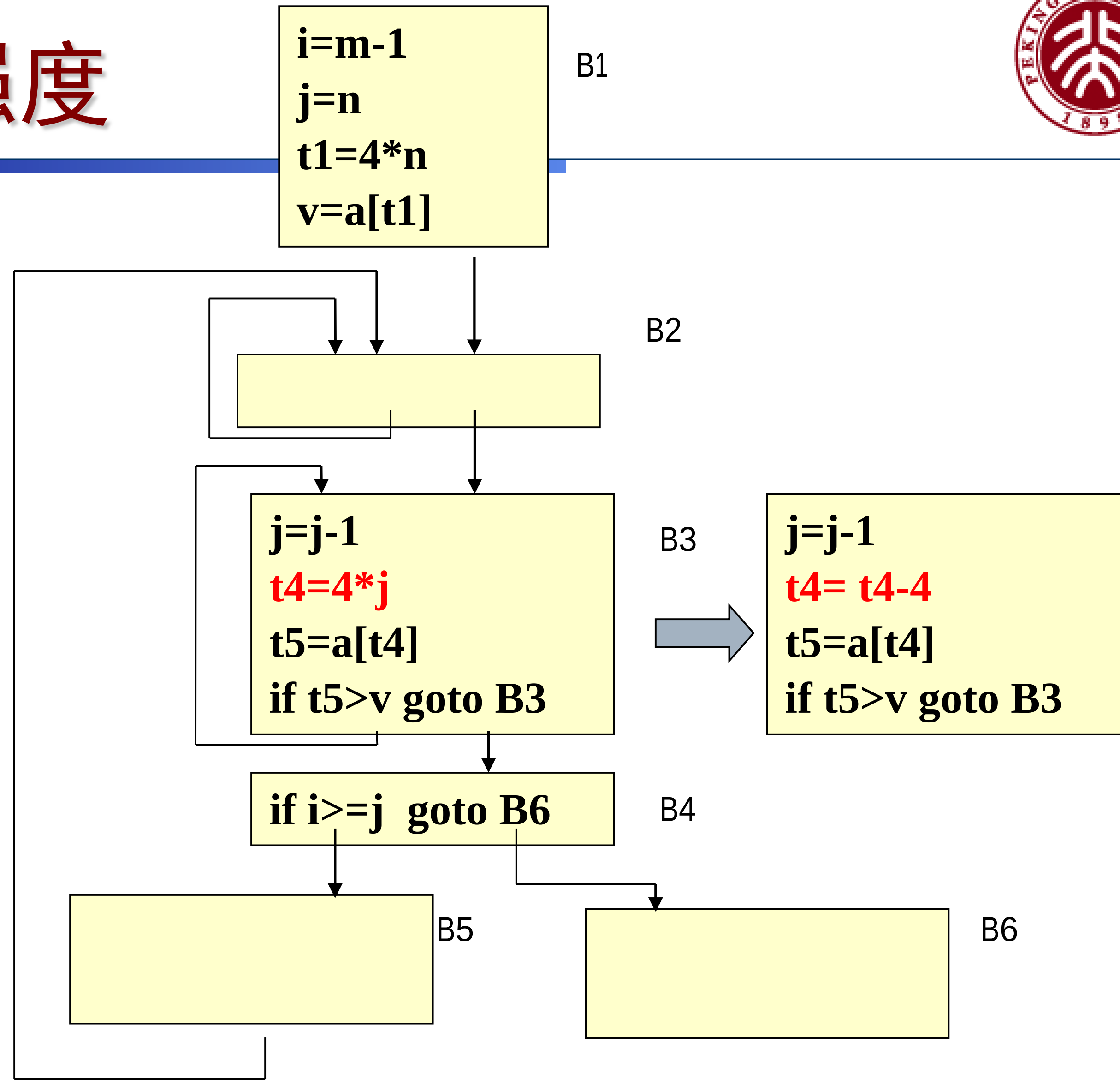
```
t = 6.28
c = t*R
```

代码外提的例子


L=x+y
i=0
Bi
i=i+1
R=L+2
If i<R goto Bj
Bj
L=x+y
i=0
R=L+2
Bi
i=i+1
If i<R goto Bj
Bj

（a）　　　　　　　　（b）

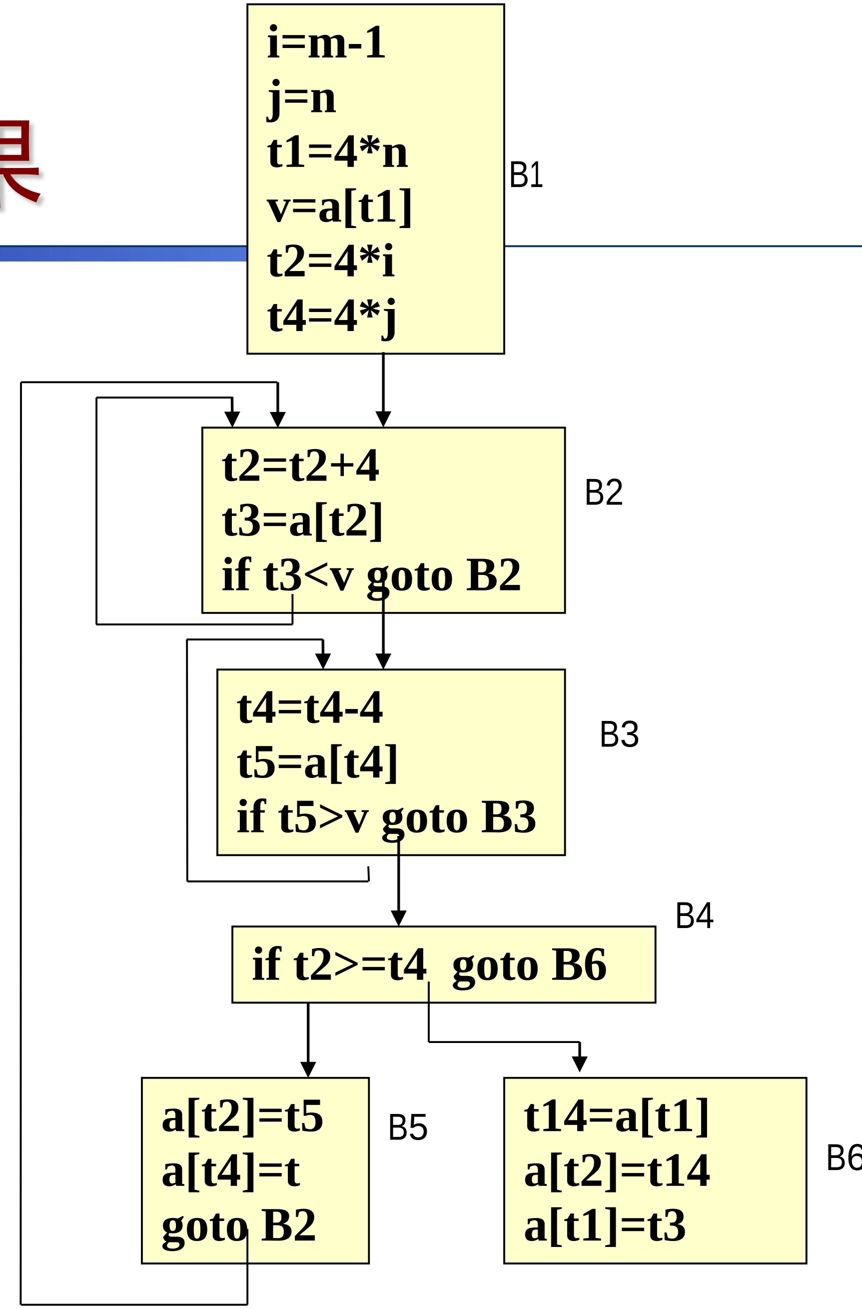
消减运算强度



消减运算强度示例

多项优化的结果

进行多项优化后的快速排序的程序流程图



流图中的循环及其查找

- **控制结点**
  - 在一个流图中，如果从流图首结点到结点n的每一条路径都包含结点d，则称d是n的**控制结点**，或称**结点d控制结点n**。
    - 记作**d DOM n**（DOM是Dominate的缩写）。
  - n的所有控制结点组成的集合称为n的**控制结点集**，记作 D(n)。
  - 如果把DOM看成是定义在流图结点集合N上的关系，那么显然有：

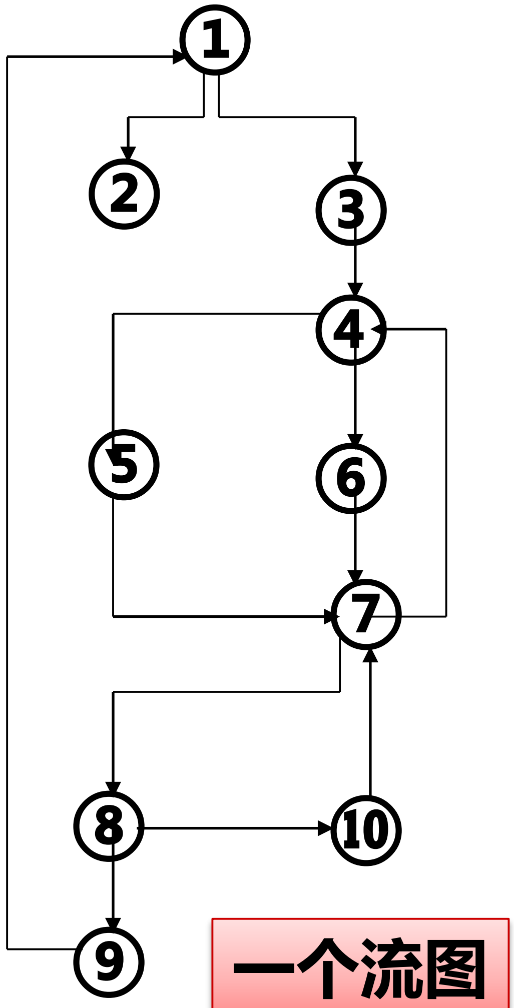
    （1）对任意的n∈N都有n DOM n

    （2）对m, n∈N，若有 m DOM n 且 n DOM m，则必有 m=n；

    （3）对m, n, q∈N，若 m DOM n 且 n DOM q，则必有m DOM q。

    所以DOM是N上的一个偏序关系。

控制结点示例



D(1) = {1}
D(2) = {1,2}
D(3) = {1,3}
D(4) = {1,3,4}
D(5) = {1,3,4,5}
D(6) = {1,3,4,6}
D(7) = {1,3,4,7}
D(8) = {1,3,4,7,8}
D(9) = {1,3,4,7,8,9}
D(10) = {1,3,4,7,8,10}

控制结点的计算

- 对于非首结点n，

$$D(n)=\{n\}\cup(\cap D(p),\ p\in P(n))$$

- 对于首结点n_0，

$$D(n_0)=\{n_0\}$$

- 其中P(n)代表结点n的所有前驱结点组成的集合。

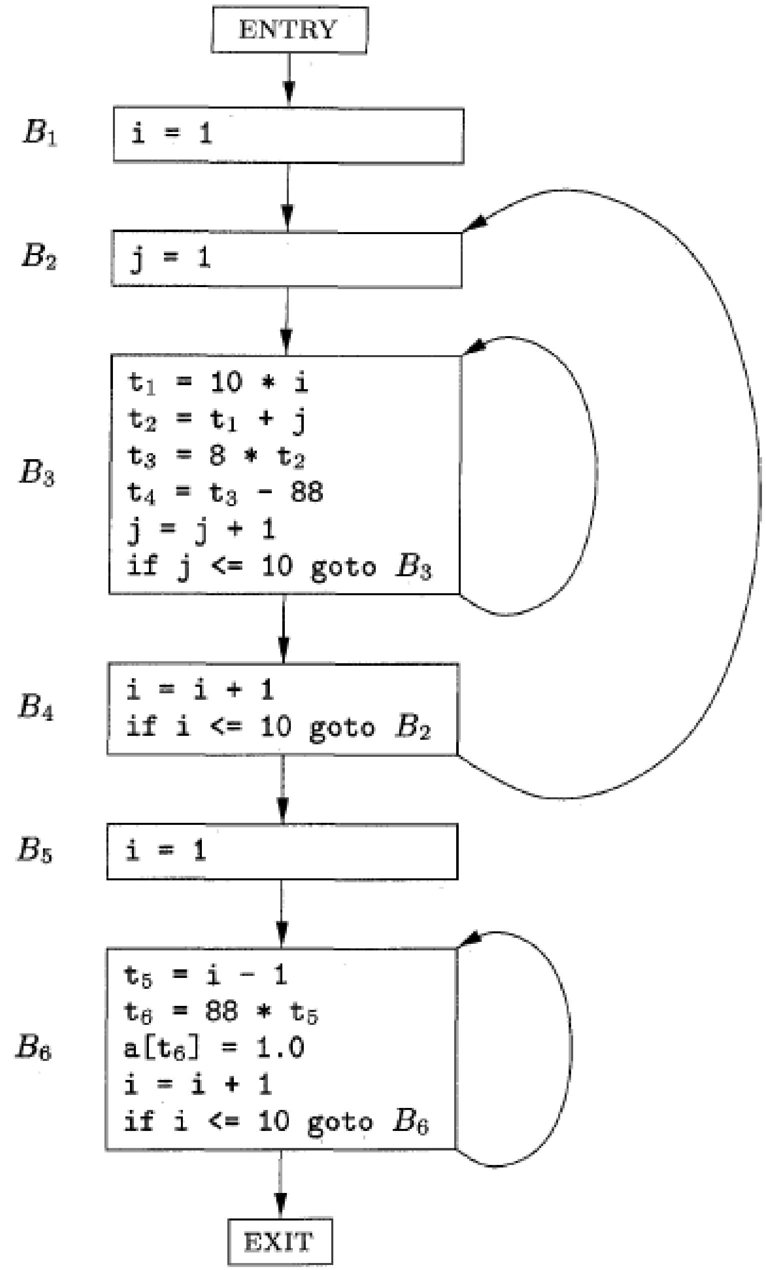
回边

- 回边 (Back Edge)
  - 定义：如果n→d是流图中一条边，并且d是n的控制结点，即d DOM n，则称n → d为回边。

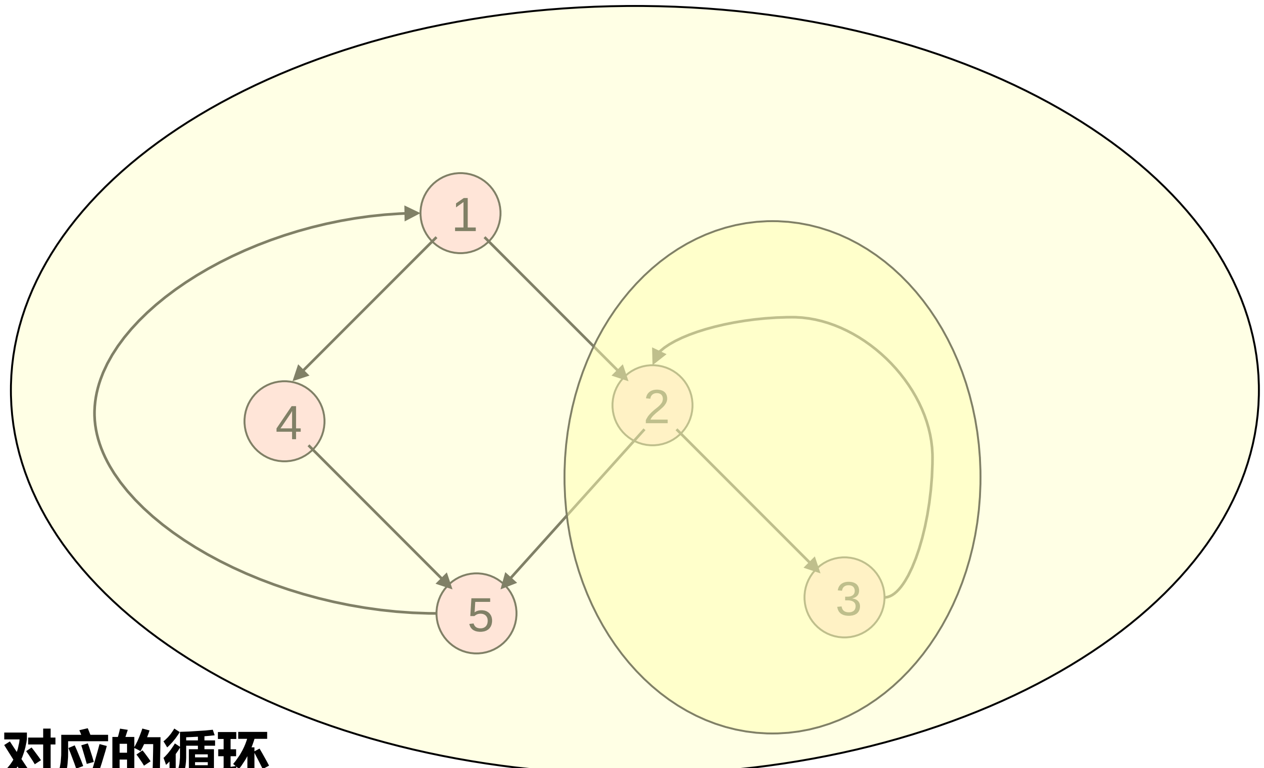
- 循环 (Loop) 的查找
  - 设t →h是一条回边，那么由t →h出发构造的循环是由结点t，h以及有路径到达t又不经过点h的所有结点及它们之间的边组成的，结点h是该循环的唯一入口结点。

回边与循环的例子

- 回边：
  - B3->B3
  - B6->B6
  - B4->B2
- 循环：
  - {B3}
  - {B6}
  - {B2, B3, B4}



回边和循环的例子

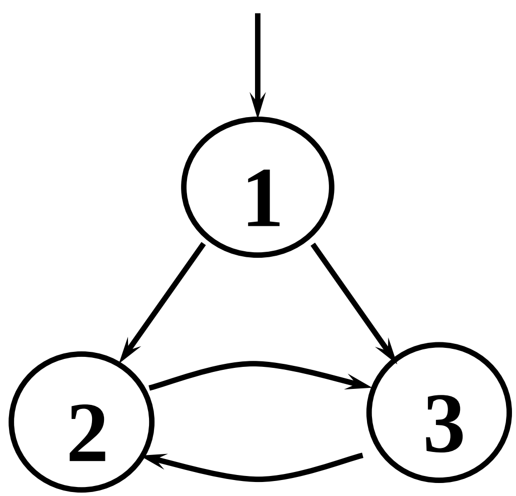


5 -> 1 对应的循环
{1，2，3，4，5}

3 -> 2 对应的循环
{2，3}

可归约流图

- 定义：一个流图是可归约的(reducible)，是指去掉其所有回边后，它不再有回路。



2->3和3->2都不是回边！

一个不可归约的流图

数据流分析简介
Dataflow Analysis

Reaching Definitions（可达定义）

Available Expressions（可用表达式）

Live Variables（活跃变量）

Part of the slides are adapted from MIT 6.035 “Computer Language Engineering”

Basic Concepts

- 点和路径的概念
  - 点(progam point) 是指四元式（或称语句）的位置（地址或编号）。
  - 路径(path) 是点p_1, p_2, ……, p_n构成的一个序列，对于任意的i ($1 \le i \le n$):

    （1）点p_{i+1}与点p_i所指的四元式在同一个基本块内并且p_{i+1}指的四元式紧跟在p_i指的四元式之后。或者，

    （2）p_i指的是基本块B的最后一个四元式，而p_{i+1}指的是B的一个后续基本块的第一个四元式。

Reaching Definitions (可达定义)

- Concept of definition and use
  - a = x+y

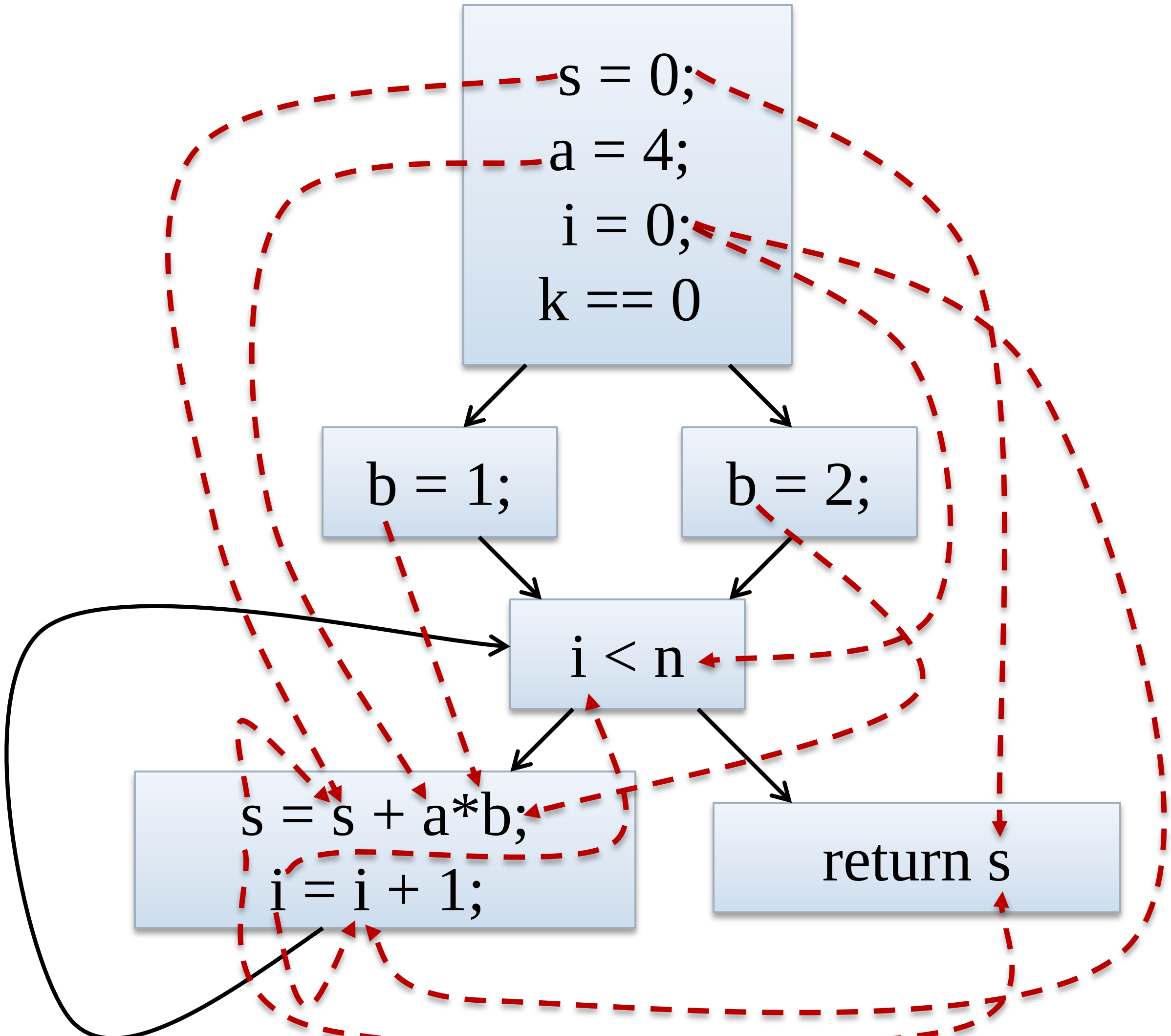
    is a **definition (定义)** of a

    is a **use (使用)** of x and y
- A definition reaches a use if

  value written by definition

  may be read by use

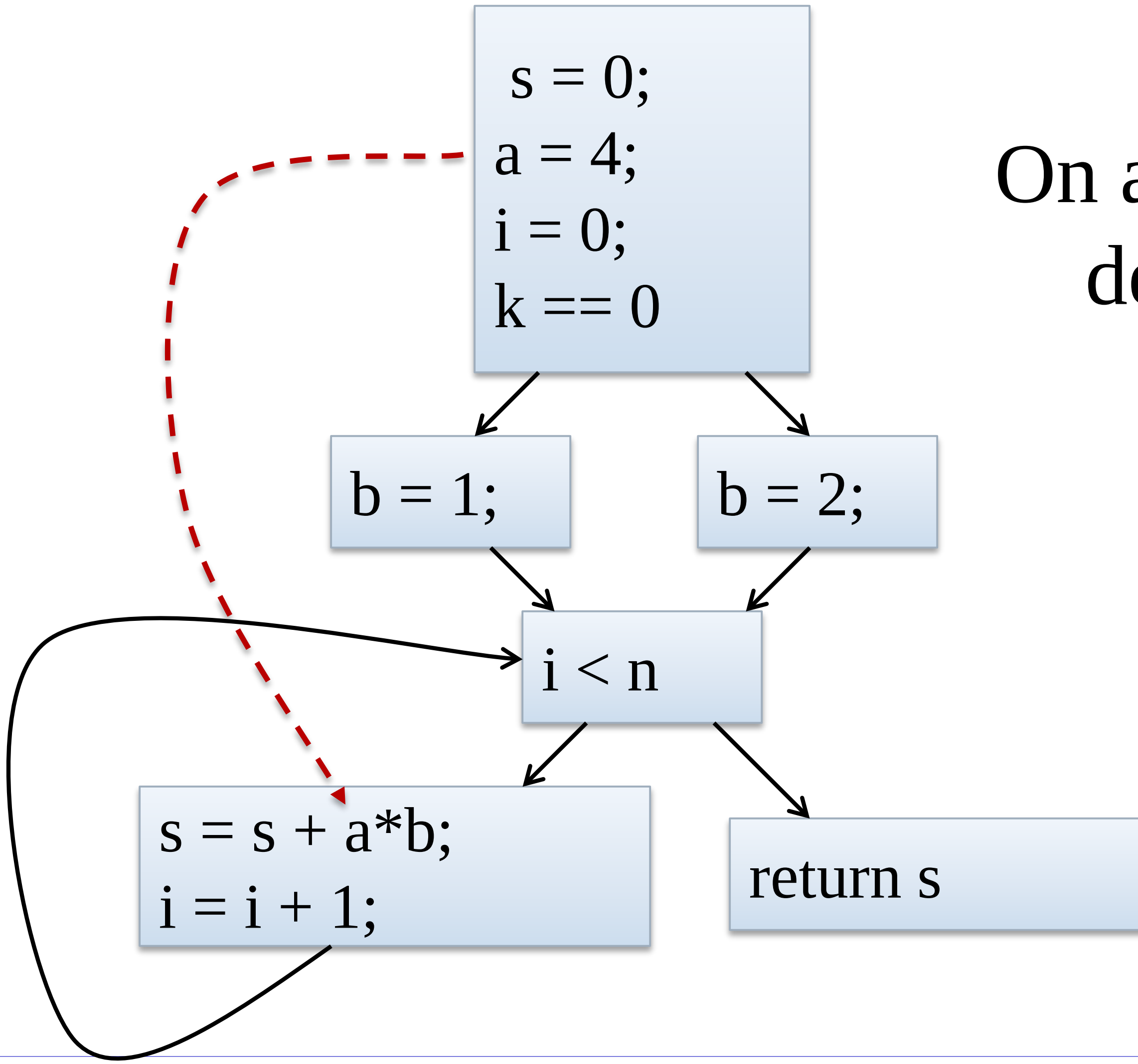
Reaching Definitions



Reaching Definitions and Constant Propagation（常量传播）

- Is a use of a variable a constant?
  - Check all the reaching definitions
  - If all assign variable to the same constant
  - Then use is in fact a constant
- Can replace the variable with the constant

Is a Constant in s = s+a*b?

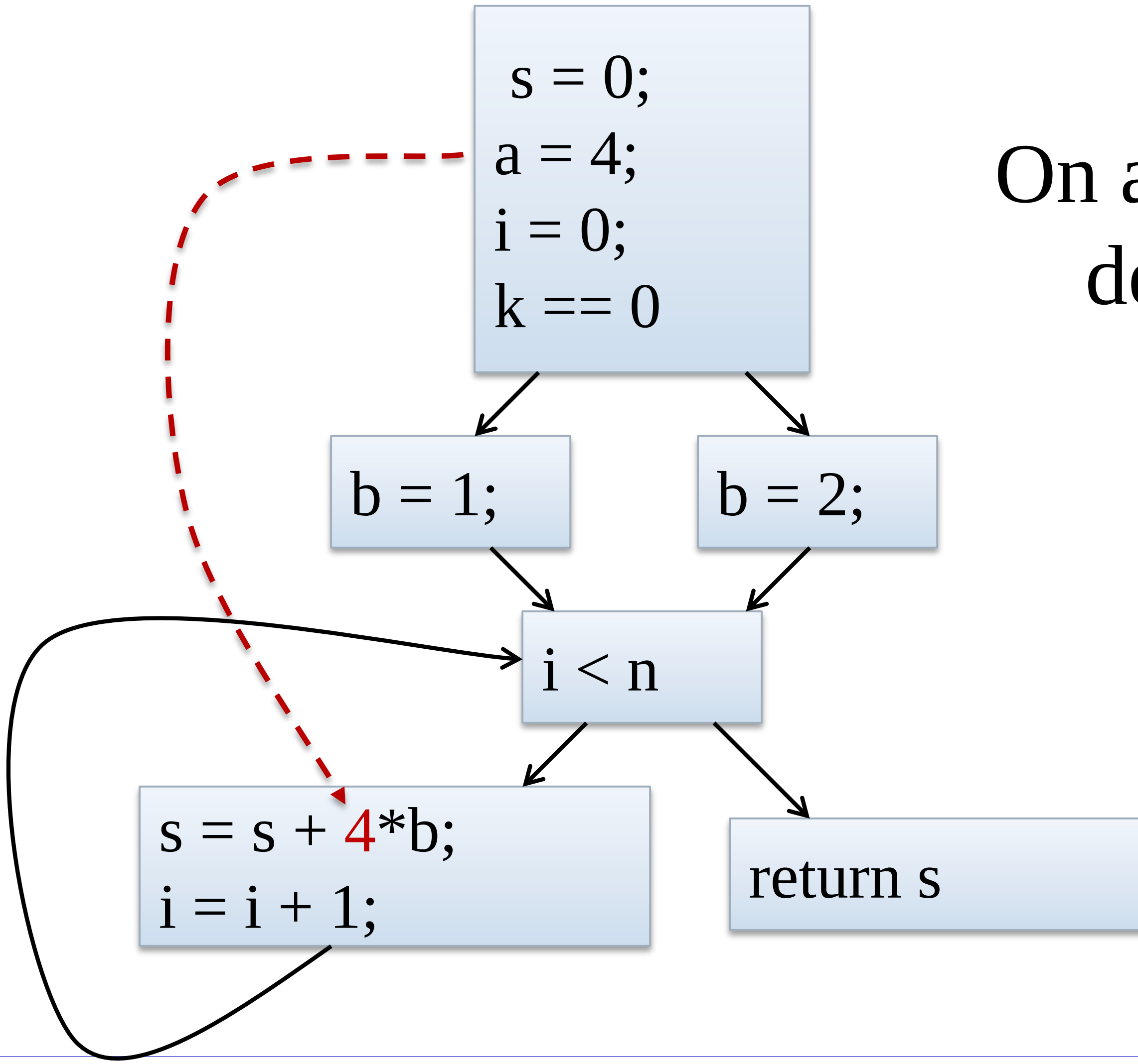


Yes!

On all reaching definitions

a = 4

Constant Propagation Transform

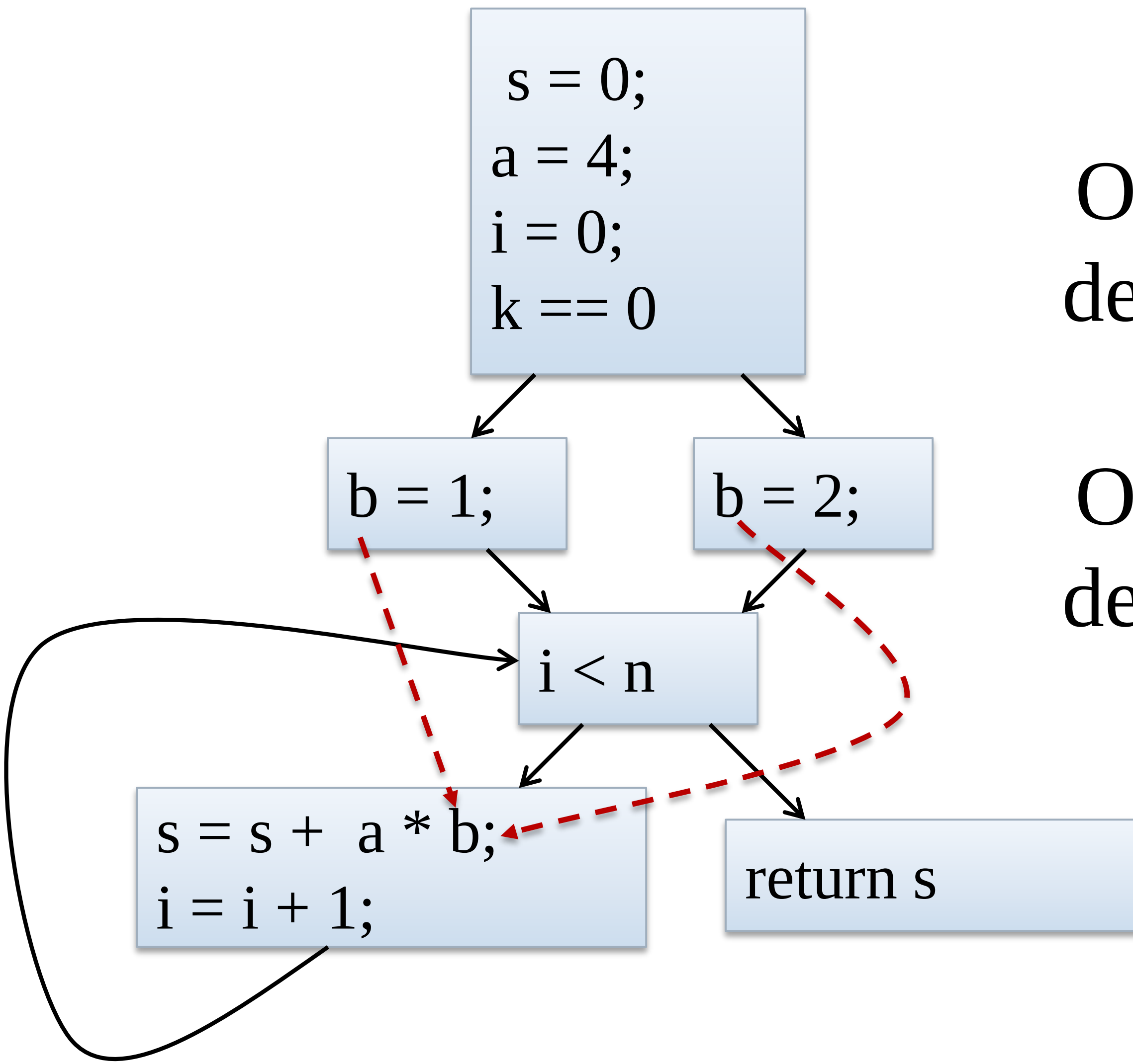


Yes!

On all reaching definitions

a = 4

Is b Constant in s = s+a*b?



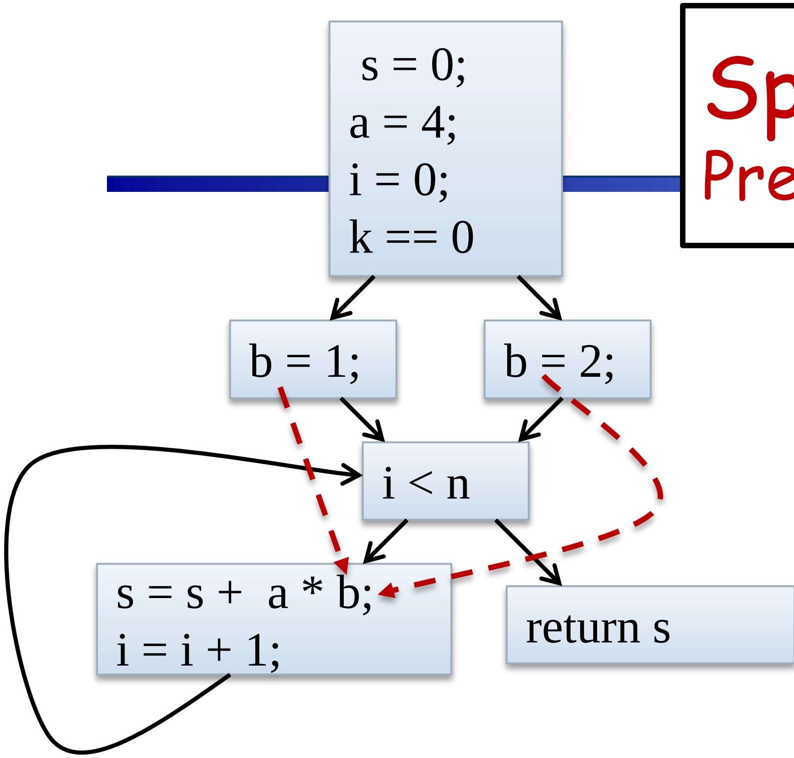
No!

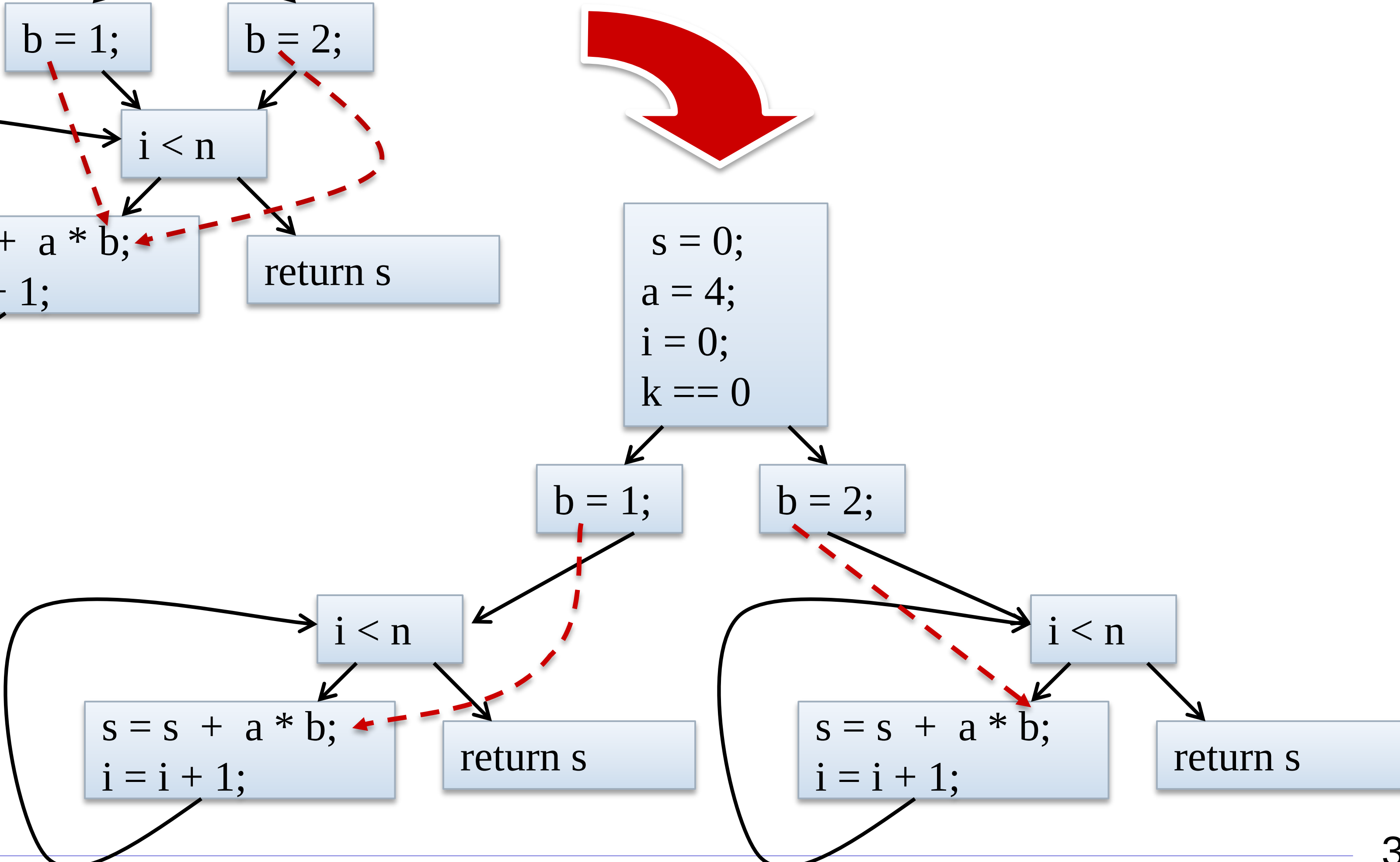
One reaching definition with b = 1

One reaching definition with b = 2

Splitting (拆分)

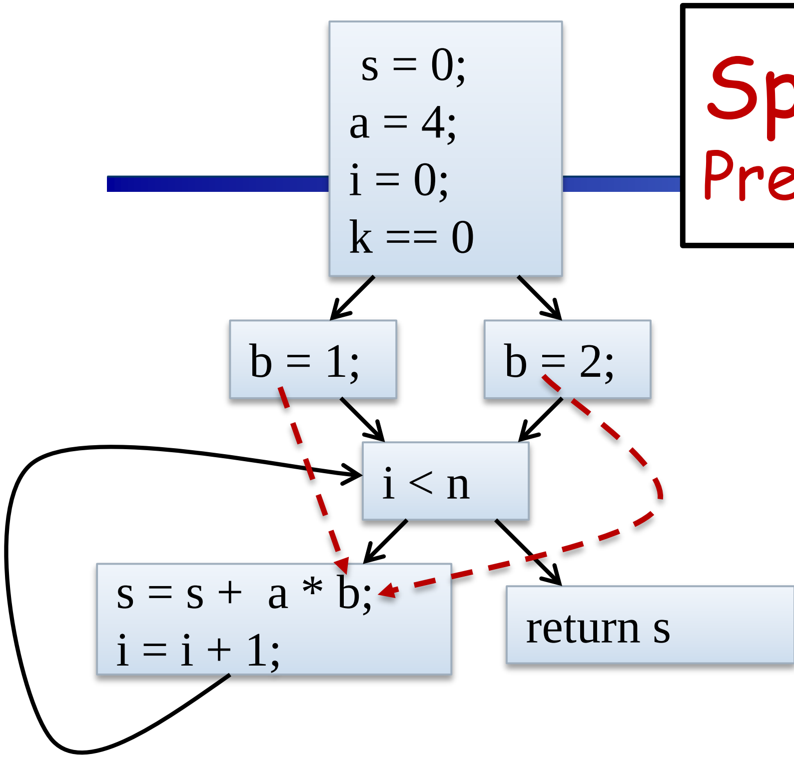
Preserves Information Lost At Merges

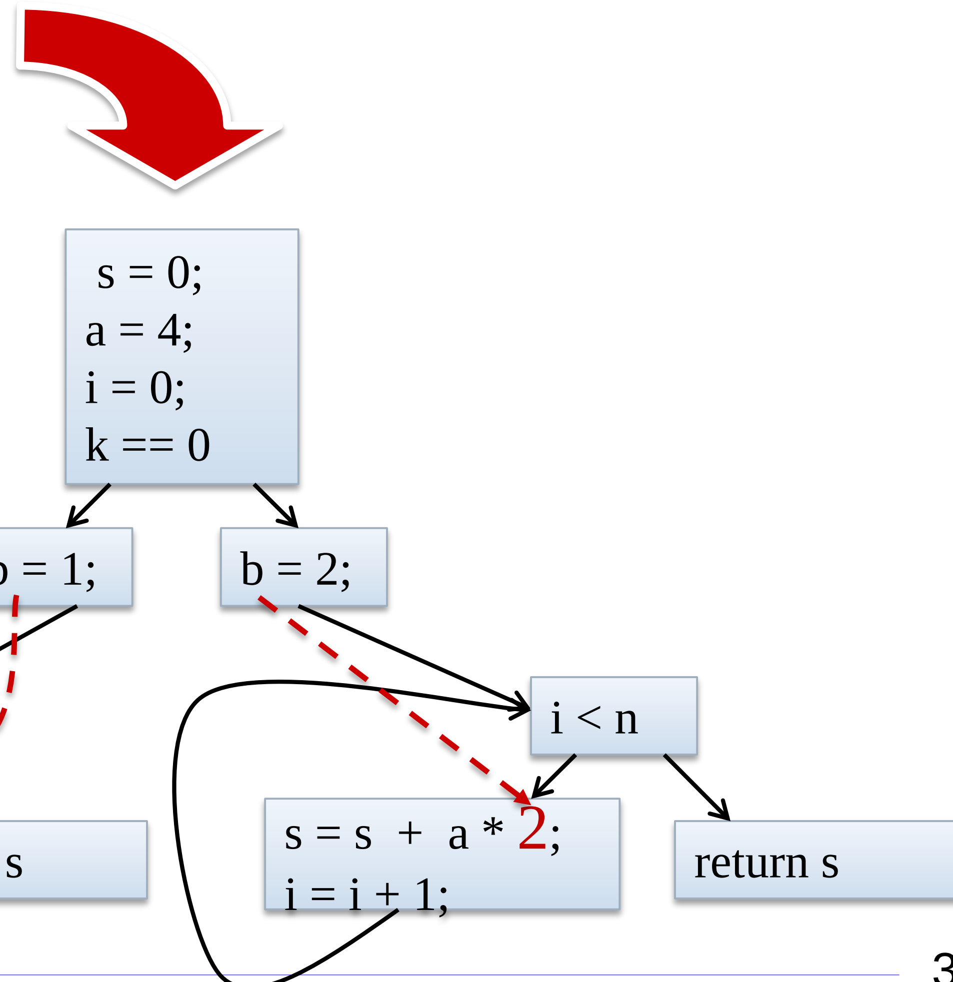




Splitting (拆分)

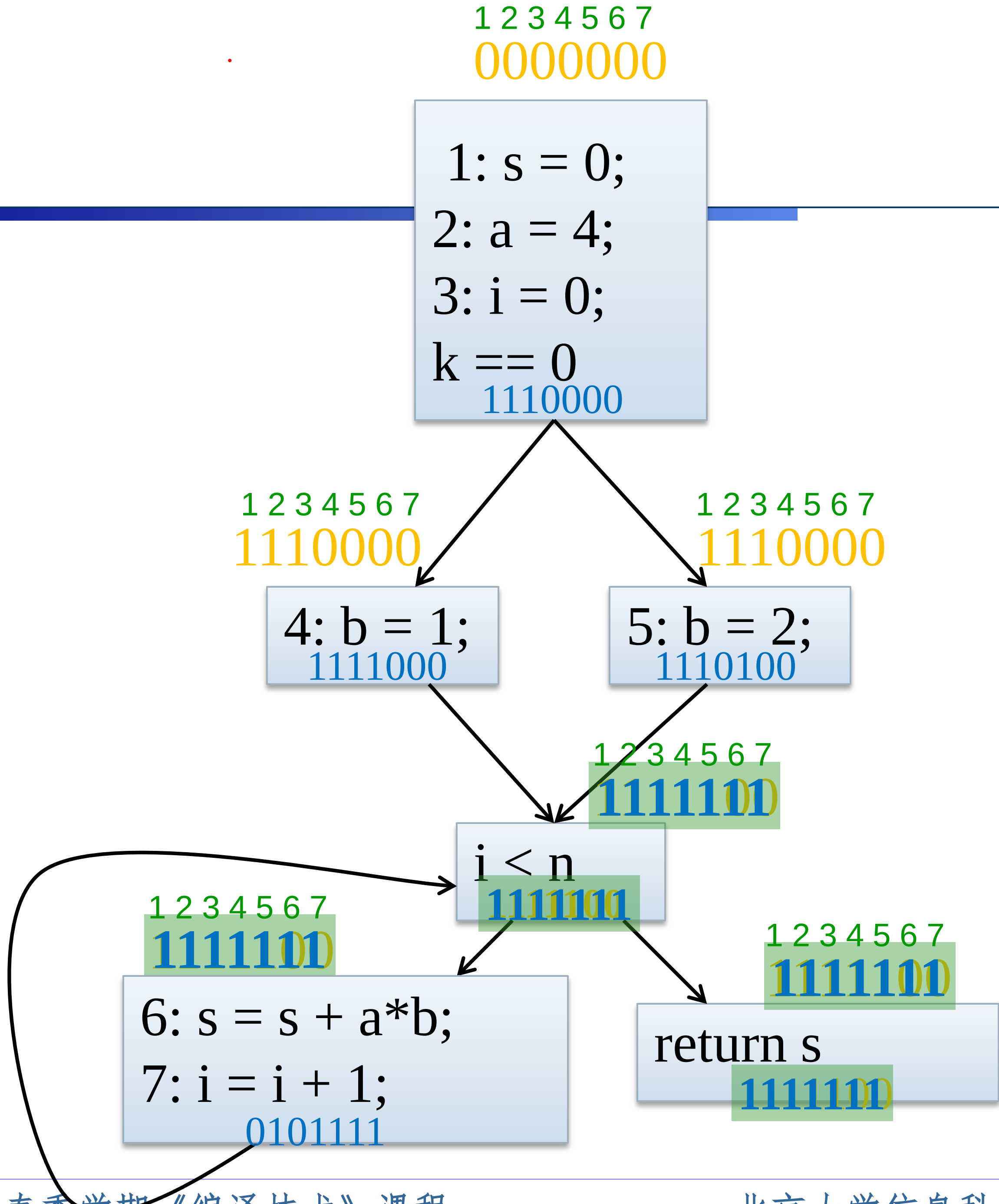
Preserves Information Lost At Merges





Computing Reaching Definitions

- Compute with sets of definitions
  - represent sets using bit vectors
  - each definition has a position in bit vector
- At each basic block, compute
  - definitions that reach start of block
  - definitions that reach end of block
- Do computation by simulating execution of program until reach fixed point (不动点)


1 2 3 4 5 6 7
0000000
1: s = 0;
2: a = 4;
3: i = 0;
k == 0
1110000
1 2 3 4 5 6 7
1110000
1 2 3 4 5 6 7
1110000
4: b = 1;
1111000
5: b = 2;
1110100
1 2 3 4 5 6 7
i < n
1 2 3 4 5 6 7
6: s = s + a*b;
7: i = i + 1;
0101111
1 2 3 4 5 6 7
return s

Data-Flow Analysis Schema

- **Data-flow value: at every program point**
- **Domain: the set of possible data-flow values for this application**
- **IN[s] and OUT[s]: the data-flow values before and after each statement s**
- **Data-flow problem: find a solution to a set of constraints on the IN [s] ‘s and OUT[s] ‘s, for all statements s**
  - **based on the semantics of the statements (“transfer functions”, 转换函数)**
  - **based on the flow of control**

Constraints （约束条件）

- **Transfer function （转换函数）: relationship between the data-flow values before and after a statement**
  - **Forward: $OUT[s] = f_s(IN[s])$**
  - **Backward: $IN[s] = f_s(OUT[s])$**

- **Within a basic block $(s_1, s_2, \ldots, s_n)$**
  - **$IN[s_i+1] = OUT[s_i]$, for all $i = 1, 2, \ldots, n-1$**

Data-Flow Schemas on Basic Blocks

- Each basic block B (s_1, s_2, …, s_n) has
  - **IN – data-flow values immediately before a block**
  - **OUT – data-flow values immediately after a block**

  - $IN[B] = IN[S_1]$
  - $OUT[B] = OUT[S_n]$
  - $OUT[B] = f_B\,(IN[B]\,)$
    - Where $f_B = fs_n \circ \bullet\bullet\bullet \circ fs_2 \circ fs_1$

Between Blocks

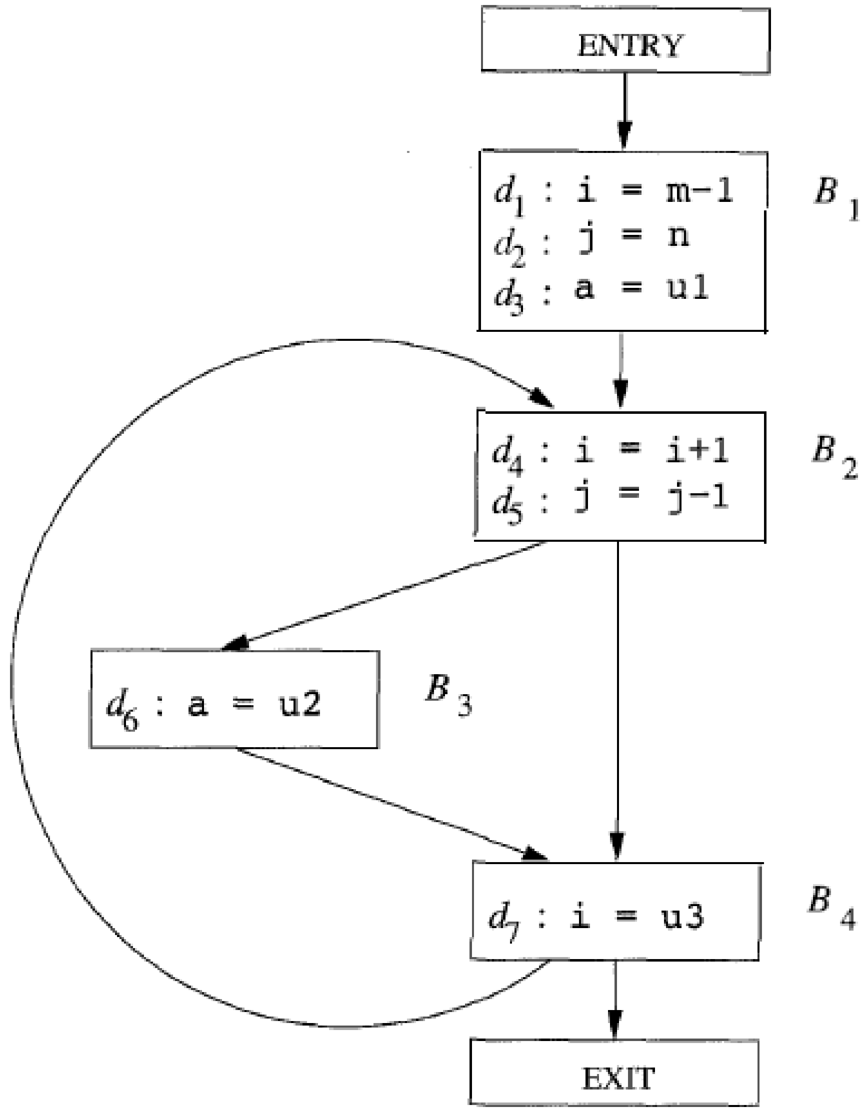
- Forward analysis（正向分析）
  - (eg: Reaching definitions)
  - $\mathrm{IN}[B] = \cup_{P \in \text{predecessors of } B} \mathrm{OUT}[P]$

- Backward analysis（反向分析）
  - (eg: live variables)
  - $\mathrm{IN}[B] = f_B(\mathrm{OUT}[B])$
  - $\mathrm{OUT}[B] = \cup_{S \in \text{successors of } B} \mathrm{IN}[S]$

Formalizing Reaching Definitions

- Each basic block has
  - IN - set of definitions that reach beginning of block
  - OUT - set of definitions that reach end of block
  - GEN - set of definitions generated in block
  - KILL - set of definitions killed in block
- GEN[s = s + a*b; i = i + 1;] = 0000011
- KILL[s = s + a*b; i = i + 1;] = 1010000
- Compiler scans each basic block to derive GEN and KILL sets

Example


ENTRY
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
EXIT

Dataflow Equations

- $IN[b] = OUT[b_1] \cup \ldots \cup OUT[b_n]$
  - where $b_1, \ldots, b_n$ are predecessors of b in CFG
- $OUT[b] = (IN[b] - KILL[b]) \cup GEN[b]$
- $IN[entry] = 0000000$
- Result: system of equations

Solving Equations

- Use **fixed point** algorithm （不动点算法）
- Initialize with solution of **OUT[b] = 0000000**
- Repeatedly apply equations
  - $IN[b] = OUT[b_1] \cup \ldots \cup OUT[b_n]$
  - $OUT[b] = (IN[b] - KILL[b]) \cup GEN[b]$
- Until reach fixed point
- Until equation application has no further effect
- Use a **worklist** to track which equation applications may have a further effect

Reaching Definitions Algorithm

```
for all nodes n in N
    OUT[n] = emptyset; // OUT[n] = GEN[n];
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry }; // N = all nodes in graph

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = emptyset;
    for all nodes p in predecessors(n)
        IN[n] = IN[n] U OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
        for all nodes s in successors(n)
            Changed = Changed U { s };
```

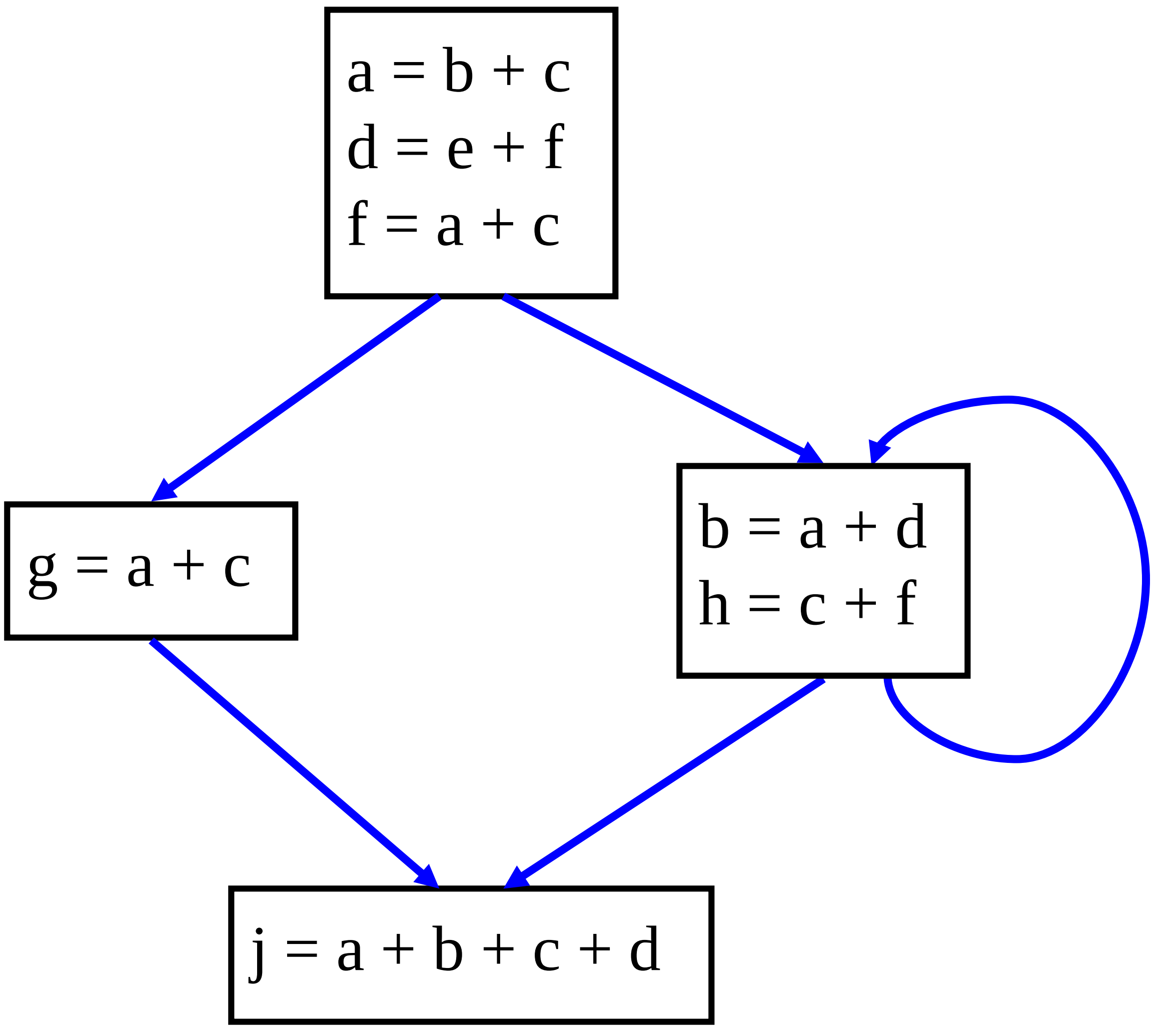
Questions

- Does the algorithm halt?
  - yes, because transfer function is monotonic
  - if increase IN, increase OUT
  - limited number of states, all bits are 1

可用表达式 (Available Expressions)

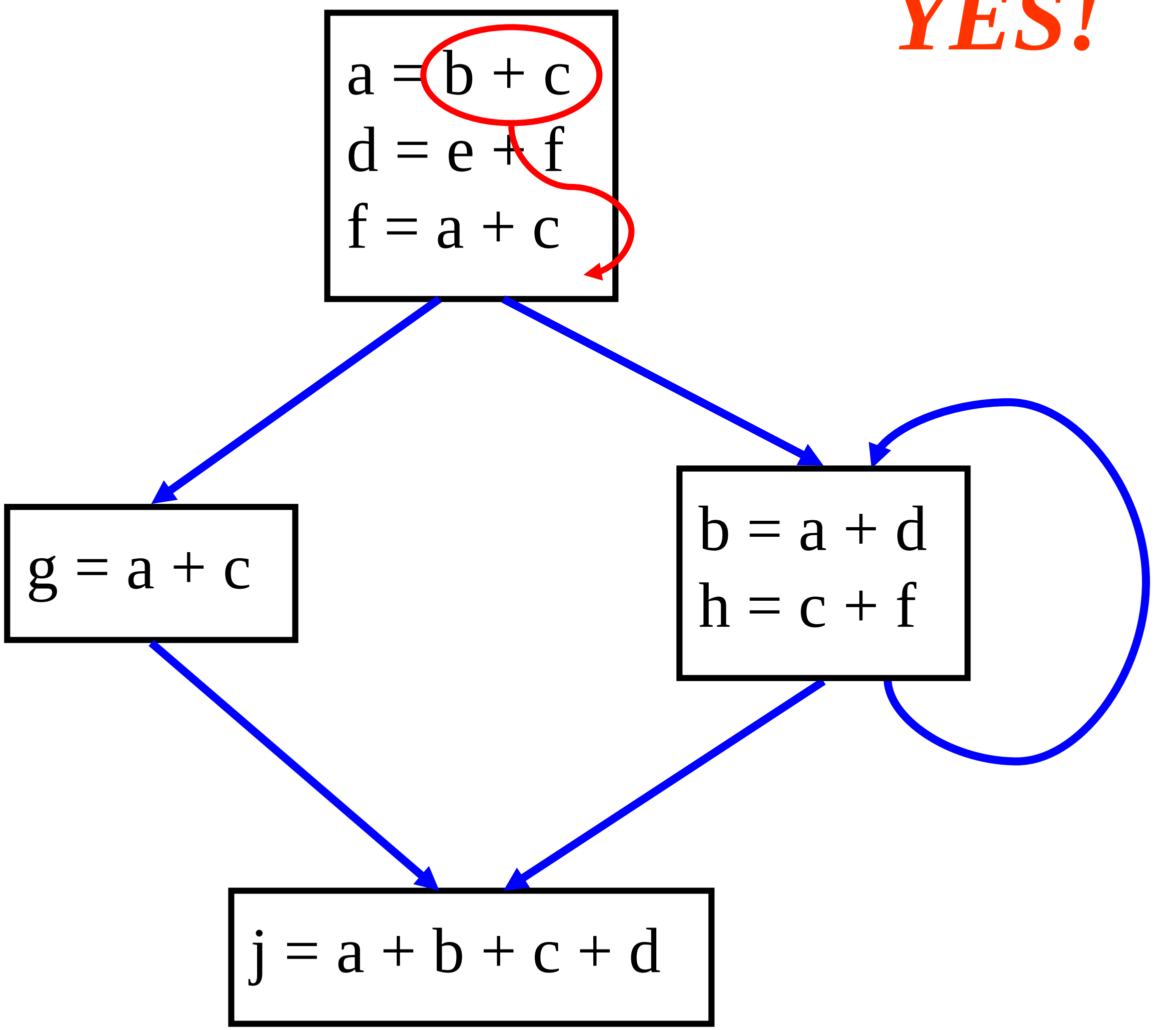
- 表达式 x+y 在程序点 p 可用指的是
  - 从流图入口结点到达 p 点的每条路径都对表达式 x+y 求值，且
  - 从最后一个这样的求值之后到 p 点的路径上没有再次对 x 或 y 赋值
- 可用表达式信息可用来寻找全局公共子表达式
  - 公共子表达式消除：Common Subexpression Elimination (CSE)
  - 如果一个表达式是可用的，我们不需要对它进行重新计算

可用表达式举例



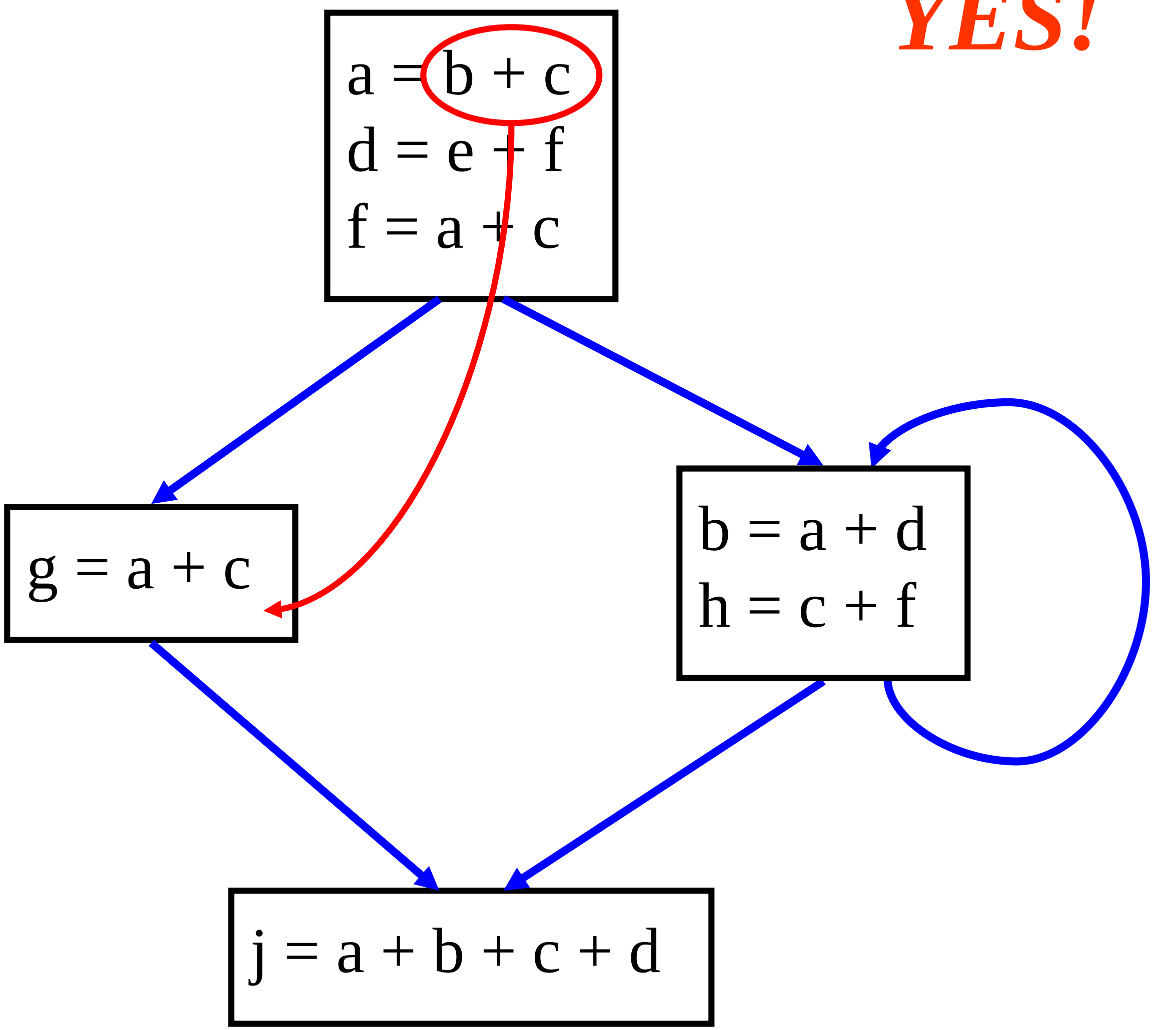
判断表达式 b+c 是否可用

YES!

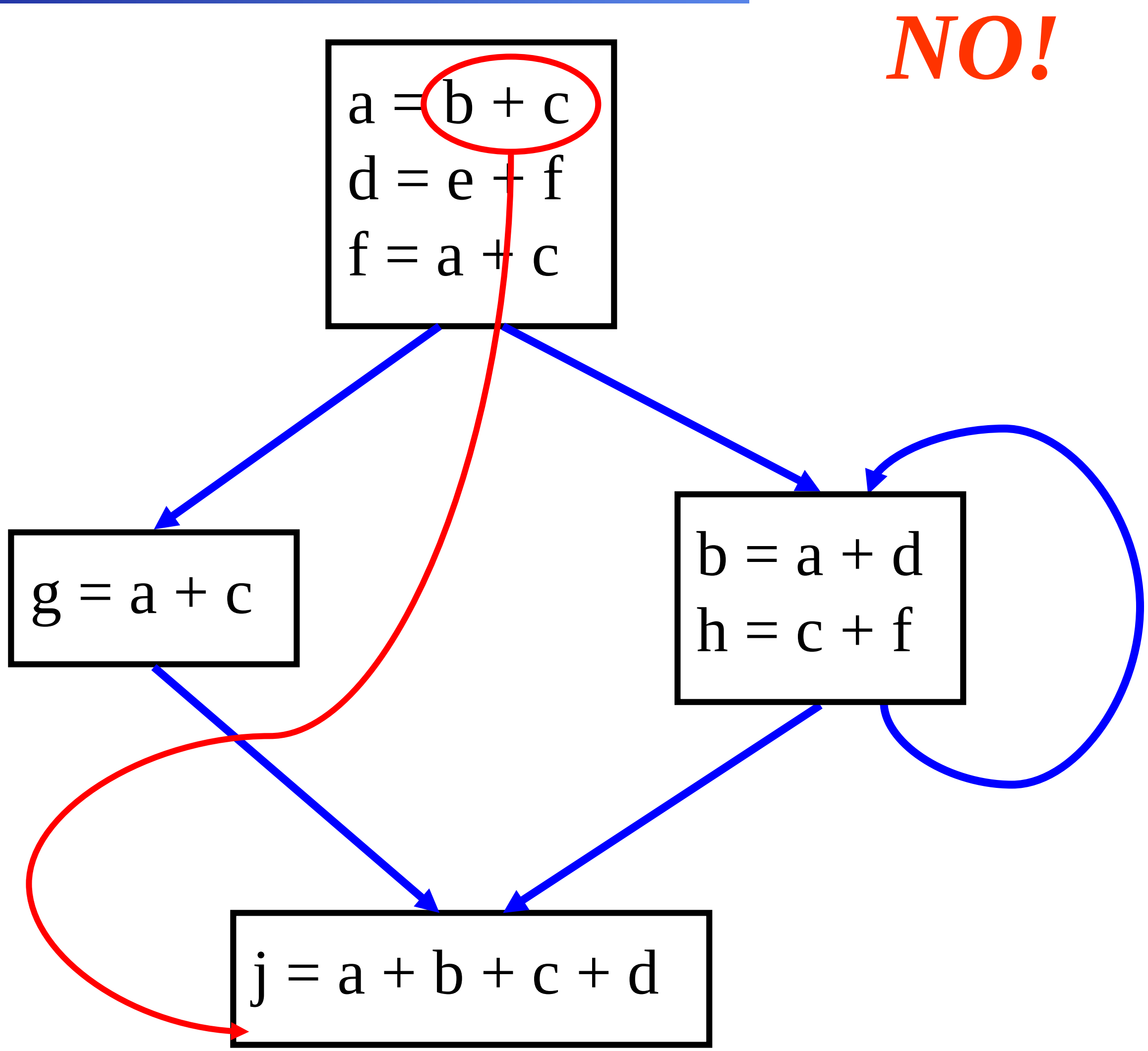


判断表达式 b+c 是否可用

YES!

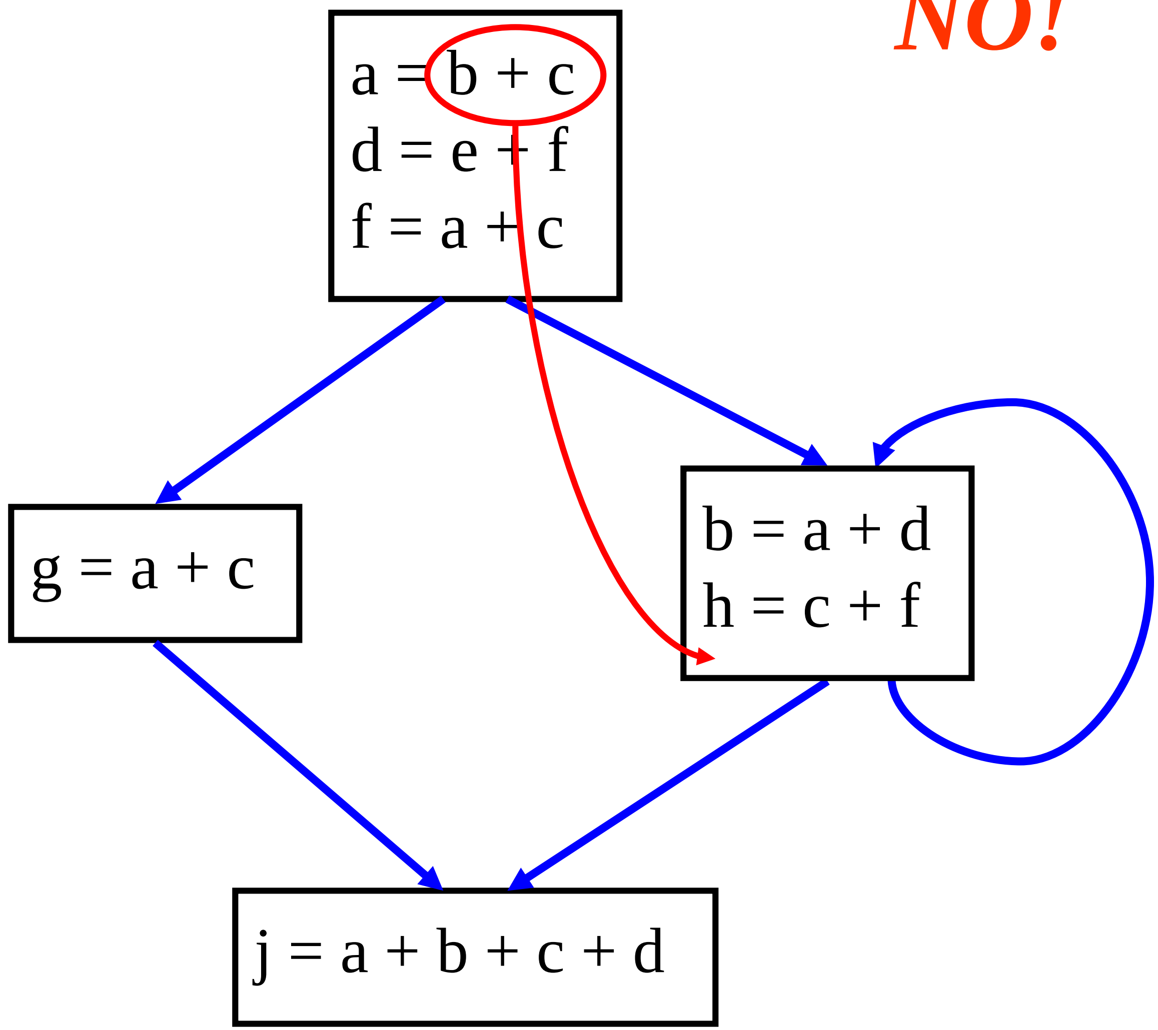


Is the Expression Available?

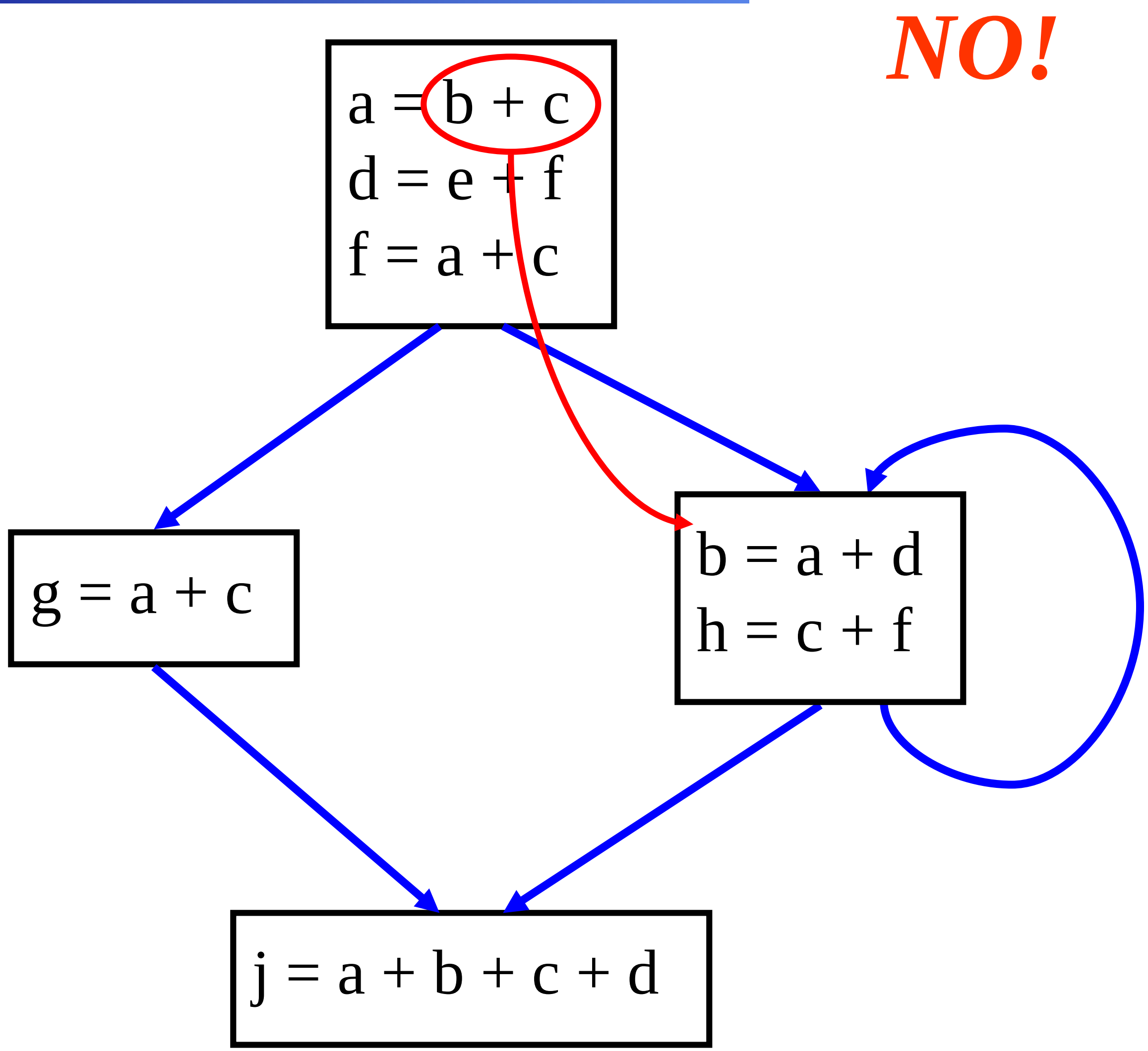


判断表达式 b+c 是否可用

NO!

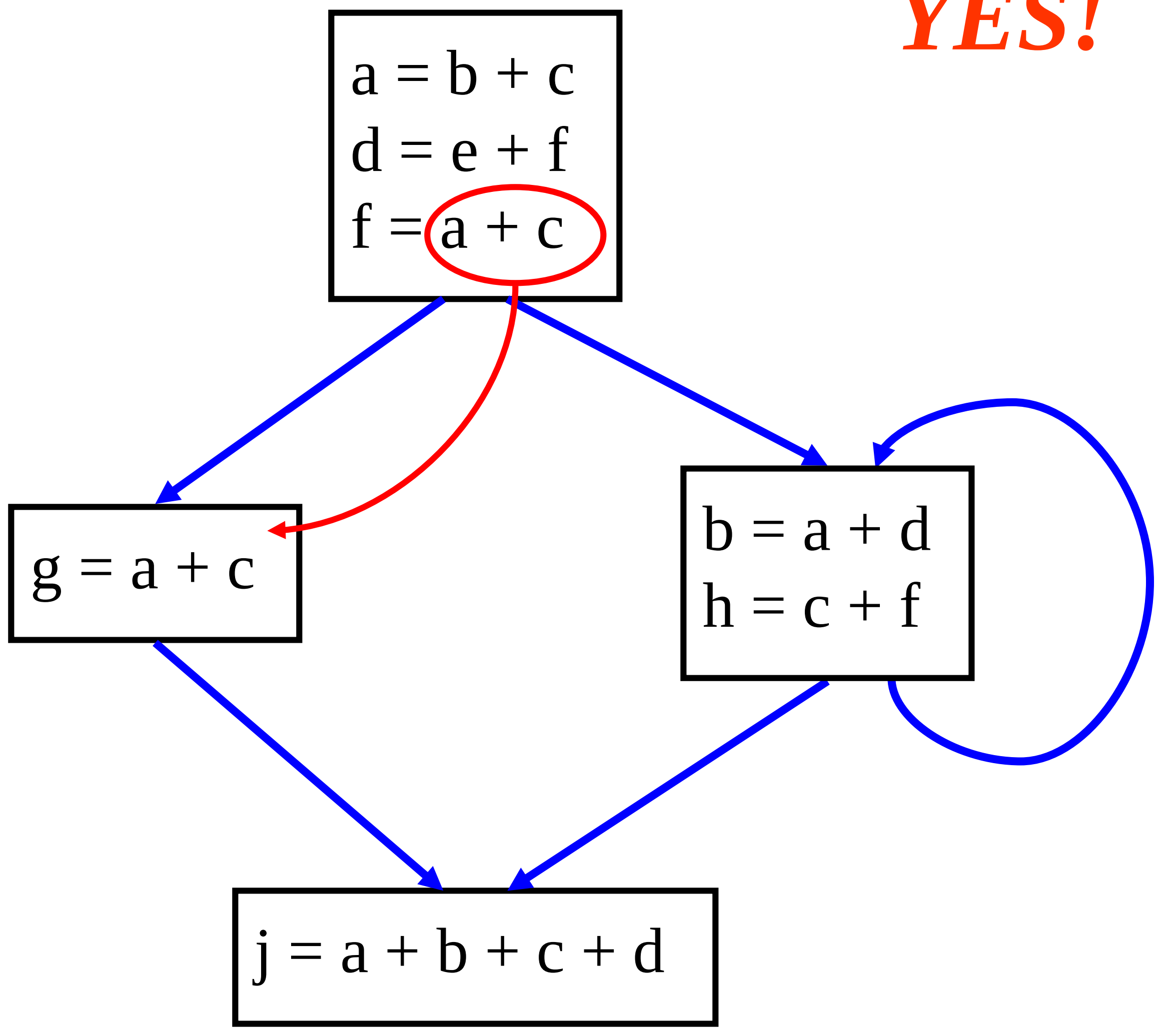


判断表达式 b+c 是否可用

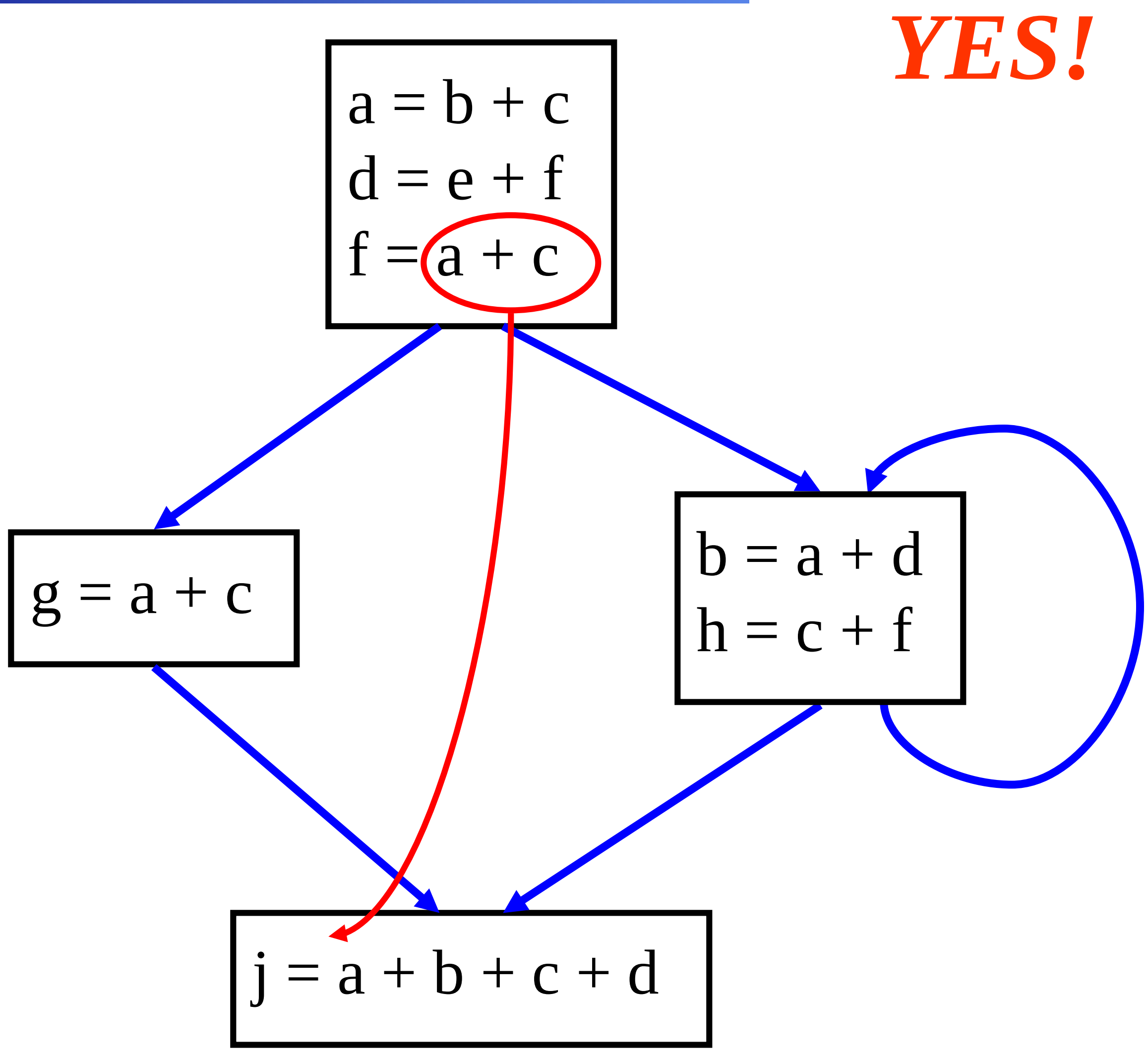


判断表达式 a+c 是否可用

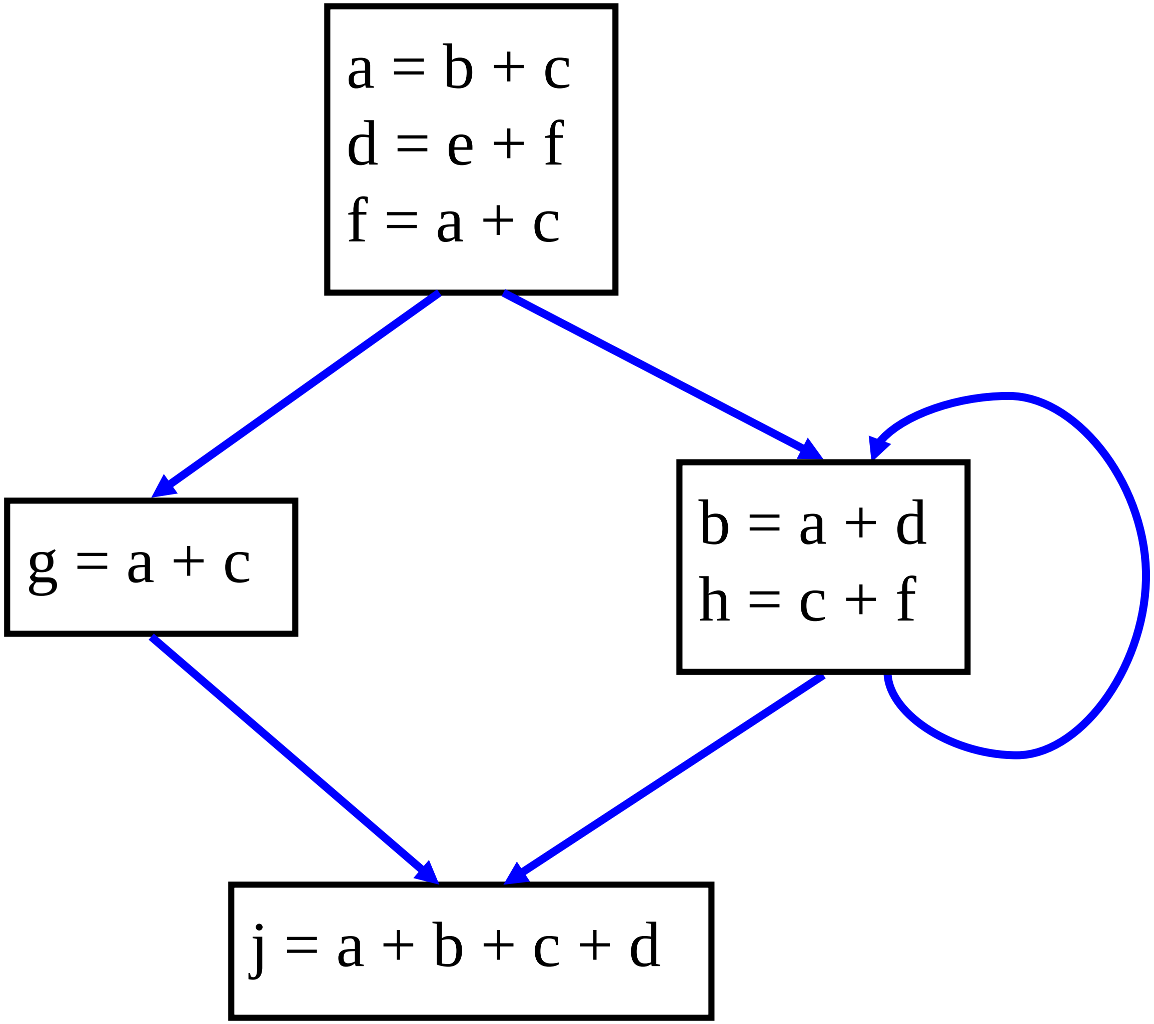
YES!



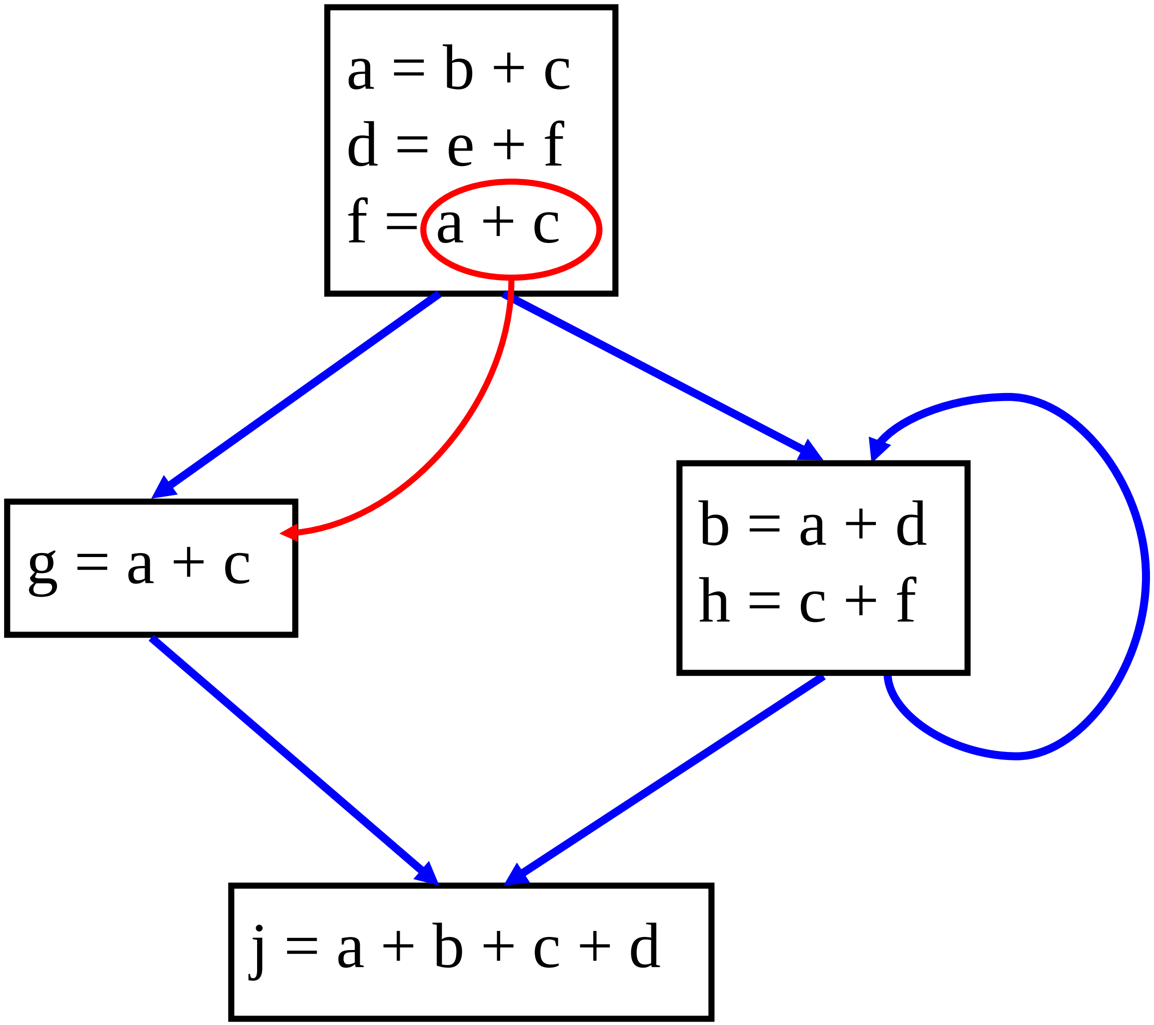
判断表达式 b+c 是否可用



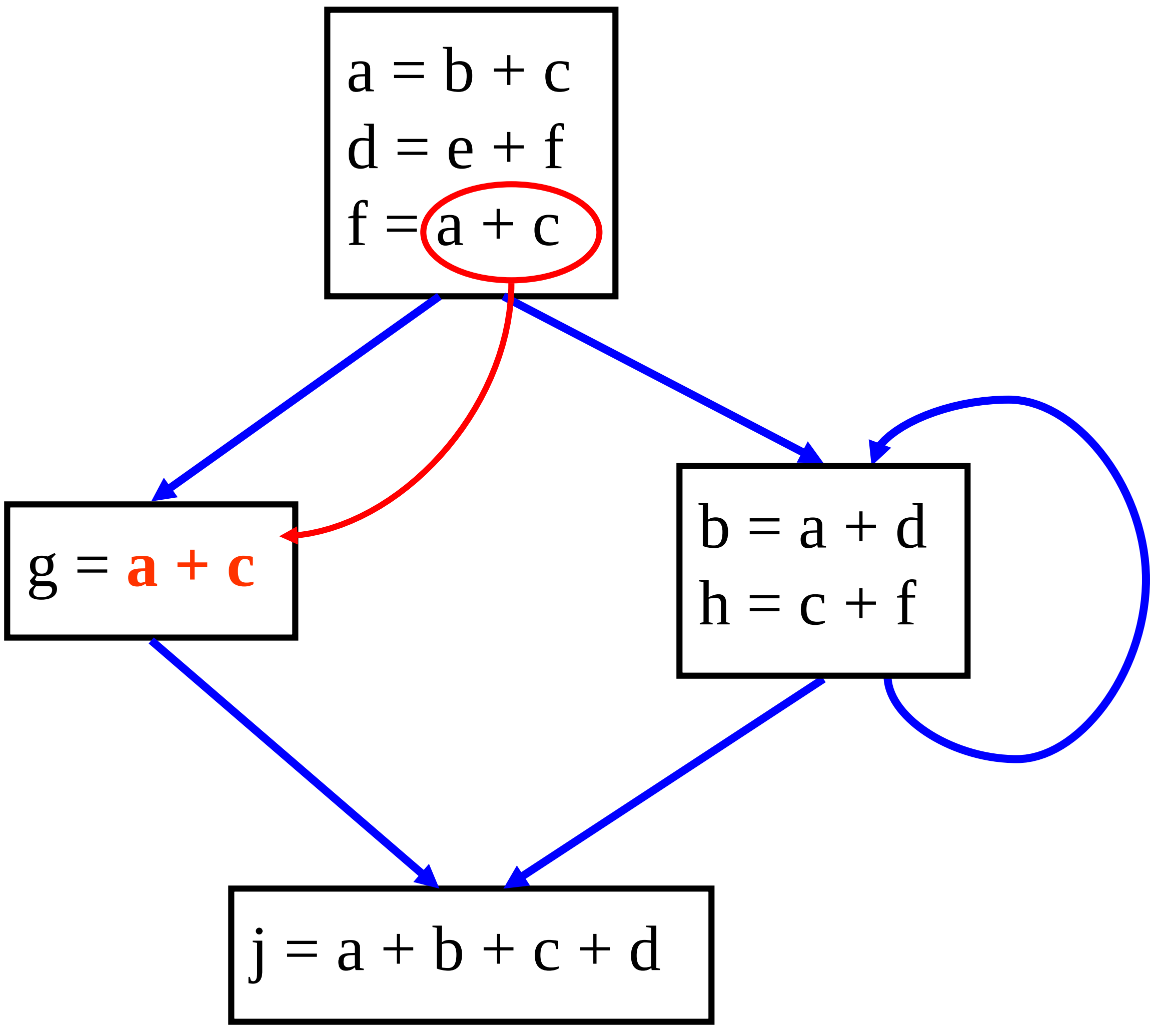
Use of Available Expressions



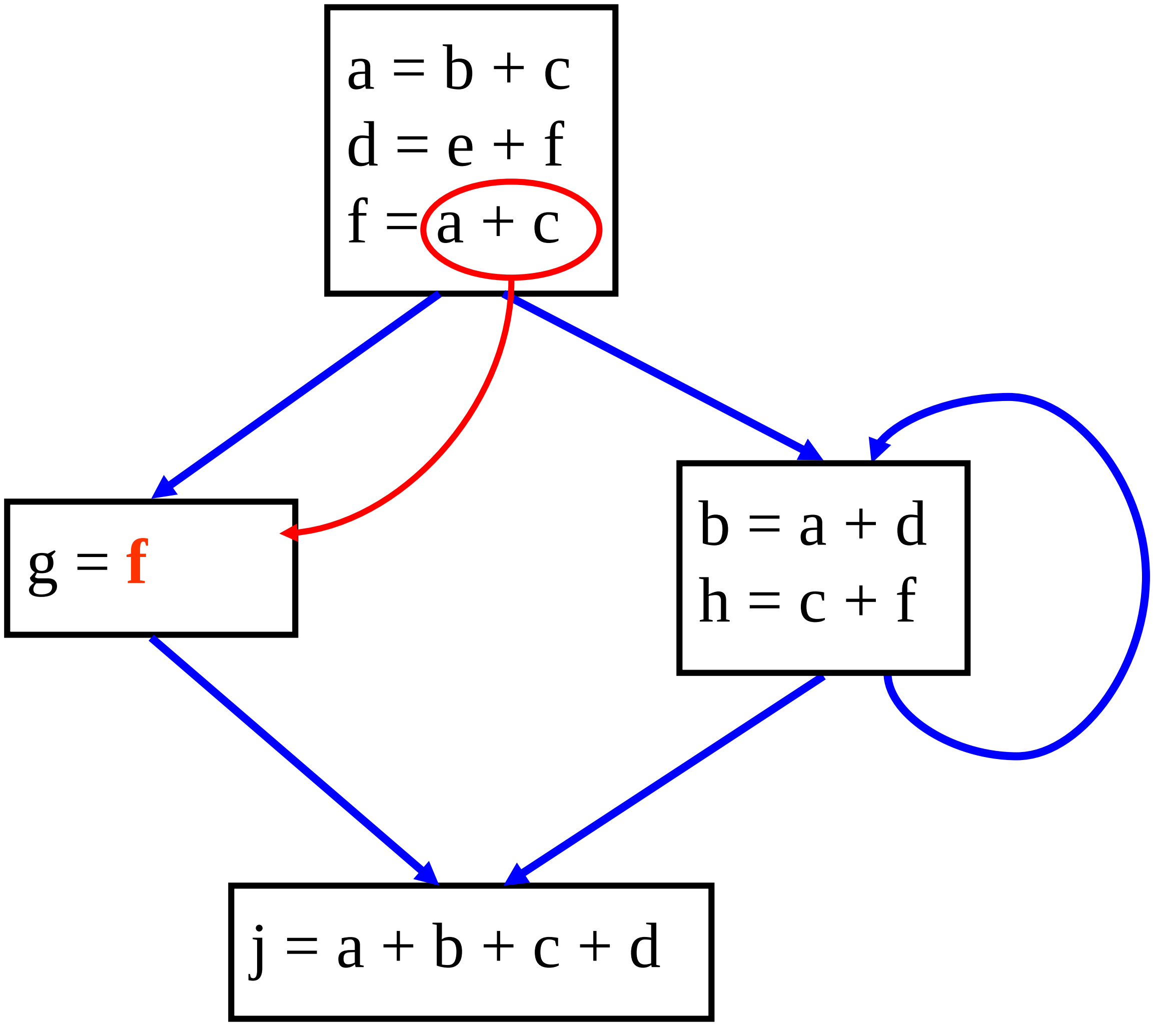
可用表达式的使用



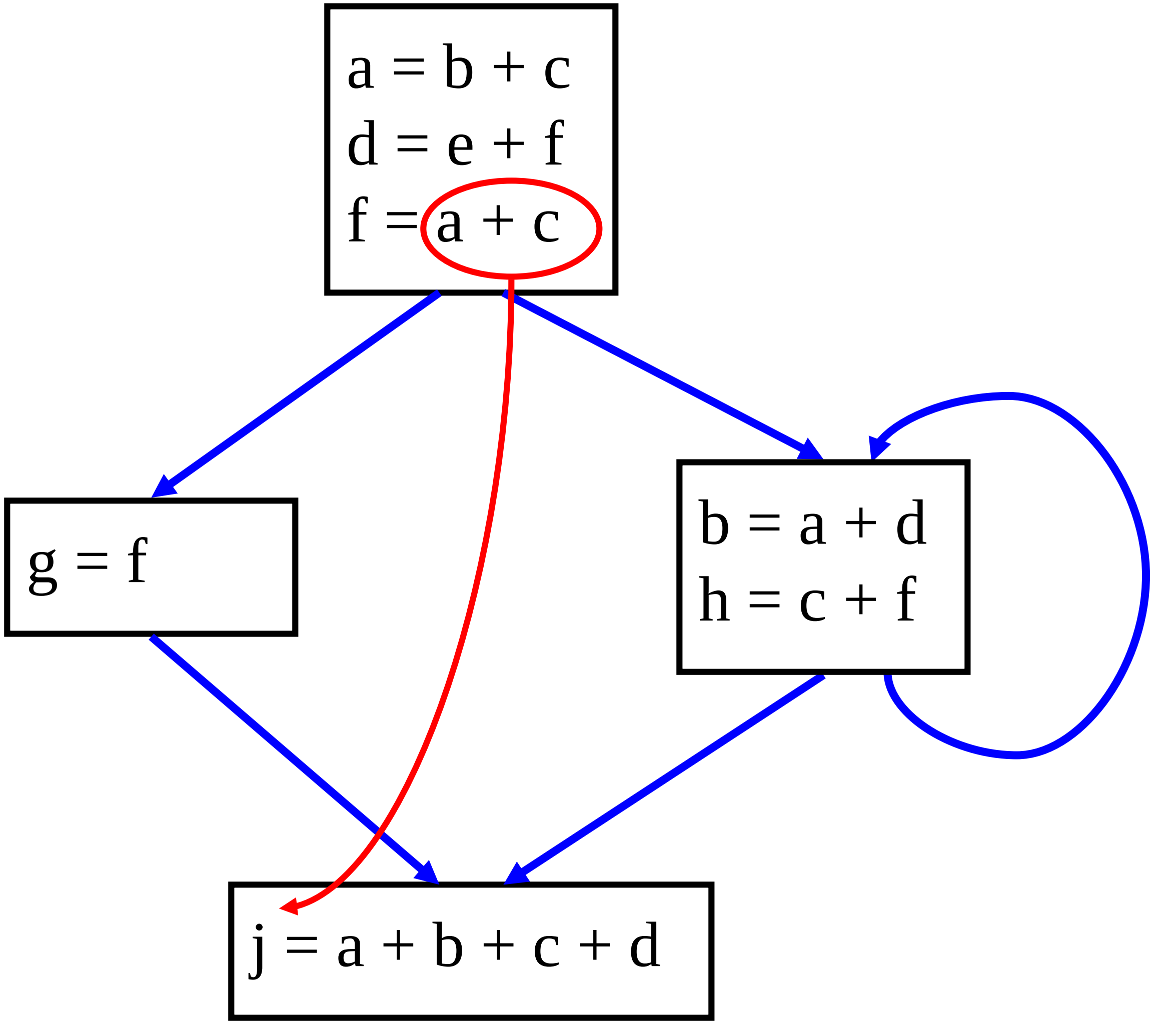
可用表达式的使用



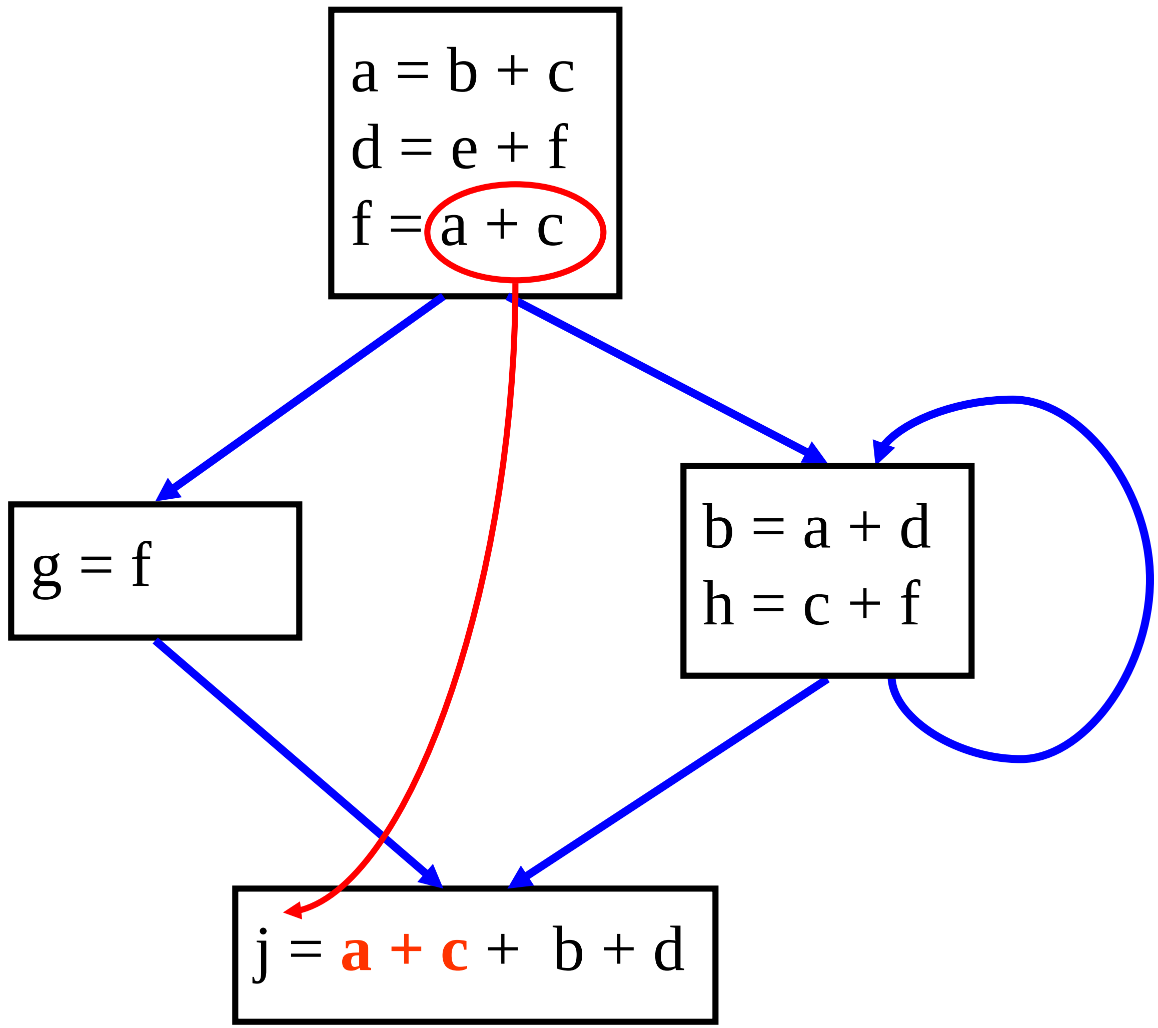
可用表达式的使用



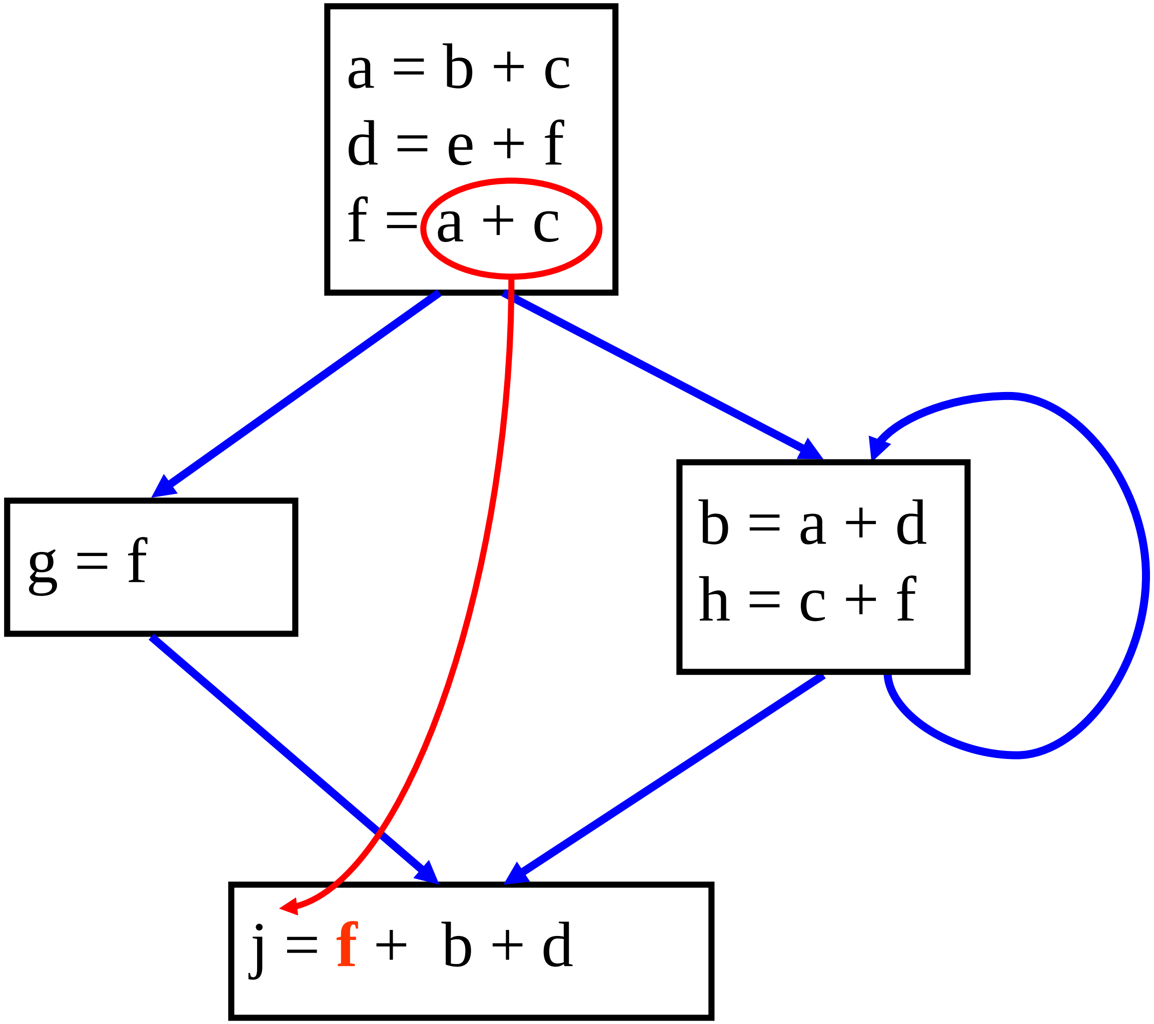
可用表达式的使用



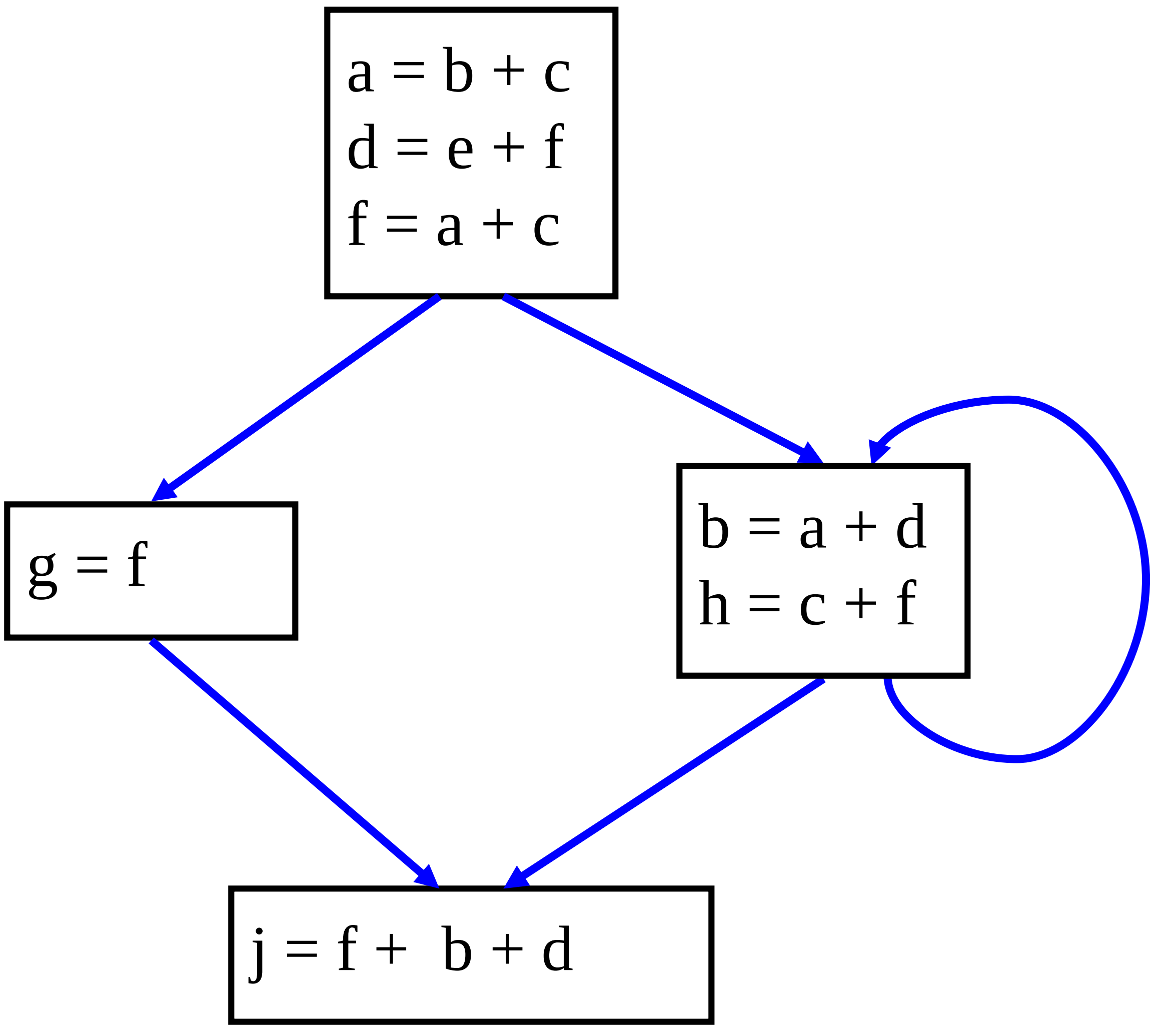
可用表达式的使用



可用表达式的使用



可用表达式的使用



可用表达式的计算

- 使用位向量来表示表达式的集合
- 每一个比特位代表一个表达式
- 调用数据流分析算法
- 与可达定义 (Reaching Definitions)的不同之处
  - 一个定义只要到达了一个基本块的任何一个前驱的结尾处，它就到达了该基本块的开头
  - 一个表达式在一个基本块的所有前驱的结尾处都可用，它才会在该基本块的开头可用

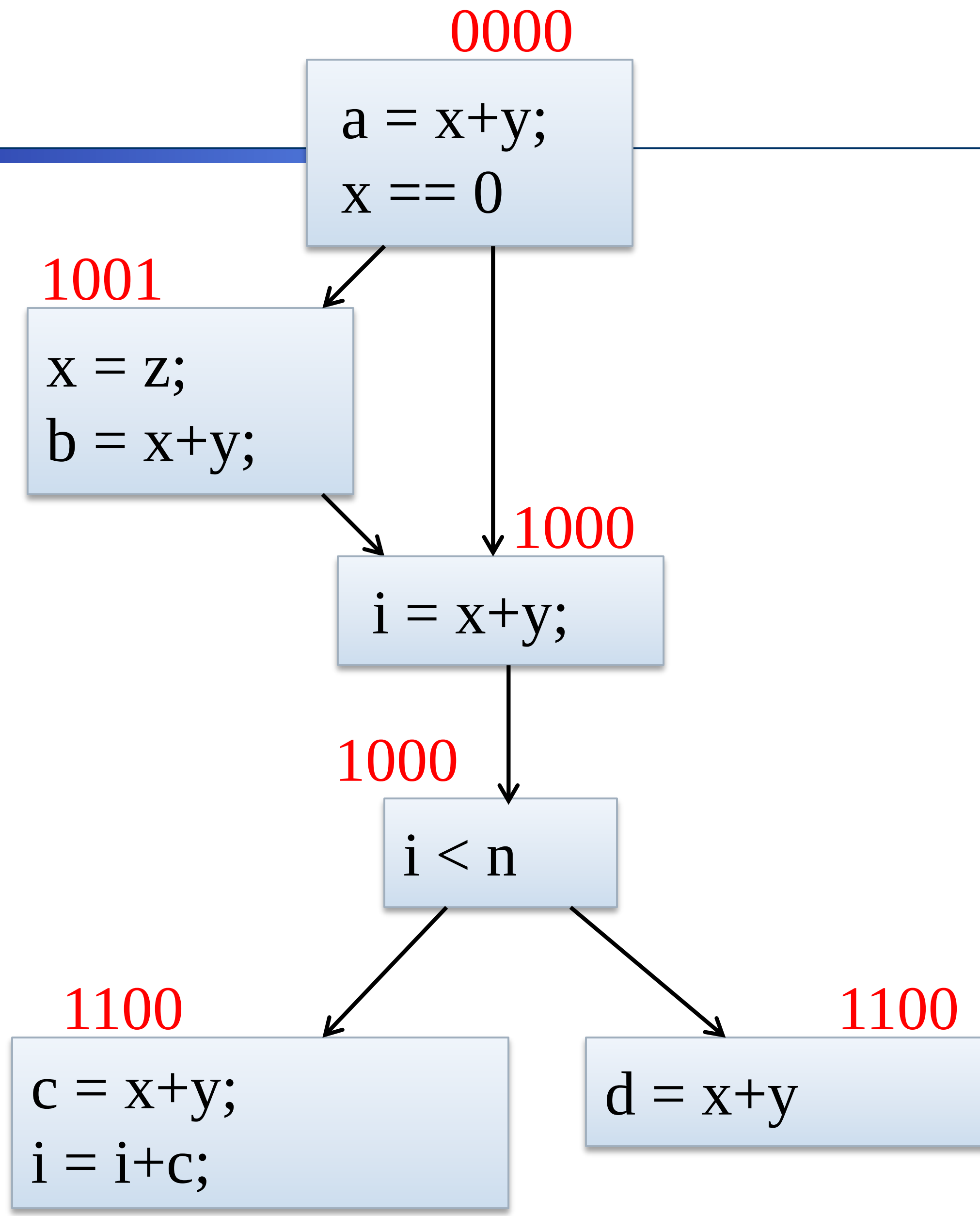
表达式集合

1: x+y

2: i<n

3: i+c

4: x==0



应用：消除全局公共子表达式

表达式集合

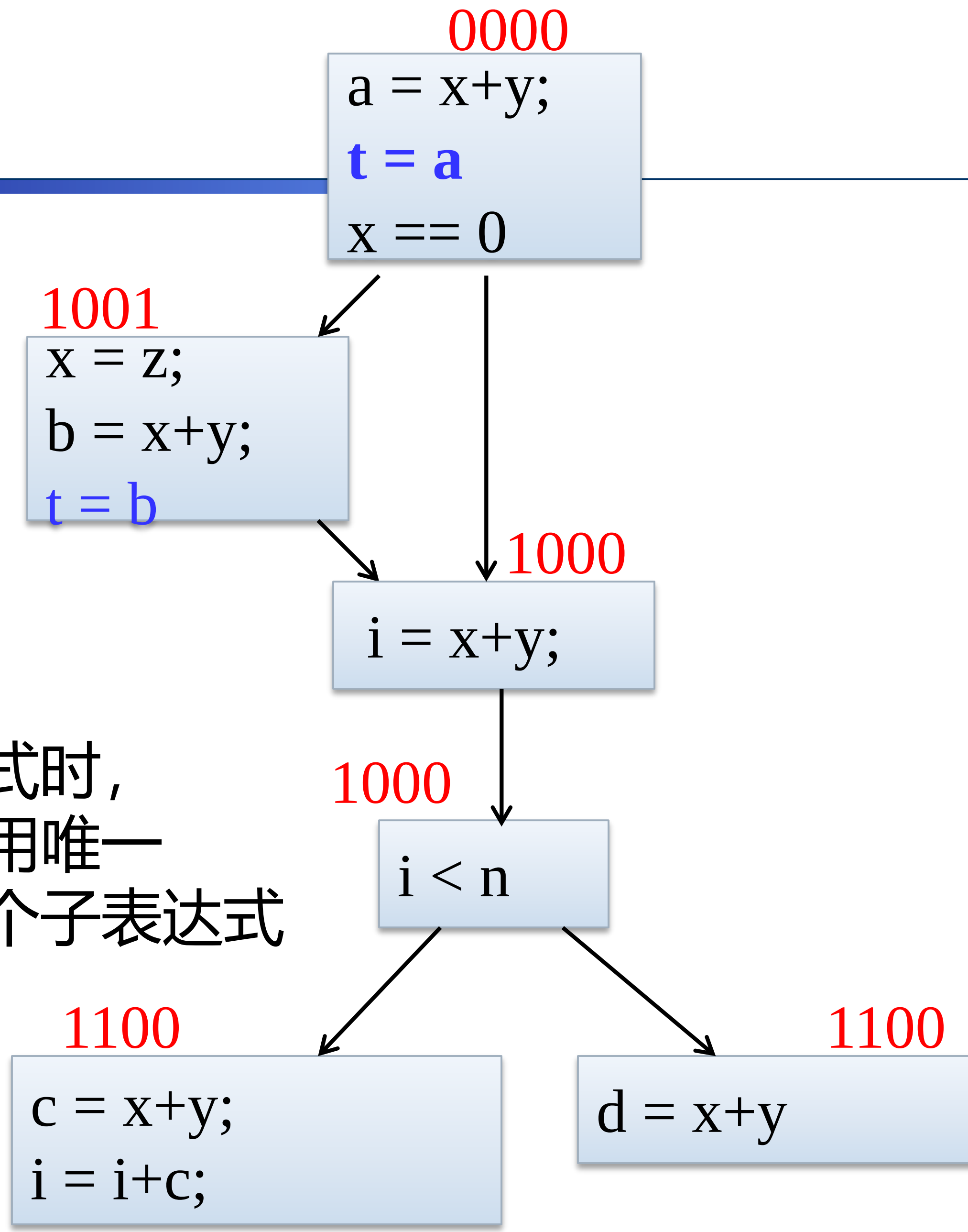
1: x+y

2: i<n

3: i+c

4: x==0

消除某个公共子表达式时，所有基本块中必须使用唯一的临时变量来代表这个子表达式



应用：消除全局公共子表达式

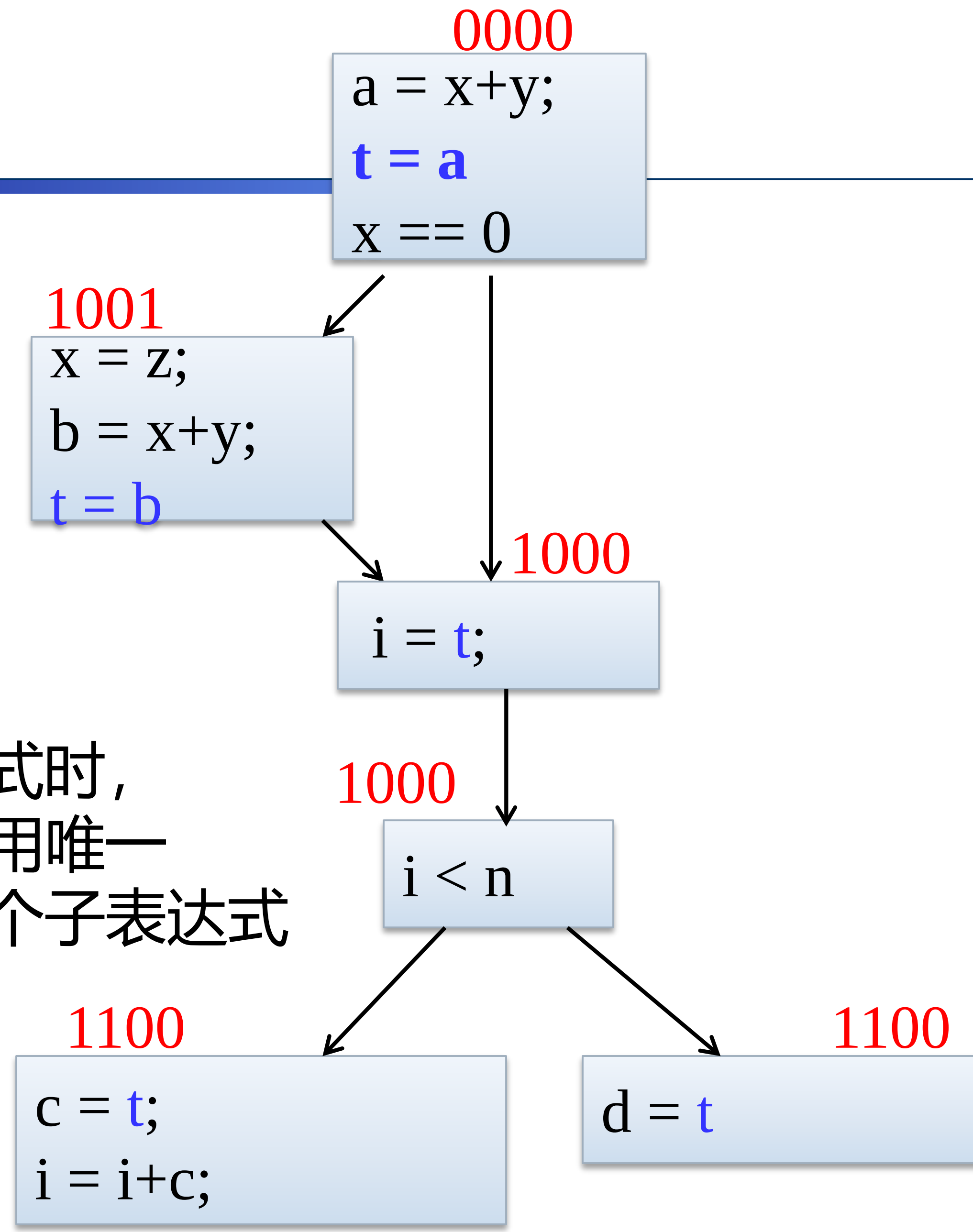
表达式集合

1: x+y

2: i<n

3: i+c

4: x==0



消除某个公共子表达式时，所有基本块中必须使用唯一的临时变量来代表这个子表达式

形式化分析

- 每一个基本块包含如下四个集合
  - **IN** -基本块开始处的可用表达式的集合
  - **OUT** -基本块结尾处的可用表达式的集合
  - **GEN** – 基本块生成的表达式的集合
  - **KILL** – 基本块杀死的表达式的集合
- **GEN[x = z; b = x+y] = 1000**
- **KILL[x = z; b = x+y] = 0001**
- Compiler 依次扫描每一个基本块来求得 GEN 和 KILL 集合

数据流方程

- $IN[b] = OUT[b_1] \cap \ldots \cap OUT[b_n]$
  - $b_1, \ldots, b_n$ 表示b在控制流图中的前驱
- $OUT[b] = (IN[b] - KILL[b]) \cup GEN[b]$
- $IN[entry] = 0000$

数据流方程组的求解

- 使用不动点算法
  - IN[entry] = 0000
  - 初始化 OUT[b] = 1111
- 重复的遍历控制流图，对每个基本块求解
  - $IN[b] = OUT[b_1] \cap \ldots \cap OUT[b_n]$
  - $OUT[b] = (IN[b] - KILL[b]) \cup GEN[b]$
- 使用**工作表（worklist）算法**到达不动点

求解可用表达式的算法

```
for all nodes n in N
    OUT[n] = E;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry }; // N = all nodes in graph

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = E; // E is set of all expressions
    for all nodes p in predecessors(n)
        IN[n] = IN[n] ∩ OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
            for all nodes s in successors(n)
            Changed = Changed U { s };
```

两个算法的对比

- 可达定义
  - 基本块汇合点处操作为 集合并
  - OUT[b] 初始化为 空集
- 可用表达式
  - 基本块汇合点处操作为 集合交
  - OUT[b] 初始化为 所有表达式的集合
- 是数据流算法框架中的两个例子

内容提要

- 代码优化概述
- 代码优化的主要方法
- 流图中的循环及其查找
- 数据流分析简介
  - 可达定义
  - 可用表达式
  - 活跃变量

活跃变量分析

- 变量 v 在程序点 p 活跃指的是
  - v 在沿着某条从 p 开始的路径上被使用到，并且
  - 在这次使用之前，在相同的路径上 v 没有被定义过

- 相反，v 在 p 处死亡指的是
  - v 在从 p 到流图出口的任意路径上都不被使用，或者
  - 从 p 开始的所有路径上都在使用 v 之前对其进行了重定义

变量活跃信息的作用

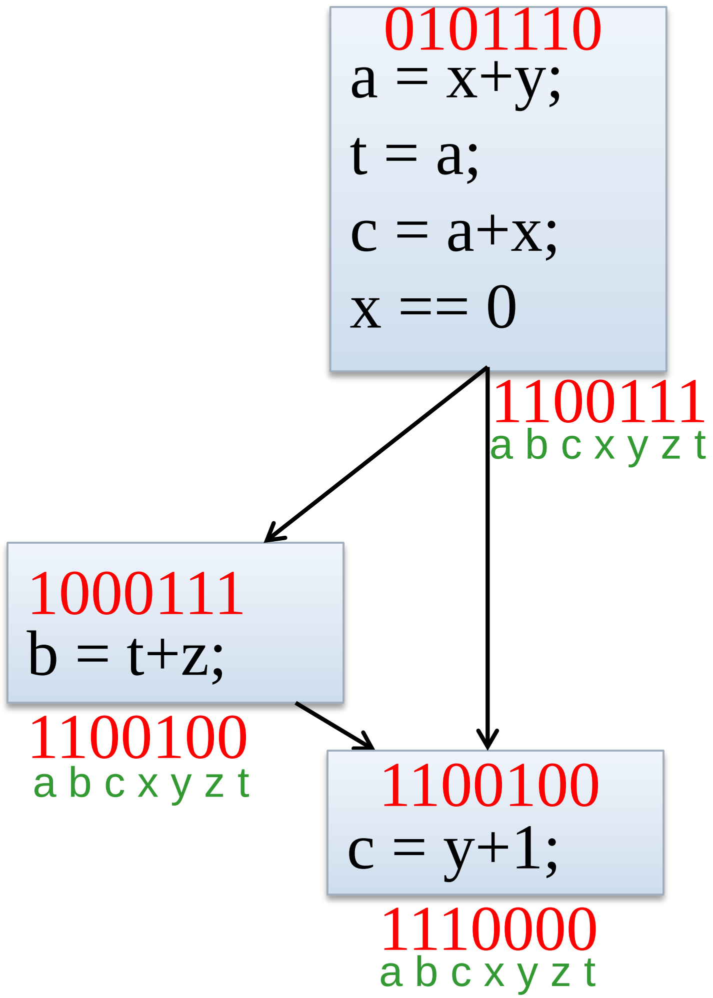
- 寄存器分配
  - 当一个变量已经死亡时，可以对它所占用的寄存器进行重新分配
- 死代码消除
  - 删除所有之后不会被使用到的变量的赋值

直观分析

- 模拟程序的执行过程
- 从CFG的出口开始自底向上分析
- 对每一个基本块，按照从结尾到开始的顺序计算活跃信息

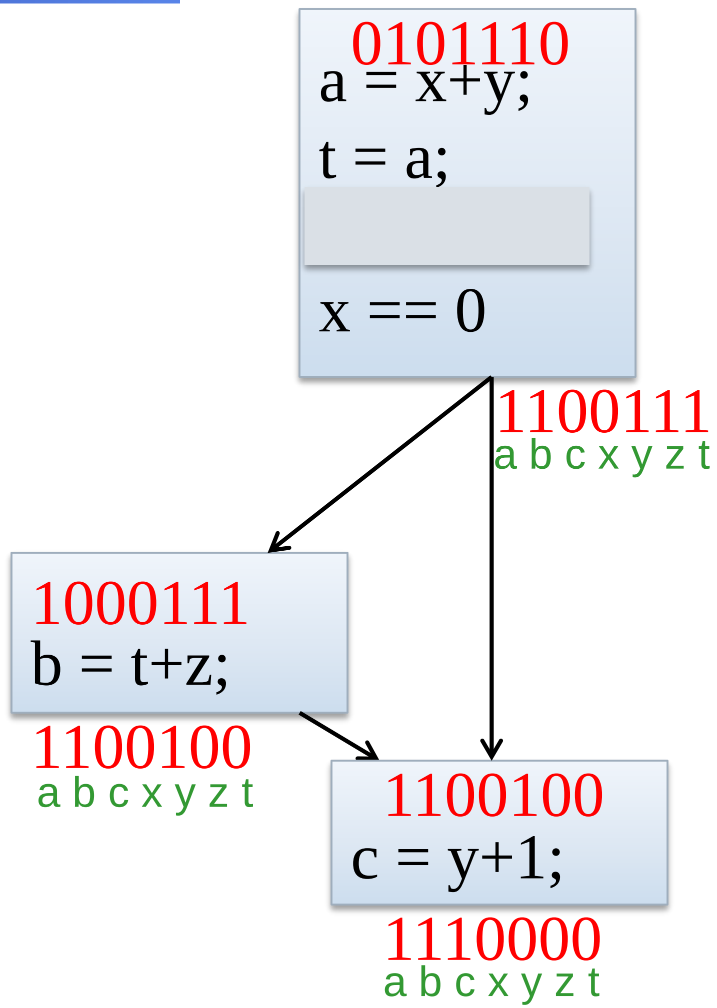
活跃变量分析举例

- 假设变量 a,b,c 在CFG出口处活跃
- 变量 x,y,z,t 不活跃
- 使用位向量来表示活跃变量
  - 按照 abcxyzt 的顺序



死代码消除

- 假设变量 a,b,c 在CFG出口处活跃
- 变量 x,y,z,t 不活跃
- 使用位向量来表示活跃变量
  - 按照 abcxyzt 的顺序



形式化分析

- 每一个基本块包含如下四个集合
  - **IN** – 基本块开始处的活跃变量集合
  - **OUT** –基本块结尾处的活跃变量集合
  - **USE** – 如下变量的集合，它们的值可能在基本块中先于任何可能对它们的定义被使用
  - **DEF** – 如下变量的集合，它们在基本块中的定义先于任何可能对它们的使用
- **USE[x = z; x = x+1;] = { z } (x 不在 USE 中)**
- **DEF[x = z; x = x+1; y = 1;] = {x, y}**
- **Compiler 依次扫描每一个基本块来求得 USE 和 DEF 集合**

Algorithm

```
for all nodes n in N - { Exit }
    IN[n] = emptyset;
OUT[Exit] = emptyset;
IN[Exit] = use[Exit];
Changed = N - { Exit };

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    OUT[n] = emptyset;
    for all nodes s in successors(n)
        OUT[n] = OUT[n] U IN[s];

    IN[n] = use[n] U (out[n] - def[n]);

    if (IN[n] changed)
        for all nodes p in predecessors(n)
            Changed = Changed U { p };
```

活跃变量分析同其他数据流分析算法的比较

- 后向分析
- 也具有转移函数
- 也具有交汇运算

三种算法的比较

可达定义

```
for all nodes n in N
    OUT[n] = emptyset;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry };

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = emptyset;
    for all nodes p in predecessors(n)
    IN[n] = IN[n] U OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
      for all nodes s in successors(n)
        Changed = Changed U { s };
```

可用表达式

```
for all nodes n in N
    OUT[n] = E;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry };

while (Changed != emptyset)
    choose a node n in Changed;
     Changed = Changed - { n };

     IN[n] = E;
     for all nodes p in predecessors(n)
    IN[n] = IN[n] ∩ OUT[p];

     OUT[n] = GEN[n] U (IN[n] - KILL[n]);

     if (OUT[n] changed)
         for all nodes s in successors(n)
           Changed = Changed U { s };
```

活跃变量分析

```
for all nodes n in N - { Exit }
    IN[n] = emptyset;
OUT[Exit] = emptyset;
IN[Exit] = use[Exit];
Changed = N - { Exit };

while (Changed != emptyset)
     choose a node n in Changed;
     Changed = Changed - { n };

    OUT[n] = emptyset;
     for all nodes s in successors(n)
     OUT[n] = OUT[n] U IN[s];

IN[n] = use[n] U (out[n] - def[n]);

     if (IN[n] changed)
      for all nodes p in predecessors(n)
         Changed = Changed U { p };
```

三种算法的比较

可达定义

```
for all nodes n in N
    OUT[n] = emptyset;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry };

while (Changed != emptyset)
        choose a node n in Changed;
        Changed = Changed - { n };

        IN[n] = emptyset;
        for all nodes p in predecessors(n)
        IN[n] = IN[n] U OUT[p];

       OUT[n] = GEN[n] U (IN[n] - KILL[n]);

        if (OUT[n] changed)
        for all nodes s in successors(n)
            Changed = Changed U { s };
```

可用表达式

```
for all nodes n in N
    OUT[n] = E;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry };

while (Changed != emptyset)
    choose a node n in Changed;
        Changed = Changed - { n };

        IN[n] = E;
        for all nodes p in predecessors(n)
        IN[n] = IN[n] ∩ OUT[p];

        OUT[n] = GEN[n] U (IN[n] - KILL[n]);

        if (OUT[n] changed)
            for all nodes s in successors(n)
            Changed = Changed U { s };
```

三种算法的比较

可达定义

```
for all nodes n in N
    OUT[n] = emptyset;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry };

while (Changed != emptyset)
        choose a node n in Changed;
        Changed = Changed - { n };

        IN[n] = emptyset;
        for all nodes p in predecessors(n)
        IN[n] = IN[n] U OUT[p];

        OUT[n] = GEN[n] U (IN[n] - KILL[n]);

        if (OUT[n] changed)
        for all nodes s in successors(n)
            Changed = Changed U { s };
```

活跃变量分析

```
for all nodes n in N
    IN[n] = emptyset;
OUT[Exit] = emptyset;
IN[Exit] = use[Exit];
Changed = N - { Exit };

while (Changed != emptyset)
        choose a node n in Changed;
        Changed = Changed - { n };

    OUT[n] = emptyset;
        for all nodes s in successors(n)
        OUT[n] = OUT[n] U IN[s];

    IN[n] = use[n] U (out[n] - def[n]);

        if (IN[n] changed)
        for all nodes p in predecessors(n)
            Changed = Changed U { p };
```

数据流分析总结

- 数据流分析
  - 控制流图
  - IN[b], OUT[b], 转移函数, 交汇运算
- 数据流分析的应用
  - 可达定义/常量传播
  - 可用表达式/公共子表达式消除
  - 活跃变量分析/死代码消除

本章小结

- 对各类优化方法的理解
  - 包括常量合并、公共子表达式删除、复写传播、死代码删除、循环优化（代码外提、归纳变量删除、强度削弱）等
- 掌握流图中循环识别的算法
- 对数据流分析框架的理解
  - 掌握三种数据流分析的基本算法

作业

- Ex. 9.1.1, 9.1.4
- Ex. 9.2.1