并行编译与优化
Parallel Compiler and Optimization

计算机研究所编译系统室

Lecture Seven: Scalar Optimization

第七课：标量优化

回顾: 数据流分析

	到达定值分析	活跃变量分析
数据流方向D	前向	后向
值集V	定值的集合	变量的集合
交汇运算∧	∪	∪
传递函数F	$f_B(x) = gen[B] \cup (x - kill[B])$	$f_B(x) = use[B] \cup (x - def[B])$
数据流方程	$in[B] = \cup\ out[pred(B)]$ $out[B] = f_B(in[B])$	$out[B] = \cup\ in[succ(B)]$ $in[B] = f_B(out[B])$
边界条件	$out[entry] = \varnothing$	$in[exit] = \varnothing$
初始值	$out[B] = \varnothing$	$in[B] = \varnothing$

回顾: 数据流分析迭代算法

```
out[Entry] = V_entry;
for (each B∈N-{Entry})
   out[B] = T //初值
while(change to any out occur){
   for(each B∈N-Entry){
      in[B] = ∧_{p∈pred[B]} out[P];
      out[B] = f_B(in[B]);
   }
}
```

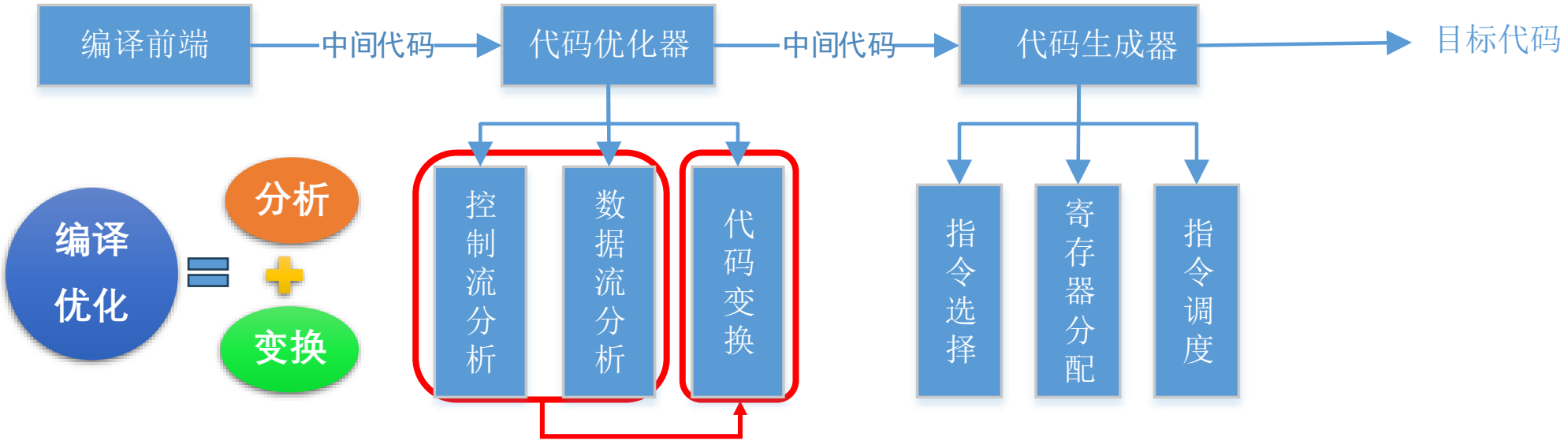
前向数据流分析的迭代算法

```
in[Exit] = V_exit;
for (each B∈N-{Exit})
   in[B] = T //初值
while(change to any in occur){
   for(each B∈N-Exit){
      out[B] = ∧_{s∈succ[B]} in[S];
      in[B] = f_B(out[B]);
   }
}
```

后向数据流分析的迭代算法

冗余代码

- 考虑访问数组元素a[i][j]对应的IR
- 中间代码中存在大量冗余代码，如
  - 公用子表达式
  - 循环不变量
  - 死代码
  - 部分冗余……



冗余优化

COMP CCRG Open MP

- 一些常用的冗余优化技术
  - 常数传播/复制传播(Const Propagation/Copy Propagation)
  - 公用子表达式删除(Common Subexpression Elimination, CSE)
  - 值编号(Value numbering, local and global)
  - 循环不变量外提(Loop-Invariant Code Motion, LICM)
  - 死代码删除(Dead Code Elimination, DCE)
  - 代码提升(Code Hoisting)
  - 部分冗余删除(Partial Redundancy Elimination, PRE)

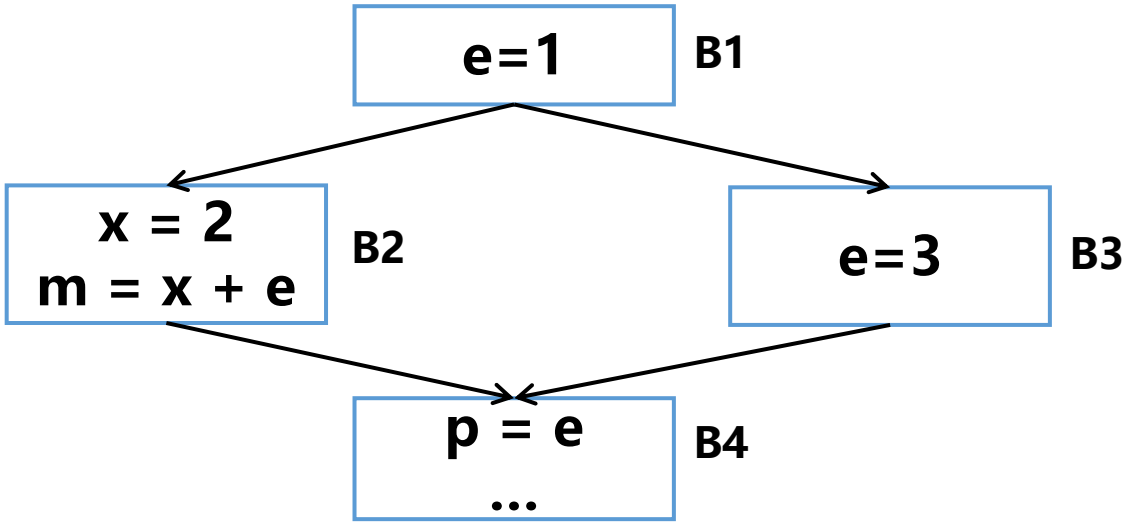
学习内容

- 7.1 常数传播
- 7.2 公用子表达式删除
- 7.3 值编号
- 7.4 循环不变量外提
- 7.5 死代码删除

学习目标

- 掌握冗余优化中的公用子表达式删除和死代码删除优化方法，熟悉常数传播、值编号和循环不变量外提优化方法

7.1.1 常数传播分析



- 在每个基本块的边界，判断哪些变量是常数?
  - 如果是，值是多少?
- 如果变量x定值为常数C，在x值没有发生改变前，用C来替换对x的使用
- 常数传播分析是一个前向数据流问题

7.1.1 常数传播分析

```
out[Entry] = V_entry;
for (each B∈N-{Entry})
   out[B] = UNDEF //初值
while(change to any out occur){
   for(each B∈N-Entry){
      in[B] = ∧_{p∈pred[B]} out[P];
      out[B] = f_B(in[B]);
   }
}
```

- 值集V中变量可能的取值类型
  - 所有符合该变量的类型的常量值c
  - 值NAC(not a constant)，表示非常量值
  - 值UNDEF，表示未定义
- 常数传播是一个“必然”问题
  - 数据流信息必须为真，而不是可能为真
  - 因此计算的是下界，∧是“集合交”

7.1.1 常数传播分析

```
out[Entry] = V_entry;
for (each B∈N-{Entry})
   out[B] = UNDEF //初值
while(change to any out occur){
   for(each B∈N-Entry){
      in[B] = ∧_{p∈pred[B]} out[P];
      out[B] = f_B(in[B]);
   }
}
```

- 边界条件V_{entry}
  - 对所有变量x，out[entry,x]=UNDEF
  - 表示所有变量x在程序开始执行的时候是没有定值的
- 初始值
  - 对所有变量x，out[B,x] = UNDEF
- in[S,x]和out[S,x]分别表示语句S前后变量x的信息, 有

$$out[S,x] = f_s(in[S,x])$$

7.1.1 常数传播分析

- 按以下方式定义传递函数 f_s
  - 如果S不是赋值语句，那么f_s是单元函数，即$f_s(x) = x$
  - 如果S是一个对变量x的赋值语句，S: x ←RHS(右部)，对所有v ≠ x的变量，有out[S,v] = in[S,v]
  - 对变量x，out[S,x]的定义如下
    1. 如果RHS 是常量c, 则out[S,x] = c
    2. 如果RHS 形如y ⊗ z, 则:
       - 如果in[S,y]和in[S,z]都是常量值，则out[S,x]= in[S,y] ⊗ in[S,z]
       - 如果in[S,y]和in[S,z]中有一个是NAC，那么out[S,x]=NAC
       - 否则，out[S,x]=UNDEF
    3. 如果RHS是其他表达式(如函数调用)，则out[S,x] =NAC

7.1.2 常数传播方法

首先进行常数传播分析

对每个基本块B∈N，执行{

对每条instr∈B，如果instr使用x，并且在B的入口点x是常数c{

① 判断在该instr入口处x是否为常数c

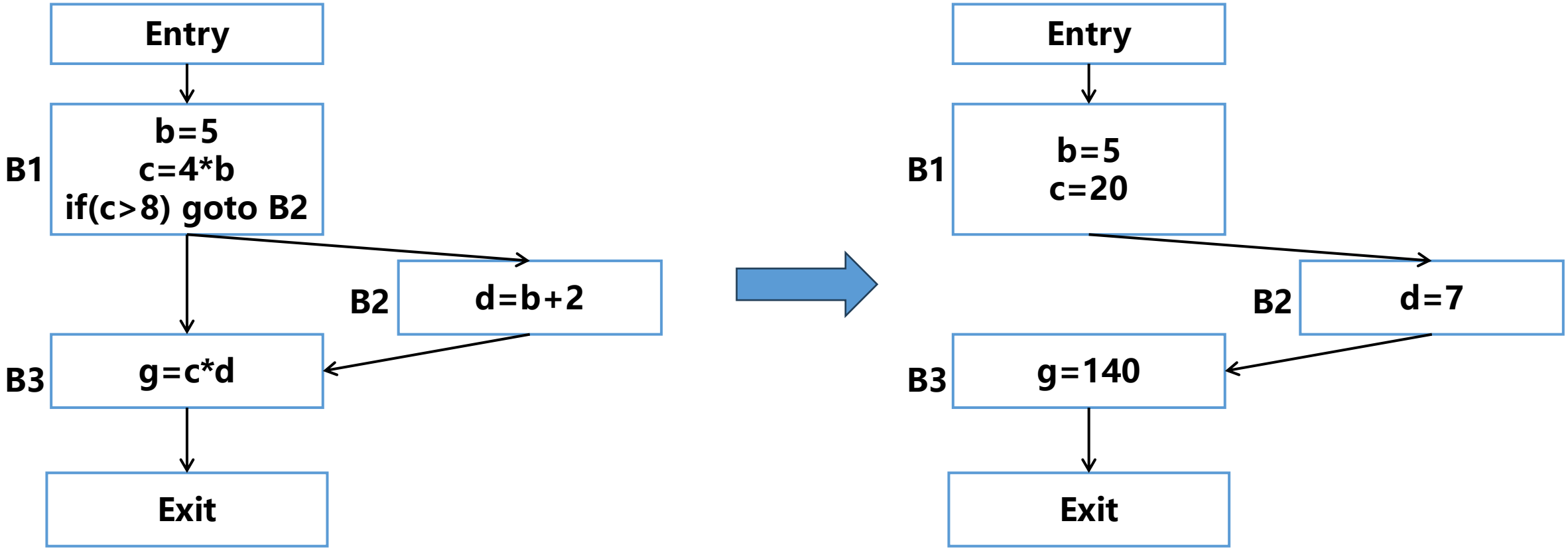
② 替换instr中的x为c

}

}

7.1.2 常数传播方法

- 示例: 常数传播+常数折叠



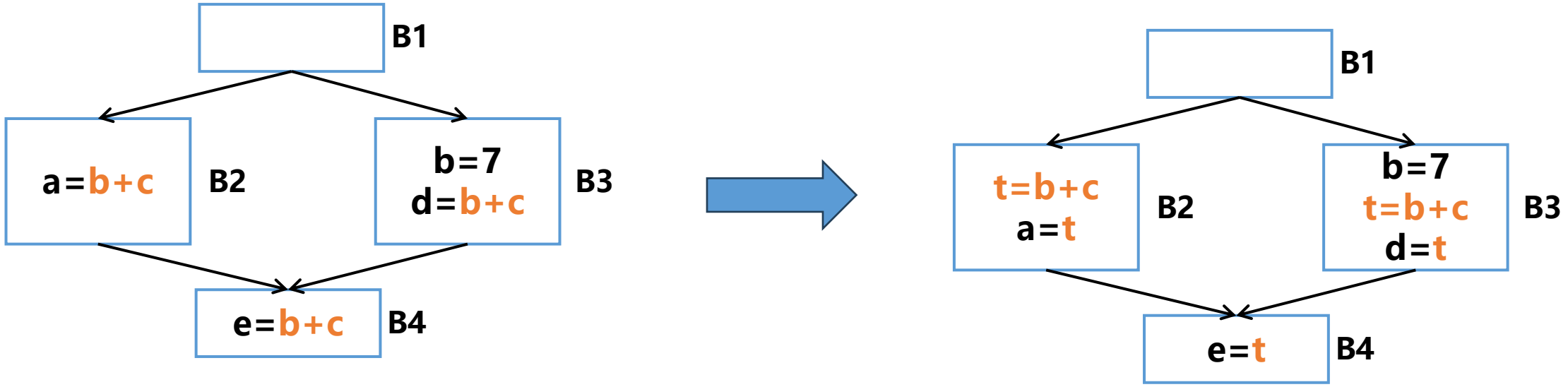
量变引起质变

学习内容

- 7.1 常数传播
- 7.2 公用子表达式删除
- 7.3 值编号
- 7.4 循环不变量外提
- 7.5 死代码删除

7.2.1 公用子表达式

- 如果表达式E在某次出现之前已经被计算过，并且E中操作数的值从那次计算之后就一直没有被改变，则E的该次出现就称为一个公用子表达式(common subexpression)
- 公用子表达式删除(common subexpression elimination, CSE)就是用保留的结果替换重复计算



7.2.2 可用表达式

- 如果一个表达式x⊗y在点p满足下列条件，则该表达式在点p可用(*available*)
  - 从entry结点到点p的每条路径都对表达式x⊗y进行计算
  - 并且从最后一次这样的计算到点p之间，没有再次对x或y的定值
- 可用表达式分析
  - 分析每个基本块边界(入口点和出口点)的所有可用表达式
  - 分析的结果用于公用子表达式删除

7.2.2 可用表达式

- 如果一个基本块一定对表达式x+y求值，并且之后没有在对x或y定值，则该基本块B生成(或计算)表达式x+y
- 如果一个基本块B对x或y赋值，并且之后没有再重新计算x+y，则该基本块B“杀死” 表达式x+y

语句	可用表达式
	∅
a = b + c	
b = a - d	
c = b + c	
d = a - d	?

7.2.2 可用表达式

- 如果一个基本块B对x或y赋值，并且之后没有再重新计算x+y，则该基本块B“杀死” 表达式x+y
- 如果一个基本块一定对表达式x+y求值，并且之后没有在对x或y定值，则该基本块B生成(或计算)表达式x+y

语句	可用表达式
	∅
a = b + c	
	{b + c}
b = a - d	
c = b + c	
d = a - d	

7.2.2 可用表达式

- 如果一个基本块B对x或y赋值，并且之后没有再重新计算x+y，则该基本块B“杀死” 表达式x+y
- 如果一个基本块一定对表达式x+y求值，并且之后没有在对x或y定值，则该基本块B生成(或计算)表达式x+y

语句	可用表达式
	∅
a = b + c	
	{b + c}
b = a - d	
	{a - d}
c = b + c	
d = a - d	

7.2.2 可用表达式

- 如果一个基本块B对x或y赋值，并且之后没有再重新计算x+y，则该基本块B“杀死” 表达式x+y
- 如果一个基本块一定对表达式x+y求值，并且之后没有在对x或y定值，则该基本块B生成(或计算)表达式x+y

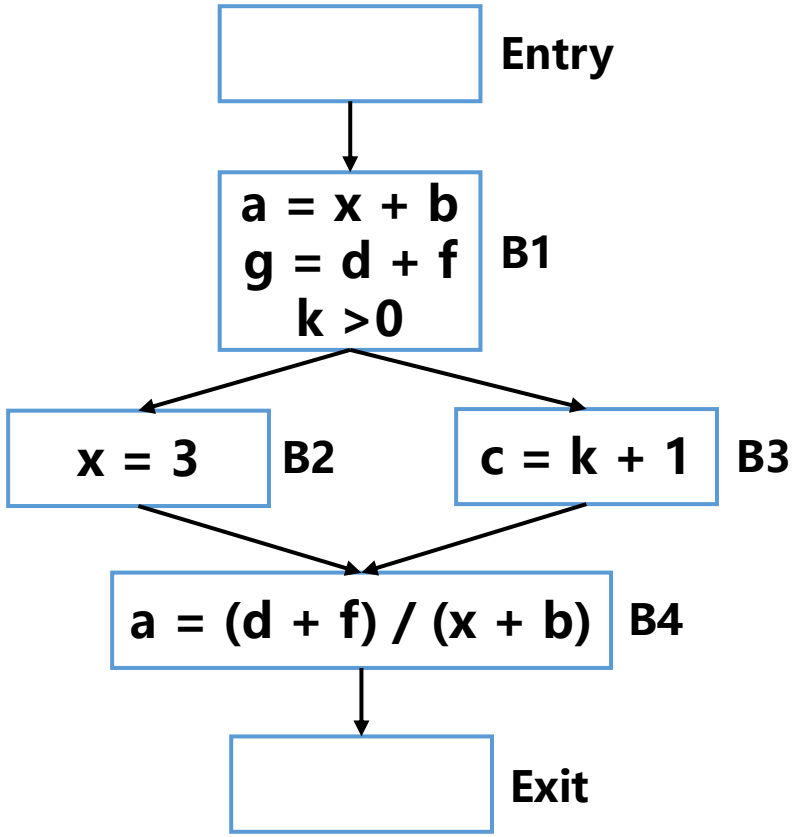
语句	可用表达式
	∅
a = b + c	
	{b + c}
b = a - d	
	{a - d}
c = b + c	
	{a - d}
d = a - d	

7.2.2 可用表达式

- 如果一个基本块B对x或y赋值，并且之后没有再重新计算x+y，则该基本块B“杀死” 表达式x+y
- 如果一个基本块一定对表达式x+y求值，并且之后没有在对x或y定值，则该基本块B生成(或计算)表达式x+y

语句	可用表达式
	∅
a = b + c	
	{b + c}
b = a - d	
	{a - d}
c = b + c	
	{a - d}
d = a - d	
	∅

7.2.3 可用表达式分析

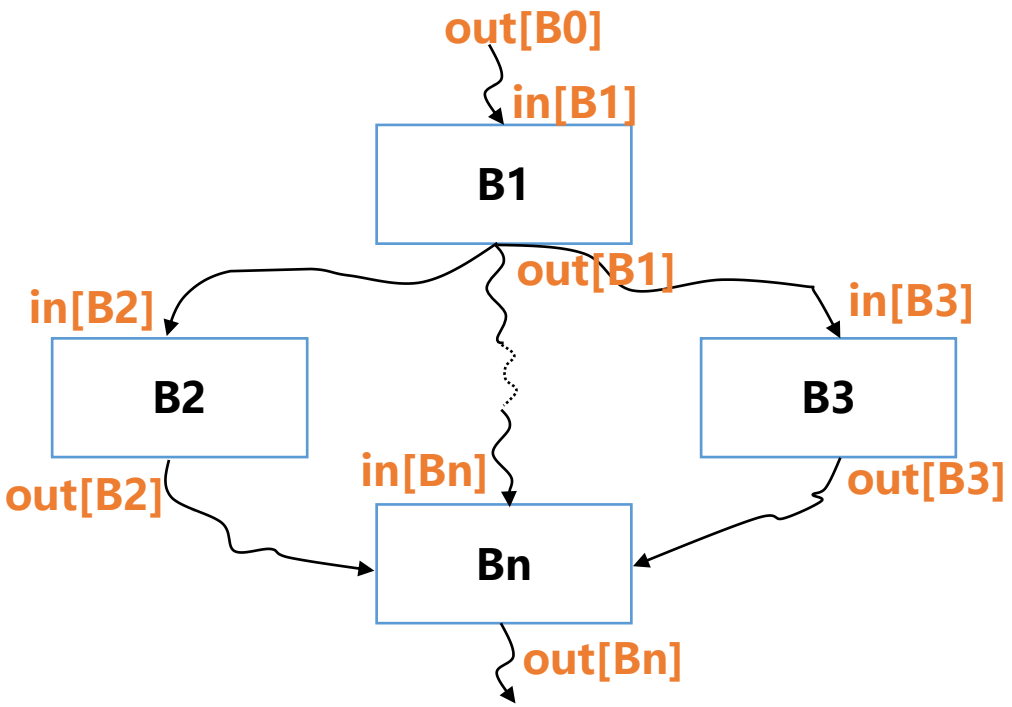


- **in[B]：基本块B入口点的可用表达式集合**
- **out[B]：基本块B出口点的可用表达式集合**
- **eval[B]：在基本块B中计算的，并且在B的出口点仍旧可用的表达式集合**
- **kill[B]：被基本块B杀死的表达式集合**
- **第一步: 计算各个基本块的eval[B]和kill[B]**

eval[B1] = {x+b, d+f}	kill[B1] = ∅
eval[B2] = ∅	kill[B2] = x+b
eval[B3] = {k+1}	kill[B3] = ∅
eval[B4] = {d+f, x+b, (d+f)/(x+b)}	kill[B4] = ∅

7.2.3 可用表达式分析

- 第二步: 建立数据流方程
  - 前向数据流分析，寻找的是从entry结点到点p的可用表达式
  - ∧是“集合交”



$$in[B] = \bigcap_{P \in pred[B]} out[P]$$

$$out[B] = eval[B] \cup (in[B] - kill[B])$$

7.2.3 可用表达式分析

COMP CCRG Open MP

第三步: 使用不动点法求解数据流方程

out[Entry]=∅

每个基本块B∈N–{Entry}

　　out[B]=U //出现在程序中的语句右部的表达式的全集

change = true

while(changes) { //循环，直到所有基本块的out集合都不再发生变化

　　change = false

　　对每个基本块B∈N–{Entry} {　　求解不动点

　　　　oldout=out[B]

　　　　in[B]= $\bigcap_{P \in pred[B]} out[P]$

　　　　out[B]= eval[B]∪(in[B]-kill[B])

　　　　if(out[B]≠oldout) change=true;

　　}

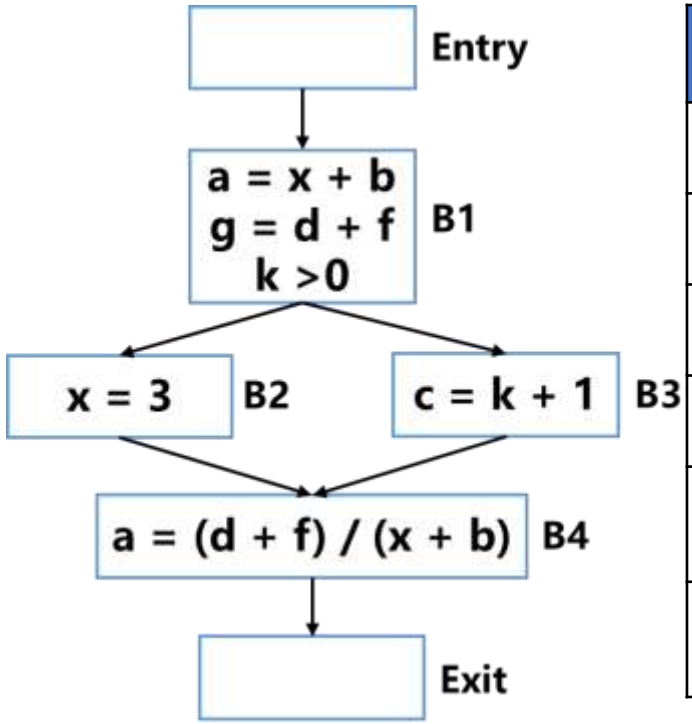
}

7.2.3 可用表达式分析

第三步: 使用不动点法求解数据流方程

$$in[B] = \bigcap_{P \in pred[B]} out[P]$$

$$out[B] = eval[B] \cup (in[B] - kill[B])$$



第一次迭代

BB	eval[B]	kill[B]	$out[B]^0$	$in[B]^1$	$out[B]^1$
Entry	∅	∅	∅	∅	∅
B1	{x+b, d+f}	∅	U	∅	{x+b, d+f}
B2	∅	{x+b}	U	{x+b, d+f}	{d+f}
B3	{k+1}	∅	U	{x+b, d+f}	{k+1, x+b, d+f}
B4	{d+f, x+b, (d+f)/(x+b)}	∅	U	{d+f}	{d+f, x+b, (d+f)/(x+b)}
Exit	∅	∅	U	{d+f, x+b, (d+f)/(x+b)}	{d+f, x+b, (d+f)/(x+b)}

7.2.3 可用表达式分析

第三步: 使用不动点法求解数据流方程

$$in[B] = \bigcap_{P \in pred[B]} out[P]$$

$$out[B] = eval[B] \cup (in[B] - kill[B])$$

第二次迭代

BB	eval[B]	kill[B]	out[B]^0	in[B]^1	out[B]^1	in[B]^2	out[B]^2
Entry	∅	∅	∅	∅	∅	∅	∅
B1	{x+b, d+f}	∅	U	∅	{x+b, d+f}	∅	{x+b, d+f}
B2	∅	{x+b}	U	{x+b, d+f}	{d+f}	{x+b, d+f}	{d+f}
B3	{k+1}	∅	U	{x+b, d+f}	{k+1, x+b, d+f}	{x+b, d+f}	{k+1, x+b, d+f}
B4	{d+f, x+b, (d+f)/(x+b)}	∅	U	{d+f}	{d+f, x+b, (d+f)/(x+b)}	{d+f}	{d+f, x+b, (d+f)/(x+b)}
Exit	∅	∅	U	{d+f, x+b, (d+f)/(x+b)}	{d+f, x+b, (d+f)/(x+b)}	{d+f, x+b, (d+f)/(x+b)}	{d+f, x+b, (d+f)/(x+b)}

out[B]^2= out[B]^1, 到达不动点

对比

	到达定值分析	活跃变量分析	可用表达式分析
数据流方向D	前向	后向	前向
值集V	定值的集合	变量的集合	表达式的集合
交汇运算∧	∪	∪	∩
传递函数F	$f_B(x) = gen[B] \cup (x - kill[B])$	$f_B(x) = use[B] \cup (x - def[B])$	$f_B(x) = eval[B] \cup (x - kill[B])$
数据流方程	$in[B] = \cup\ out[pred(B)]$ $out[B] = f_B(in[B])$	$out[B] = \cup\ in[succ(B)]$ $in[B] = f_B(out[B])$	$in[B] = \cap\ out[pred(B)]$ $out[B] = f_B(in[B])$
边界条件	$out[entry] = \varnothing$	$in[exit] = \varnothing$	$out[entry] = \varnothing$
初始值	$out[B] = \varnothing$	$in[B] = \varnothing$	out[B] = U

7.2.4 公用子表达式删除方法

首先进行可用表达式分析

对每个基本块B∈N，执行{

对每条instr∈B，如果instr形如z = x⊗y，且x⊗y在B的入口点可用{

① 判断x⊗y是否在这条指令instr处可用

② 获得到达z的x⊗y计算的定值指令w = x⊗y （instr本身除外）

③ 创建新的临时变量t

④ 将步骤②获得的定值指令w = x⊗y 替换为两条连续指令

t = x⊗y; w = t;

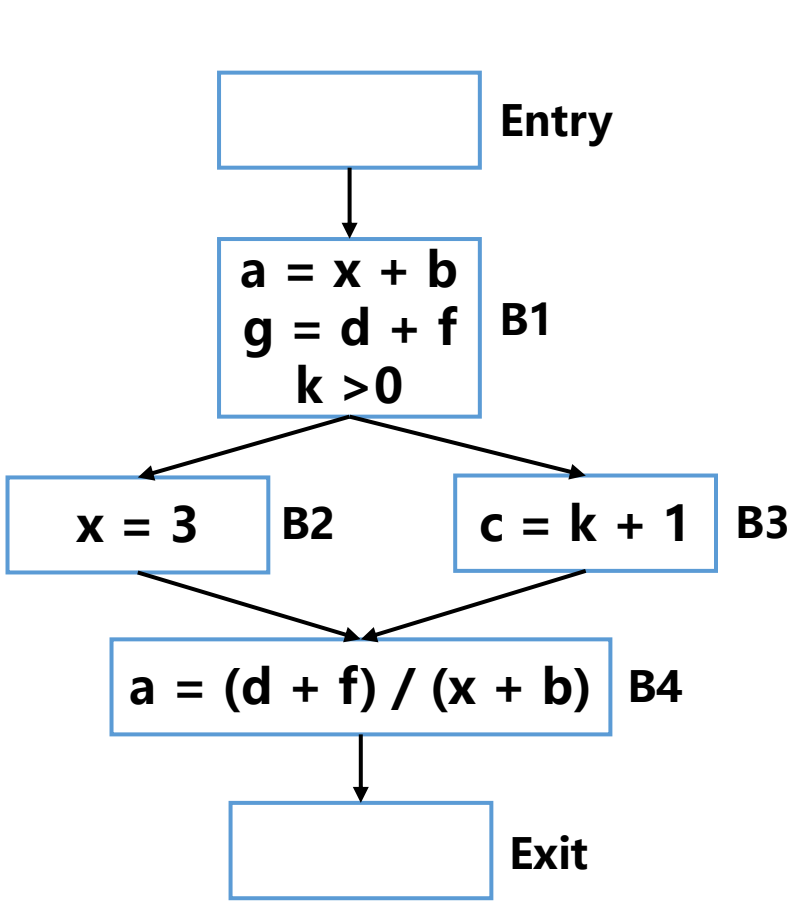
⑤将instr中的z = x⊗y替换为z = t

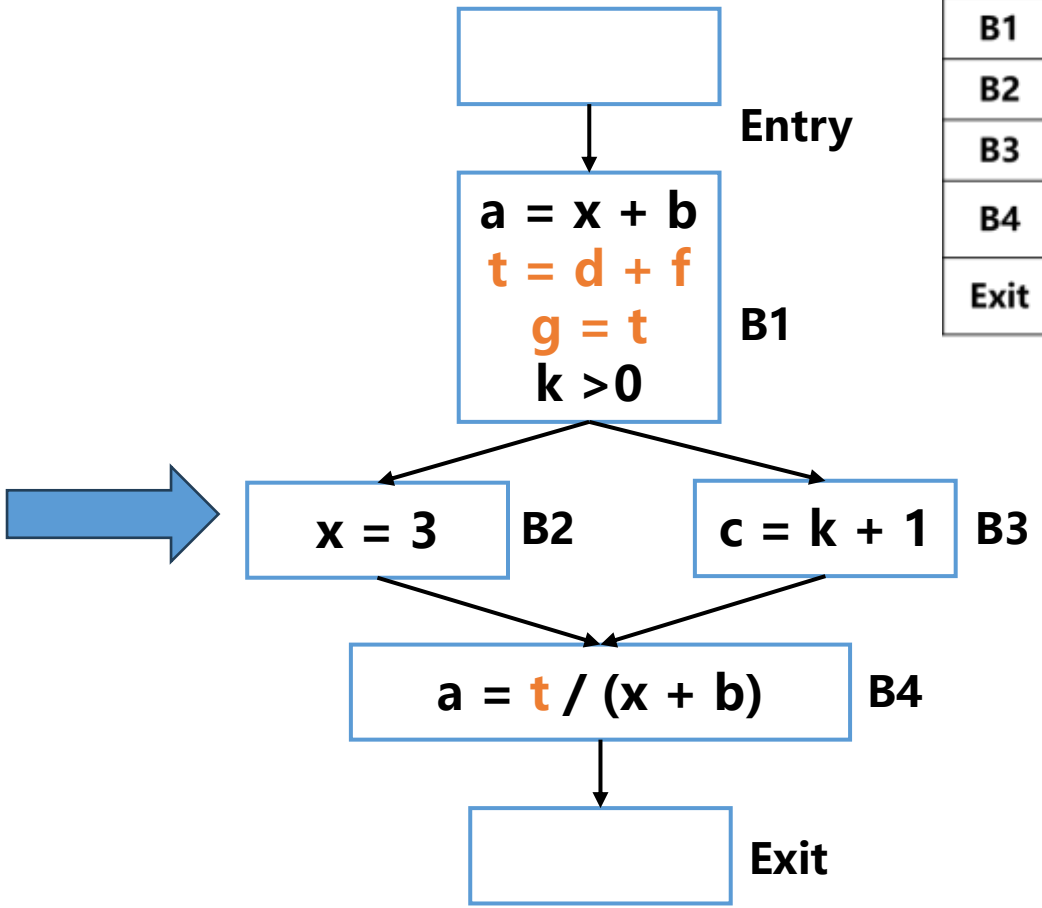
}

}

7.2.4 公用子表达式删除方法

练习：画出CSE优化后的控制流图



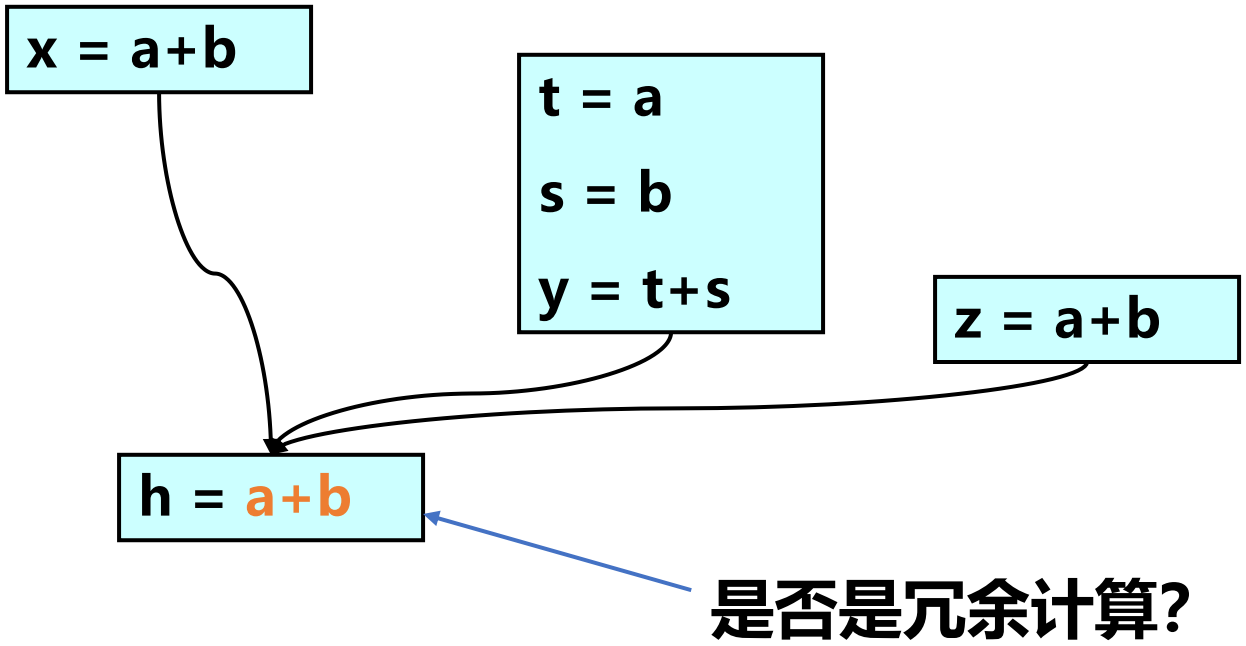


BB	in[B]²	out[B]²
Entry	∅	∅
B1	∅	{x+b, d+f}
B2	{x+b, d+f}	{d+f}
B3	{x+b, d+f}	{k+1, x+b, d+f}
B4	{d+f}	{d+f, x+b, (d+f)/(x+b)}
Exit	{d+f, x+b, (d+f)/(x+b)}	{d+f, x+b, (d+f)/(x+b)}

学习内容

- 7.1 常数传播
- 7.2 公用子表达式删除
- 7.3 值编号
- 7.4 循环不变量外提
- 7.5 死代码删除

7.3.1 值编号问题

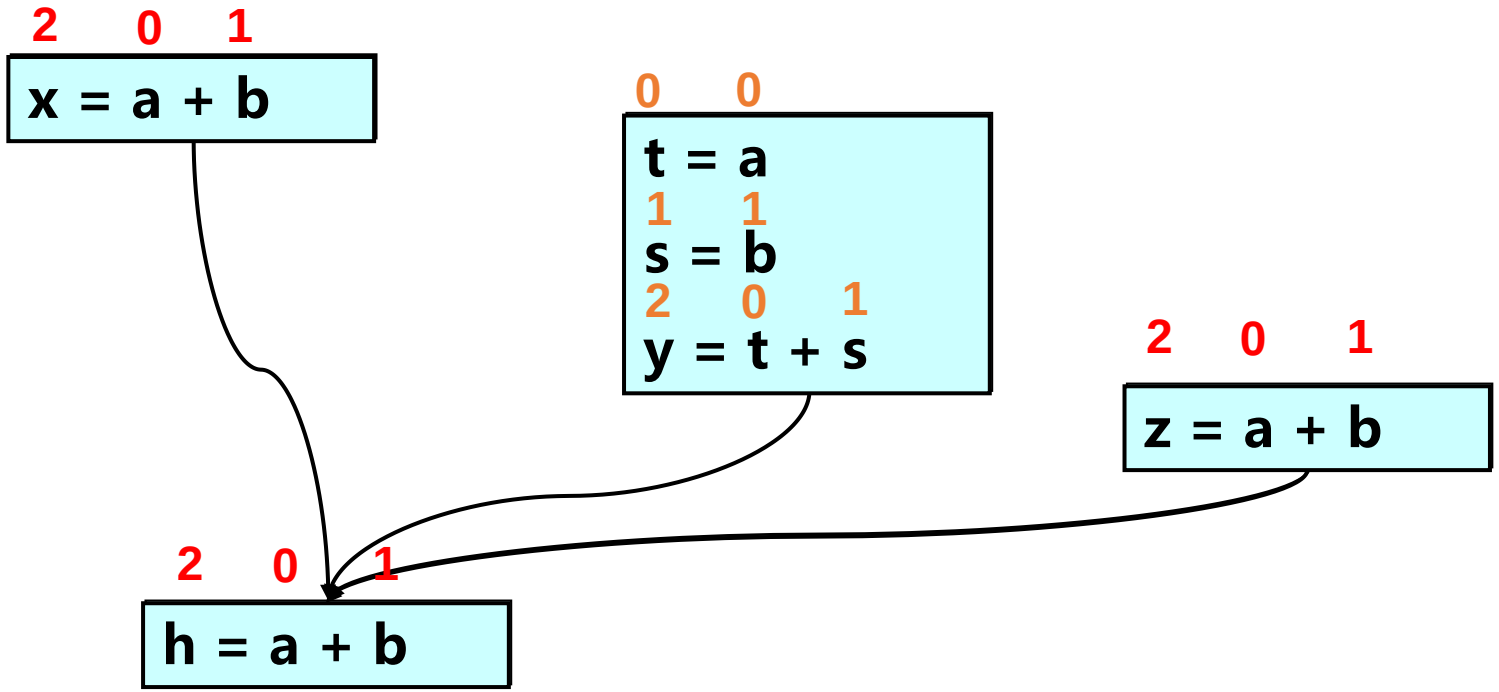


公用子表达式删除: NO! ☹

值编号(Value Numbering, VN): YES! ☺

7.3.1 值编号问题

- 每个表达式赋予唯一的值编号VN
  - 每个变量、表达式、常数
- 全等
  - 如VN(a)=VN(b)，则a全等于b
  - 如果 ⊕ 和 ⊗ 是相同的操作符，并且 x全等于a 且 y全等于b，则x ⊕ y 全等于 a ⊗ b （考虑：⊕满足交换律的情况）



7.3.2 计算值编号

- 编译器如何实现值编号？—— 基于hash值的值编号
  - a = 常数c

    VN(a) = c
  - 表达式x = y ⊗ z

    VN(x) = hash(⊗, VN(y), VN(z))
  - 根据 ⊗ 的可交换性选择不同的Hash函数

7.3.3 局部值编号方法

对基本块内的每条指令r：x =op1 ⊗ op2，执行 {

① 判断op1的值编号VN(op1)和op2的值编号VN(op2)是否已存在Hash表中

是，则获取VN(op1)和VN(op2)

否，则计算VN(op1) 和VN(op2)，并加入表中

② 计算key=VN(⊗, VN(op1), VN(op2))

③ 判断key是否已存在Hash表中

是，则将r右部的表达式计算替换为表中key对应的已计算好的指令左部

并令VN(x)=key

否，则将VN(x)=key加入表中

}

7.3.3 局部值编号方法

COMP CCRG Open MP

示例

```
a = i+1
b = 1+i
i = j+1
t1 = i+1
if(t1) goto L1
```

VN(Hash)	Name	Expr

7.3.3 局部值编号方法

示例

```
a = i+1
b = 1+i
i = j+1
t1 = i+1
if(t1) goto L1
```

VN(Hash)	Name	Expr
h1	i	
1	1	
h2	a	i+1

```
a = i+1
```

7.3.3 局部值编号方法

示例

```
a = i+1
b = 1+i
i = j+1
t1 = i+1
if(t1) goto L1
```

VN(Hash)	Name	Expr
h1	i	
1	1	
h2	a, b	i+1

```
a = i+1
b = a  //替换表达式1+i
```

7.3.3 局部值编号方法

示例

```
a = i+1
b = 1+i
i = j+1
t1 = i+1
if(t1) goto L1
```

VN(Hash)	Name	Expr
~~h1~~	~~i~~	
1	1	
h2	a, b	i+1
h3	j	
h4	i	j+1

```
a = i+1
b = a  //替换表达式1+i
i = j+1
```

7.3.3 局部值编号方法

示例

```
a = i+1
b = 1+i
i = j+1
t1 = i+1
if(t1) goto L1
```

VN(Hash)	Name	Expr
~~h1~~	~~i~~	
1	1	
h2	a, b	i+1
h3	j	
h4	i	j+1
h5	t1	i+1

h5 = hash(+,h4,1) ≠ h2

```
a = i+1
b = a  //替换表达式1+i
i = j+1
t1 = i+1 //非冗余计算
If(t1) goto L1
```

对比

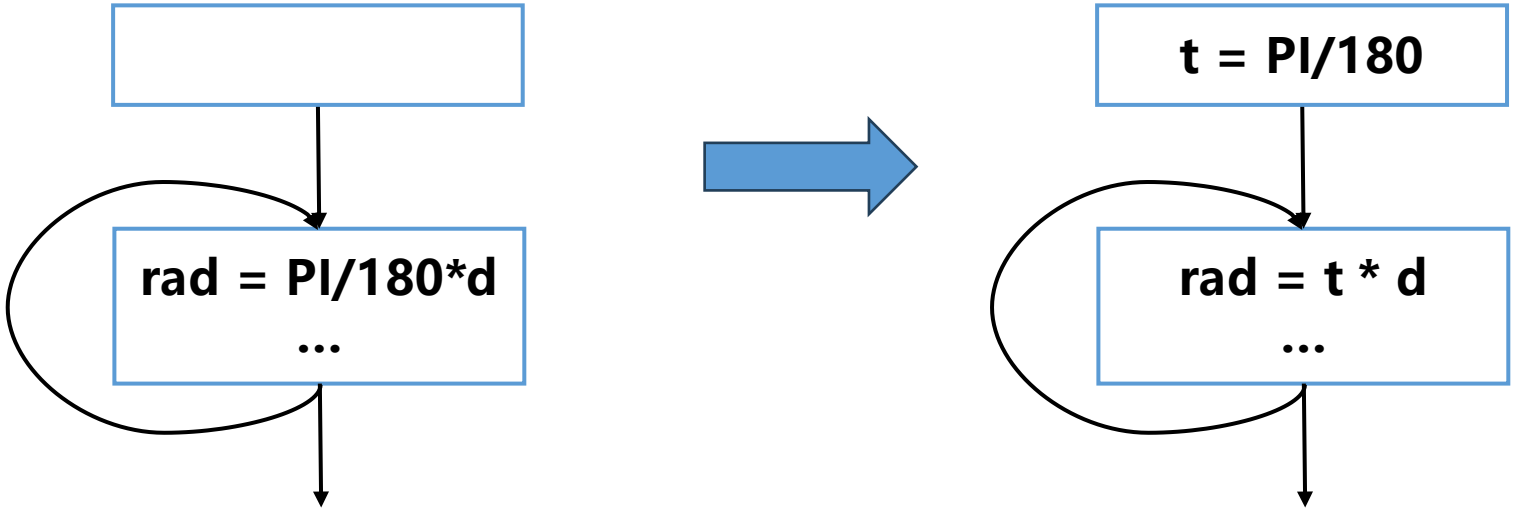
- CSE vs VN
- 冗余删除的基本方法
  - CSE：语法角度
  - VN：语义角度
- 实际解决方案：都实现！

学习内容

- 7.1 常数传播
- 7.2 公用子表达式删除
- 7.3 值编号
- 7.4 循环不变量外提
- 7.5 死代码删除

7.4.1 循环不变量

- 如果循环中的一个计算在循环的每次迭代都产生相同的值，则该计算被称作**循环不变量**
- 循环不变量外提(Loop-Invariant Code Motion, LICM)：将循环不变量移动到循环之外，避免冗余计算，减少循环内执行的指令



7.4.1 循环不变量

- 对于循环内的基本块的一条指令，如果它的每一个操作数都满足下列条件，则该指令是**循环不变**的：
  - 操作数是常数，或者
  - 所有到达操作数的定值都在循环之外，或者
  - 只存在一个到达操作数的定值，该定值在循环内，并且该定值本身是循环不变的

7.4.1 循环不变量

标记循环不变量

1. 首先标记两种简单的不变量

   ①标记所有操作数都是常数的指令为不变的

   ②标记所有操作数的到达定值都在循环外的指令为不变的

2. 重复下列步骤直到找出所有的循环不变量：

   标记操作数符合下列条件，并且没有被标记的指令，为不变的

   ①操作数是循环不变量，或者

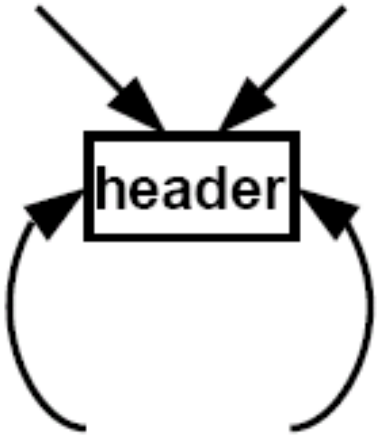
   ②只有一个定值到达操作数，并且该定值已经标记为循环的不变量

7.4.2 循环前置结点

思考

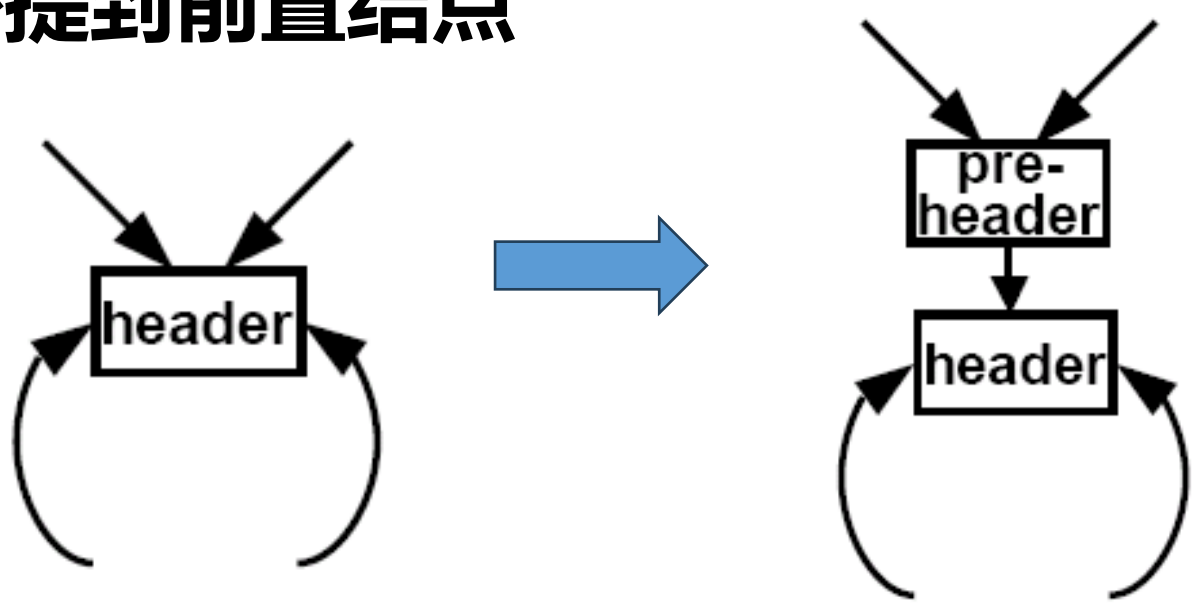
- 不变量外提后，放在哪里?

  循环头结点？循环头结点的前驱?



7.4.2 循环前置结点

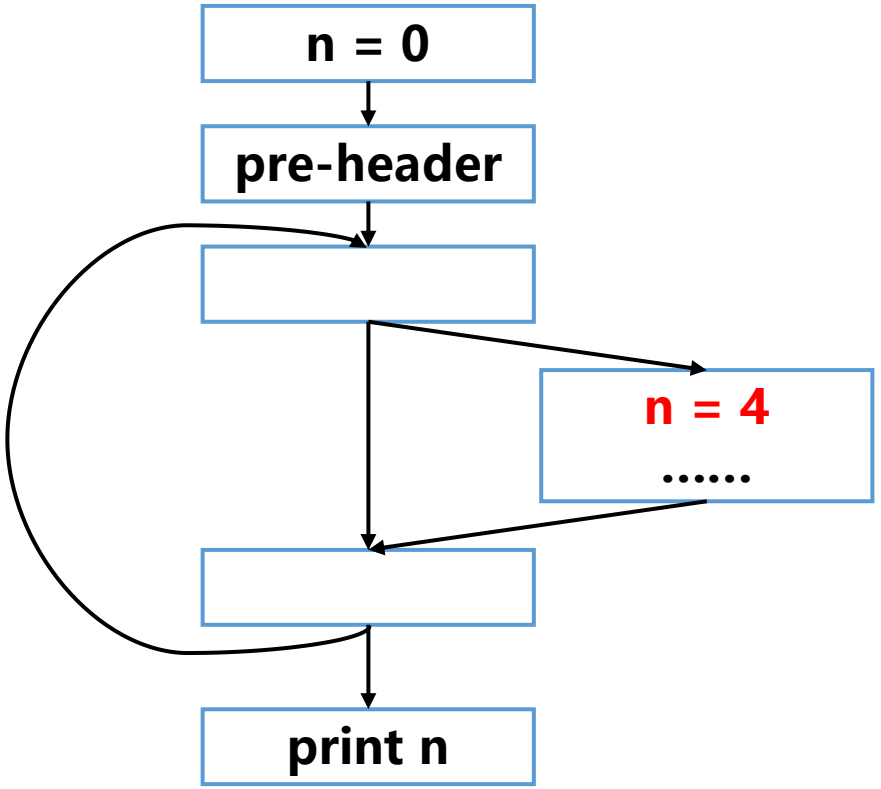
- 循环前置结点
  - 只有一个后继：循环头结点
  - 来自于循环之外、到循环头结点的边，改为到前置结点
  - 来自于循环内部，到循环头结点的边，保持不变
- 循环不变量外提到前置结点

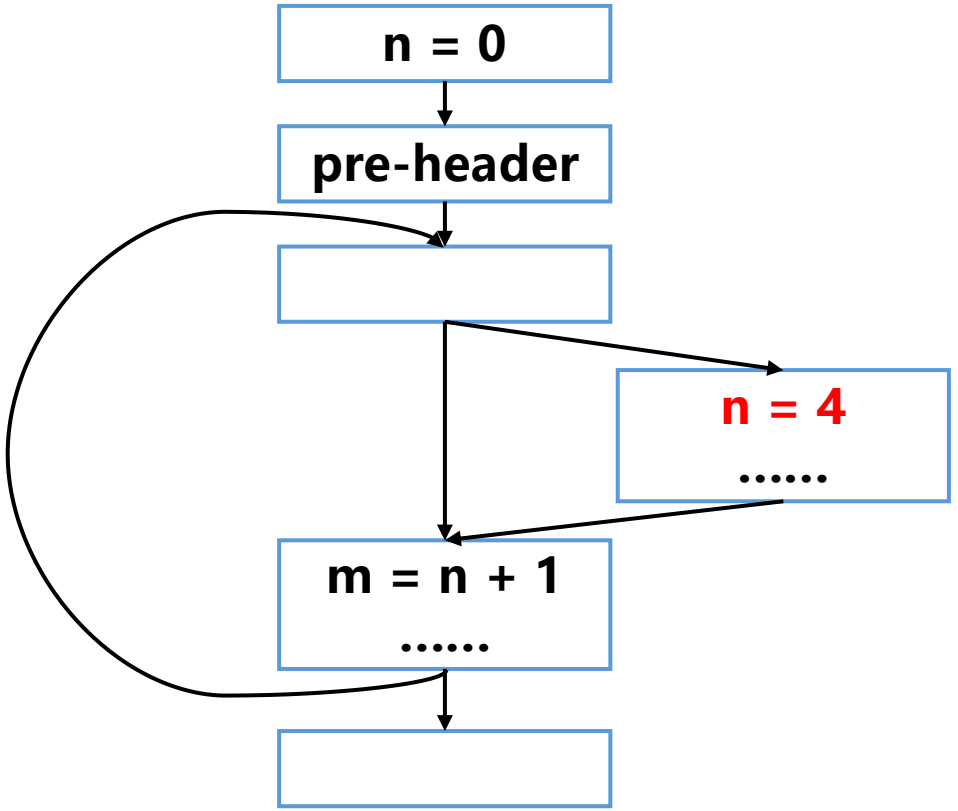


7.4.3 循环不变量外提条件

思考

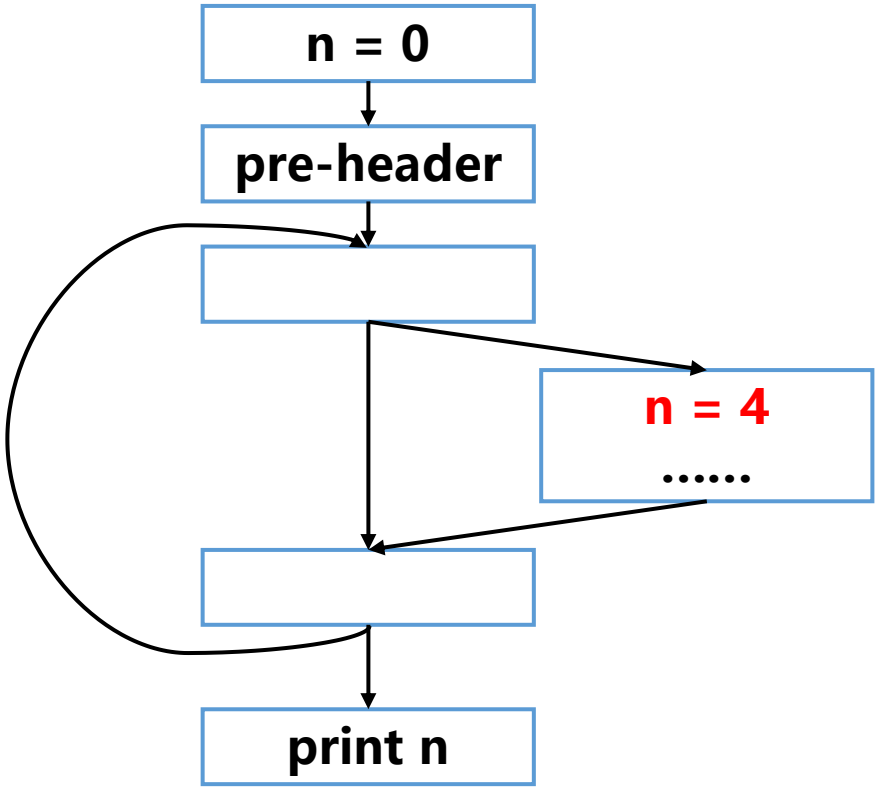
- 所有不变量都可以外提么?





7.4.3 循环不变量外提条件

不变量外提条件一



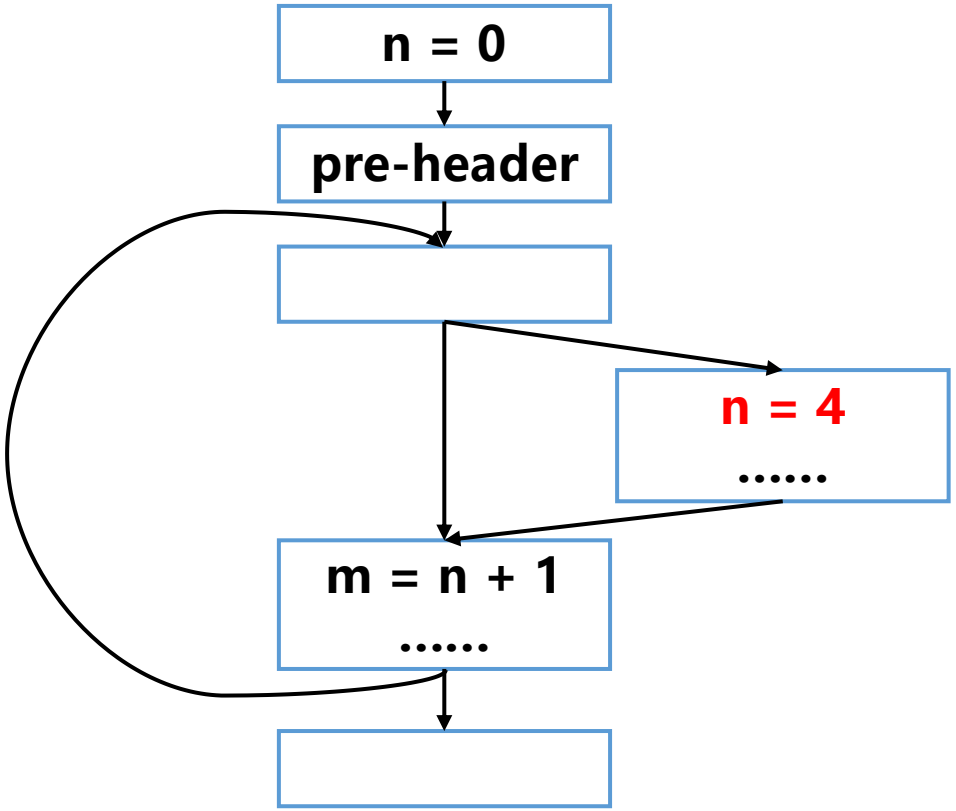
假设不变量s： v=x⊗y 或 v=x

对v进行定值，使用x, y

条件一: 语句s是循环L所有出口的必经结点

7.4.3 循环不变量外提条件

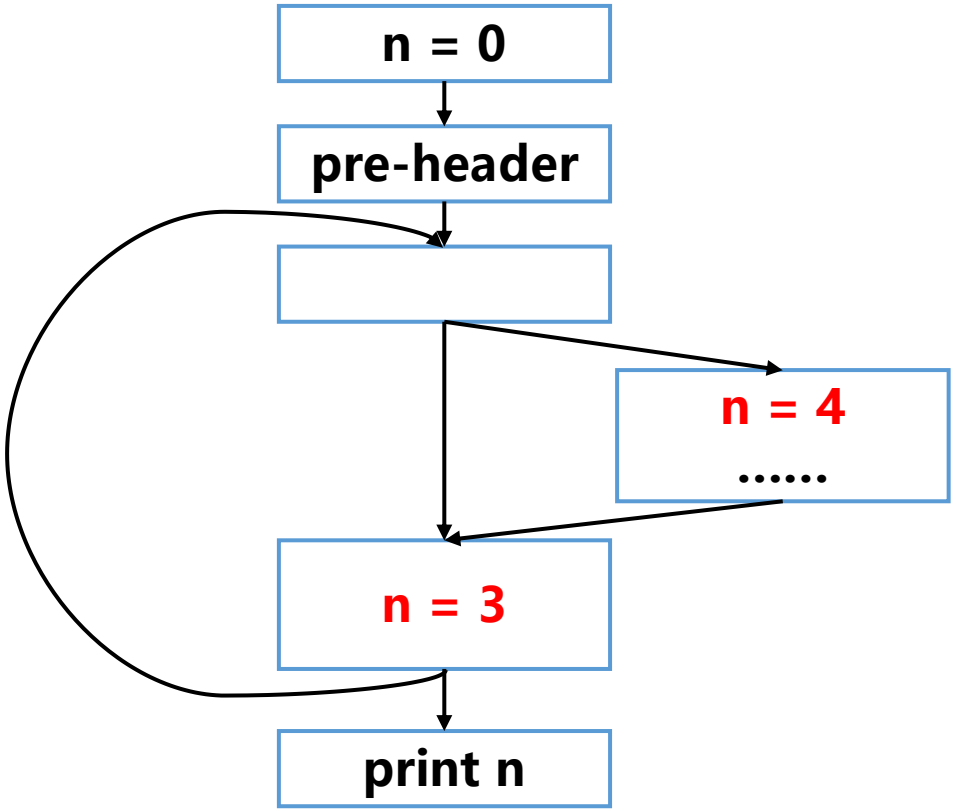
不变量外提条件二



条件二: 循环L中对v的所有使用只能被s中的定值到达

7.4.3 循环不变量外提条件

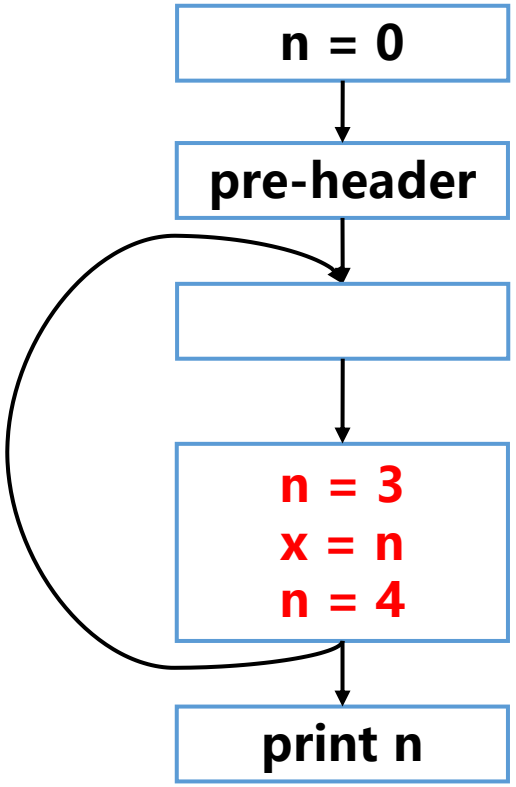
不变量外提条件三



条件三: v不能在循环L中有其他定值

7.4.3 循环不变量外提条件

不变量外提条件四



条件四：不变量使用的操作数x、y的定值都来自于循环L之外(可以是一开始就在循环外，或者外提后在循环外)

7.4.4 循环不变量外提方法

执行到达定值分析，并构建UD链

找出循环L中的循环不变量

对每个循环不变量指令s，假设s定值变量v，使用变量x和y，执行{

如果 (1. s所在基本块是L的所有出口的必经结点&&

2. 在循环L中v没有其它定值&&

3. L中v的所有使用只被s中的定值到达&&

4. x和y的定值在循环外(可以是一开始就在循环外，或者外提后在循环外))

{

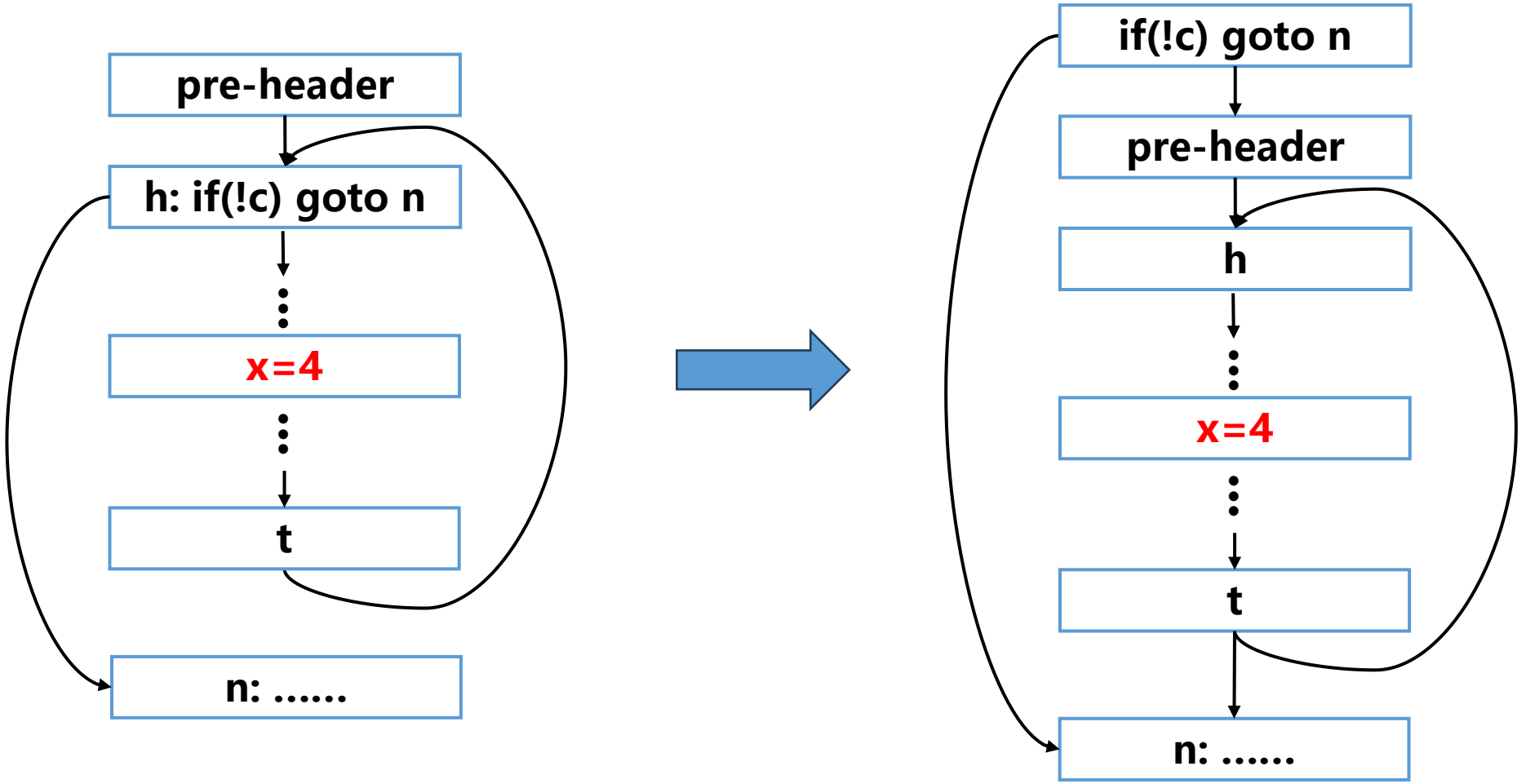
将s外提，移到L的前置结点

}

}

7.4.4 循环不变量外提方法

- 如果循环迭代次数为0，怎么办?
- 通过先测试循环是否执行来避免零迭代循环带来的问题



思考

- 循环不变量外提算法还可以如何优化?
  - 条件1、2是否可以放松?
- 分析LLVM的循环不变量外提算法
  - llvm/lib/Transforms/Scalar/LICM.cpp
  - 和讲授算法相比，做了什么优化?

思考


```
#include <math.h>
void loop(double *a, const double b, int m)
{
    int i;
    while (i++ < m)
        a[i] = sin(b);   //函数调用
    return;
}
```

学习内容

- 7.1 常数传播
- 7.2 公用子表达式删除
- 7.3 值编号
- 7.4 循环不变量外提
- 7.5 死代码删除

7.5.1 死代码删除

- 死变量
  - 如果一个变量v在从点p开始的某条路径上被使用，那么变量v在点p是活跃的
  - 否则(从点p开始的任何路径上都未被使用)，变量v在点p是死变量

- 死代码删除(Dead Code Elimination, DCE): 试图删除在基本块中被定值但是在基本块出口不活跃的死变量

7.5.2 死代码删除方法

```
change = false;
do{
    执行活跃变量分析;
    遍历所有基本块，对基本块B中每一条定值变量v的语句s {
        如果定值点的活跃变量集合out(s)不包含v {
            删除s;
            change = true;
        }
    }
}while(changed)
```

7.5.3 死代码删除示例

COMP CCRG Open MP

- C代码dce.c

```
int foo(int x, int y) {
    int a = x+y;
    a = 1;
    return a;
}
```

无dce优化（opt dce.ll –S –mem2reg）

```
define dso_local i32 @foo(i32 %0, i32 %1) #0 {
  %3 = add nsw i32 %0, %1
  ret i32 1
}
```

dce优化（opt dce.ll –S –mem2reg -dce）

```
define dso_local i32 @foo(i32 %0, i32 %1) #0 {
  ret i32 1
}
```

7.5.3 死代码删除示例

- C代码dce2.c

```
int b;  //global variable
int foo(int x, int y) {
    int a = x+y;
    b = a;
    return x;
}
```

dce优化（opt dce2.ll –S –mem2reg -dce)

```
define dso_local i32 @foo(i32 %0, i32 %1) #0 {
  %3 = add nsw i32 %0, %1
  store i32 %3, i32* @b, align 4
  ret i32 %0
}
```

DCE优化不能删除对全局变量b的写，因此不能删除对a的赋值

7.5.3 死代码删除示例

- 下面的C代码dce3.c?

```
int b;  //global variable
int foo(int x, int y) {
    volatile int a = x+y;
    b = a;
    return x;
}
```

dce优化（opt dce3.ll –S –mem2reg -dce)

```
define dso_local i32 @foo(i32 %0, i32 %1) #0 {
  %3 = alloca i32, align 4
  %4 = add nsw i32 %0, %1
  store volatile i32 %4, i32* %3, align 4
  %5 = load volatile i32, i32* %3, align 4
  store i32 %5, i32* @b, align 4
  ret i32 %0
}
```

DCE优化通常不删除对volatile 变量的读写

7.5.4 激进的死代码删除

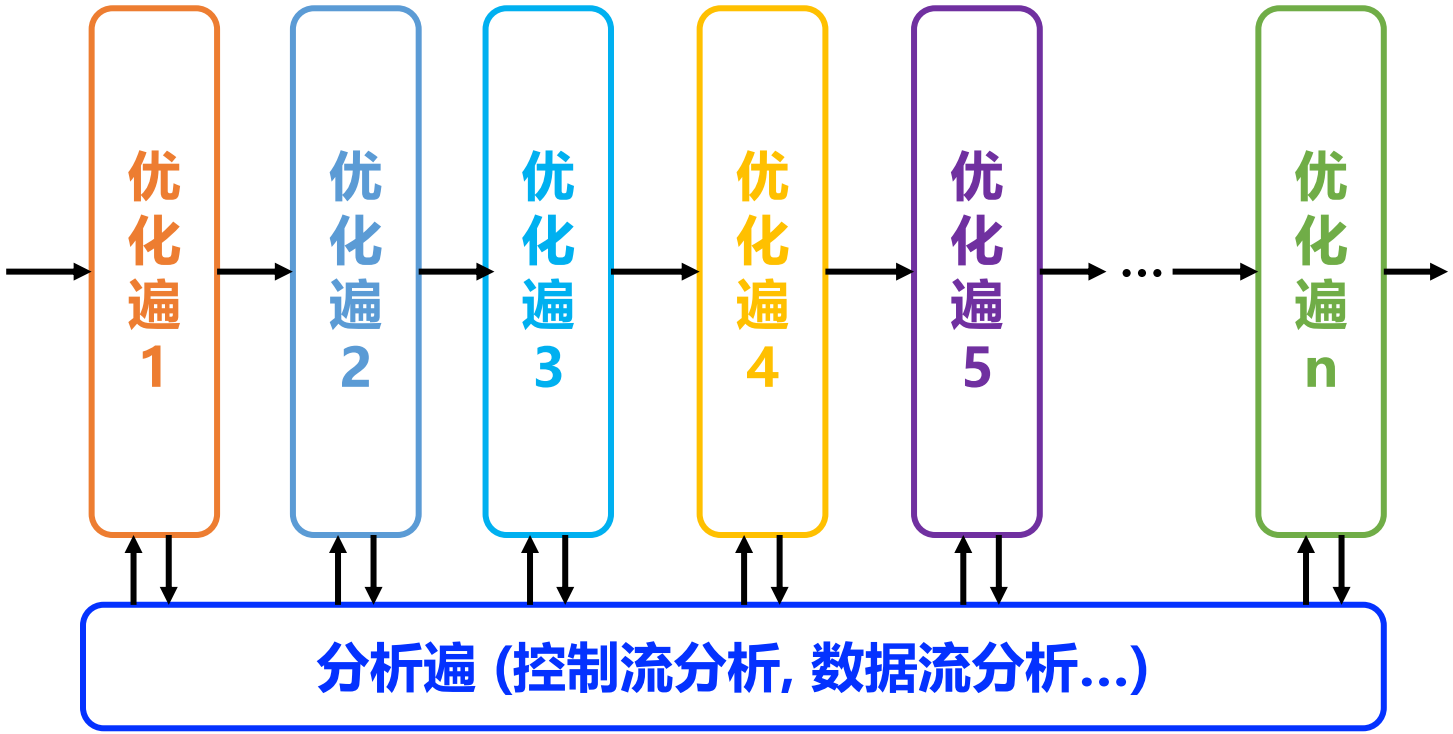
- Aggressive Dead-Code Elimination, ADCE
- 思想: 假设所有代码都是“死”代码，除非有证据表明该代码对程序最终结果有贡献：
  - 执行IO，写存储、从函数返回，或者调用具有副作用的函数
  - 对其他活跃代码中使用的变量赋值
  - 是一条条件分支语句，并且有其他活跃代码控制依赖于它
- 方法: 遍历所有代码，删除所有未标记为“活跃”的代码

思考

- 阅读代码
  - llvm/lib/Transforms/Scalar/ADCE.cpp

本讲小结

- 常数传播
- 公用子表达式删除
- 值编号
- 循环不变量外提
- 死代码删除



作业

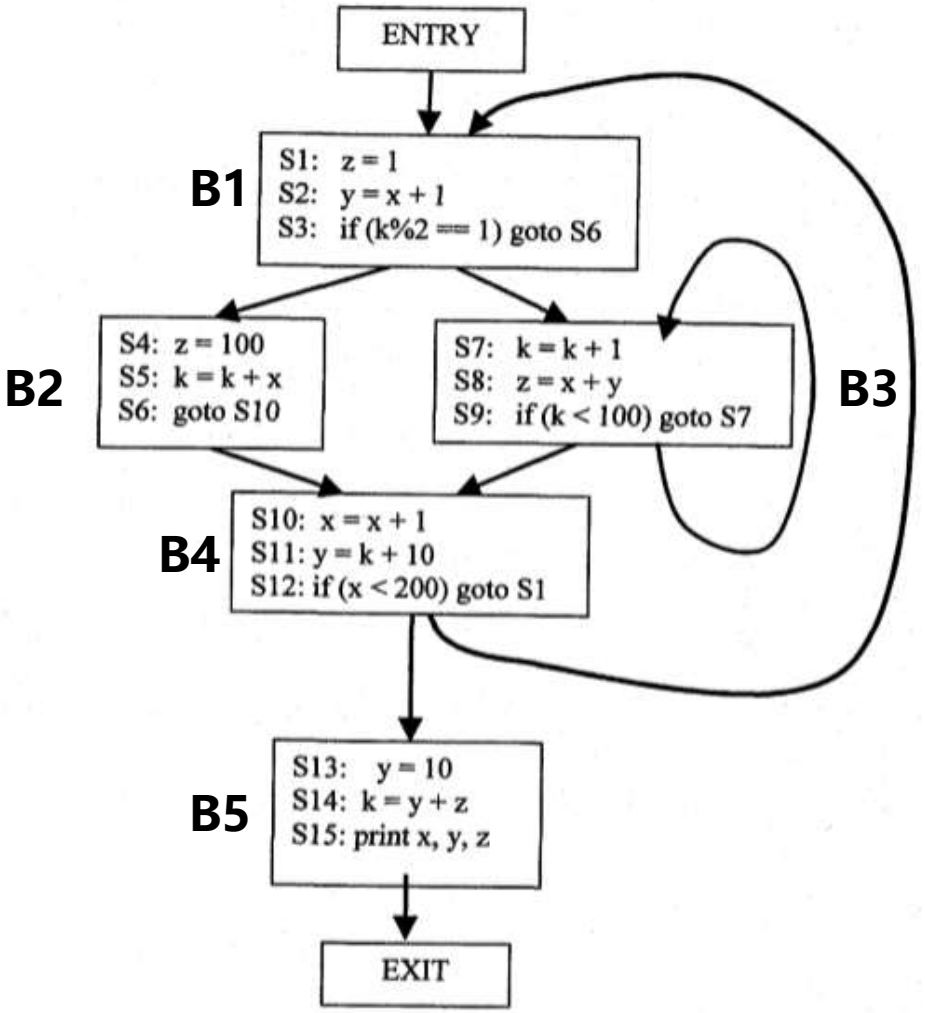
- 对如下控制流图:

(1)进行活跃变量分析

给出各基本块边界处(入口处和出口处)的活跃变量情况，给出具体分析过程和最终结果

(2)画出DCE优化后的CFG

给出具体分析过程和最终结果



参考资料

- 《编译原理》(龙书) 第9章
- 《高级编译器设计与实现》(鲸书) 第8章
- LLVM源码