代码优化

代码优化：概览

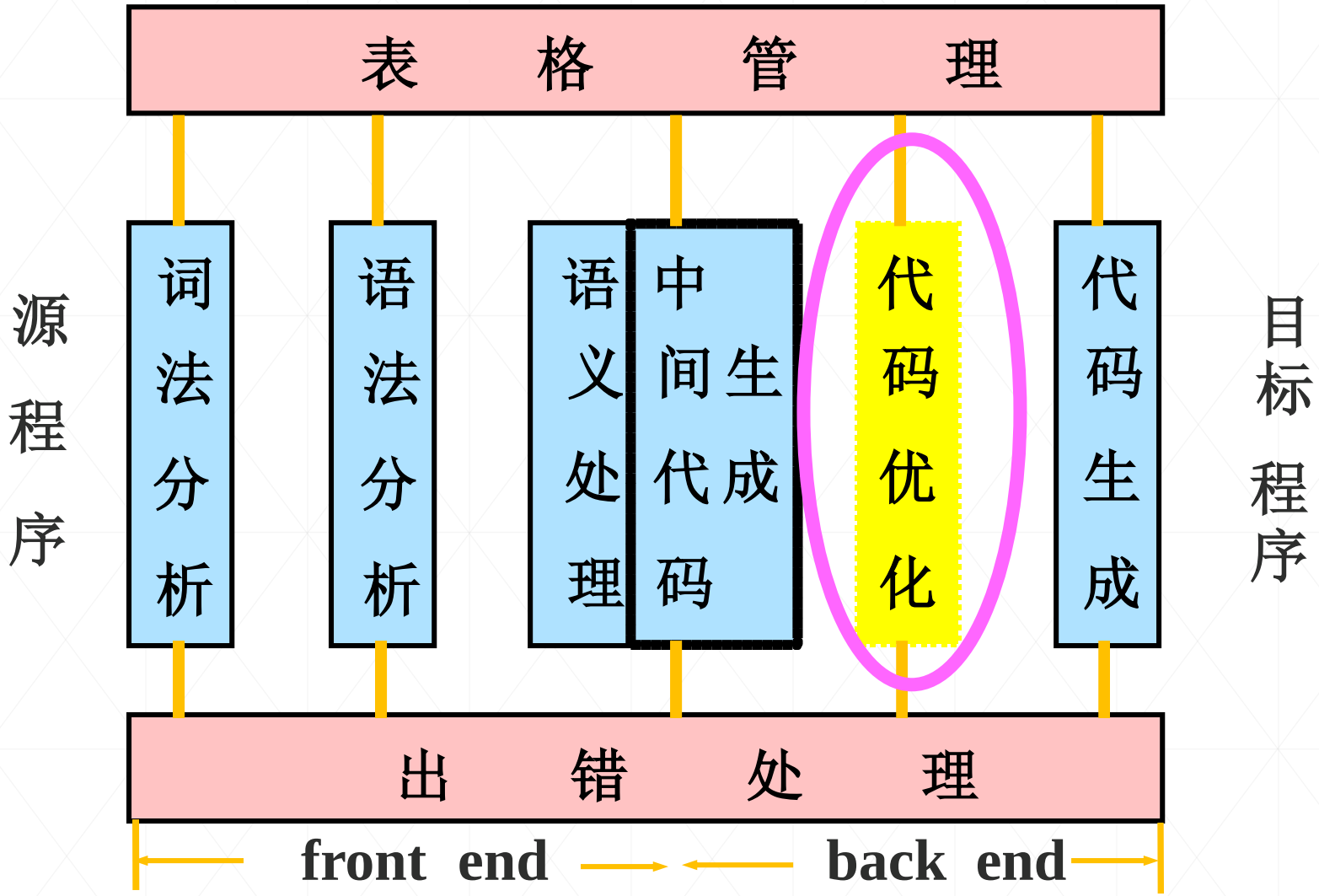
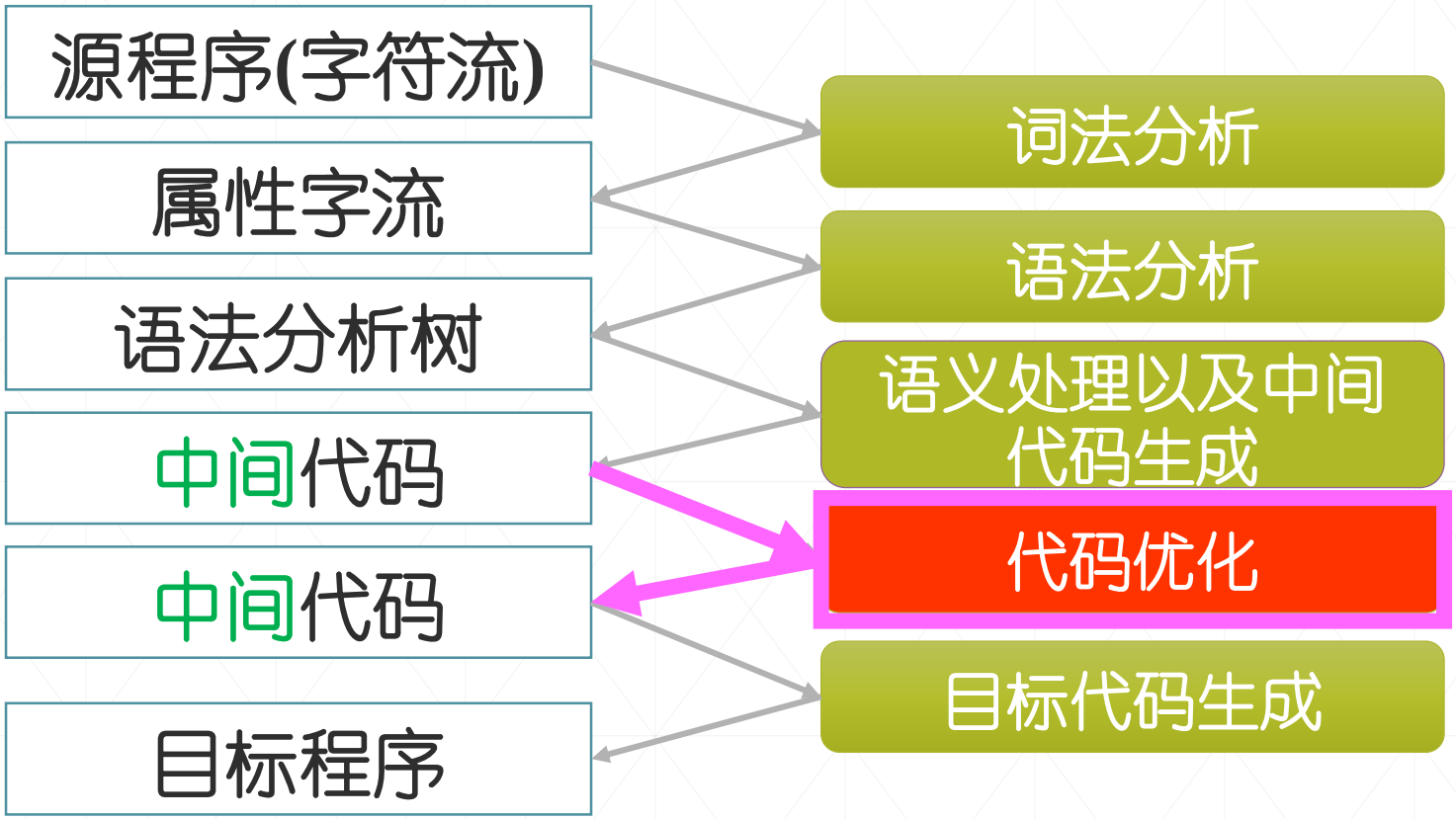


表 格 管 理
源程序
词法分析
语法分析
语义处理
中间代码生成
代码优化
代码生成
目标程序
出 错 处 理
front end
back end

- 基本功能


源程序(字符流)
属性字流
语法分析树
中间代码
中间代码
目标程序
词法分析
语法分析
语义处理以及中间代码生成
代码优化
目标代码生成

第 8 章 代码优化（optimization）

8.1 代码优化概述

8.2 局部优化

8.3 控制流分析与循环查找

8.4 数据流分析基础

8.5 循环优化的实施

8.1 代码优化概述

8.1.1 代码优化的概念

8.1.2 优化技术分类

8.1.3 具优化功能编译器的组织

- 代码优化

在不改变程序运行效果的前提下，对被编译的程序进行等价变换，使之能生成更加高效的目标代码。

- 优化整体过程
  - 等价：不改变程序执行效果；
  - 变换：引起程序形式上的变动

提高程序效率的途径

1） 改进算法；

2） 在源程序级上等价变换；

3） 充分利用系统提供的程序库；

4） 编译时优化等。

放心 我懂

优化目的

产生高效的目标代码。

- 时间：源程序运行时间尽可能短；
- 空间：程序及数据所占空间尽可能少；

为什么要实施优化

- 优化程度是编译器的一个重要技术、质量目标；
- 无法苛求用户对源语言的掌握，编程技巧，编写源程序的优化；
- 编译程序固有的缺陷：不是面对一个或一类具体问题的程序，而是统一处理该语言的各种源程序，无法尽善尽美。

例如，

```
int  a[25][25] , b[25][25];
  …
a[i][j] = b[i][j];
  …
```

对a[i][j] = b[i][j]翻译的目标代码：

计算a[i][j] 的addr
计算b[i][j] 的addr
赋 值

重复计算了地址的变化部分

8.1 代码优化概述

8.1.1 代码优化的概念

8.1.2 优化技术分类

8.1.3 具优化功能编译器的组织

一. 优化所涉及的源程序的范围

- 局部优化 — 基本块内优化；
- 循环优化 — 隐式、显式循环体内优化；
- 全局优化 — 一个源程序范围内优化；

二. 优化相对于编译逻辑功能实现的阶段

- 中间代码级 — 目标代码生成前的优化；
- 目标代码级 — 目标代码生成后的优化。

三. 优化具体实现技术的角度

1.常量合并

Before optimization

```
X = 2;
Y = X + 10;
Z = 2 * Y;
```

After optimization

```
X = 2;
Y = 12;
Z = 24;
```

2．公共子表达式删除

Before optimization

```
d = e + f + g;
y = e + f + z;
```

After optimization

```
x = e + f ;
d = x + g;
y = x + z;
```

3. 循环不变量或不变代码外提

Before optimization

```
b = c;
for(i=0; i<3; i++)
    d[i] = 2 * b + 1;
```

After optimization

```
b = c;
z= 2 * b + 1;
for(i=0; i<3;i++)
    d[i] = z ;
```

下一例

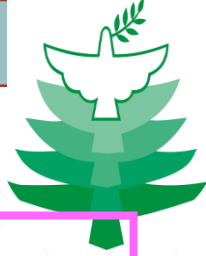
4. 无用赋值删除

Before optimization

```
a = 5;
…
a = 7;
```

After optimization

```
a = 7;
```



6. 死代码删除

Before optimization

```
char c;
if  (c > 300) a = 1;
else
    a = 2 ;
```

After optimization

```
a = 2;
```

永假式

7. 循环内的运算强度削弱

C source code

```
int table[100];
step = 1;
for(i=0; i<100; i+=step)
table[i] = 0;
```

before optimization

```
    i = 0;
L1: if(i<100) goto L2
    if(i>100) goto L3
L4: i++; goto L1;
L2: t1 = i * 4;
    table[t1] = 0; goto L4;
L3:
```

Loop

after optimization

```
    i = 0;
    t1 = i * 4;
L1:if(i<100) goto L2
    if(i>100) goto L3
L4: i++; goto L1;
L2: t1 = t1 + 4;
    table[t1] = 0 ; goto L4;
L3:
```

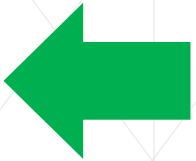
返回

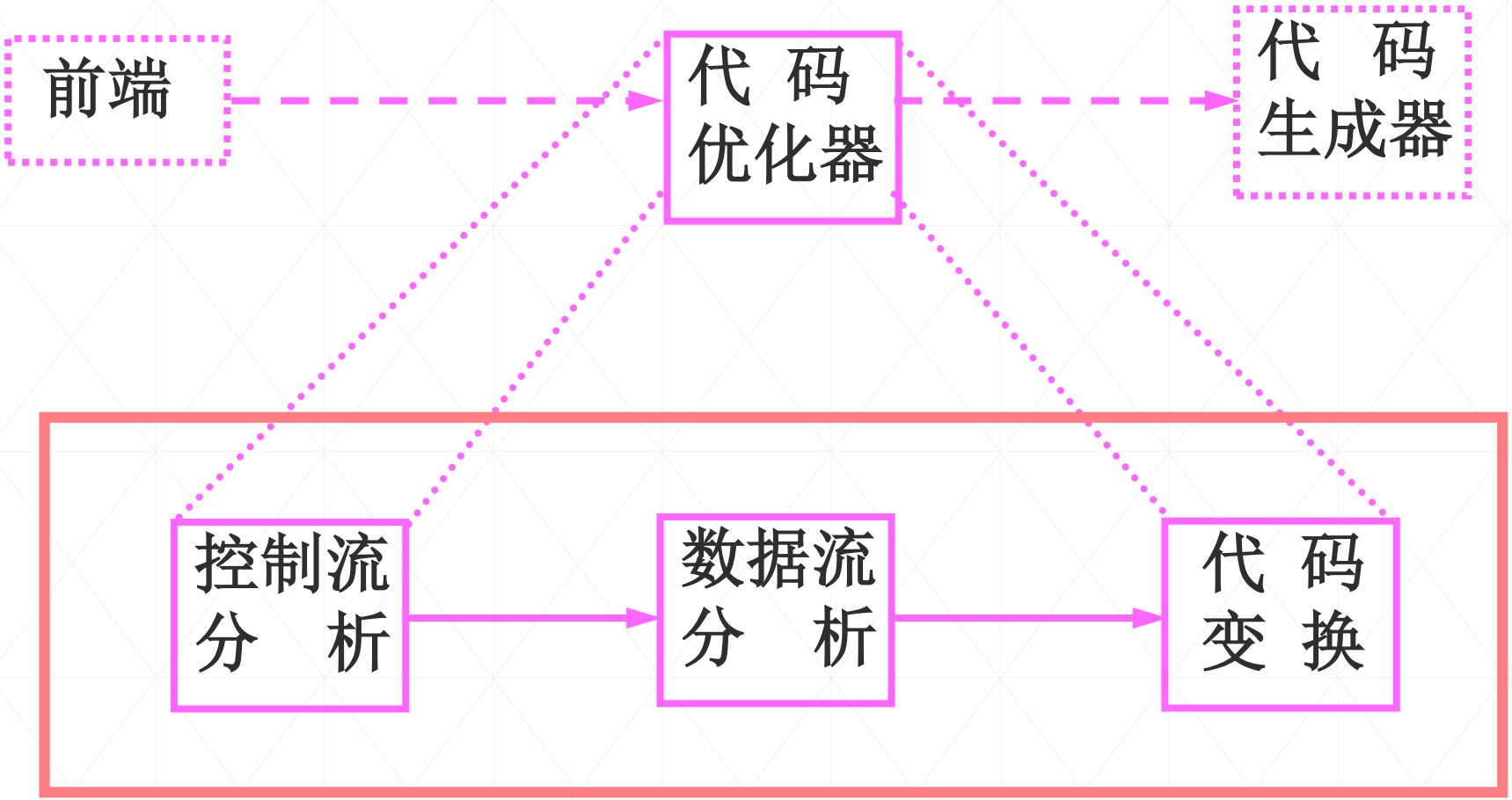
8.1 代码优化概述

8.1.1 代码优化的概念

8.1.2 优化技术分类

8.1.3 具优化功能编译器的组织




前端
代 码
优化器
代 码
生成器
控制流
分 析
数据流
分 析
代 码
变 换

第 8 章 代码优化（optimization）

8.2 局部优化

8.2.1 基本块定义与划分

8.2.2 程序的控制流图

8.2.3 基本块的DAG表示与应用

- **局部优化**

  指在程序的一个**基本块**内进行的优化。

- **基本块**

一顺序执行的最大语句序列。

特点1：

惟一入口和惟一出口。

序列的第一个语句

序列的最后一个语句

特点2：

没有转进转出，分叉汇合 。

基本块划分

第1步：确定每个基本块的入口语句。

据基本块结构特点，入口语句是下述语句之一：

(1)程序的第一个语句；

(2)分叉汇合的语句：

(a)转移语句转移到的语句； 汇合语句

(b)紧跟转移语句后面的语句。 分叉语句

引起分叉汇合的语句：转移语句

第2步：根据基本块的入口语句，构造基本块。

一个入口语句对应一个基本块，每个基本块的范围：

⑴该入口语句到下一个入口语句之间有停止、暂停语句或该入口语句是程序的最后一个入口语句，该入口语句到此停止、暂停语句或程序的最后一个语句。

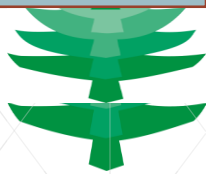
⑵否则：该入口语句到下一个入口语句(不包含下一个入口语句)；

第3步：凡是未包含在基本块中的语句，都是程序的控制流不可到达的语句，直接从程序中删除。

死代码删除

例 对如下程序段实施基本块的划分。

```
(1)   read ( limit );          0
(2)   i=1;
(3)   if  (i>limit)  goto (11);  1
(4)   read(j)                  2
(5)   if  (i=1) goto (8);
(6)   sum=sum+j;               3
(7)   goto (9);
(8)   sum=j;                   4
(9)   i + +;                   5
(10)  goto (3);
(11)  write(sum);              6
```



基本块的确定

Step1:找各基本块的入口语句；

Step2:由每一个入口语句构造相应的基本块；

Step3:凡不属于某一基本块中的语句，皆是程序控制流程无法到达的语句，直接删除；

例：逆向分析是安全领域对可执行程序进行分析的常用技术手段之一，如下为一段MIPS汇编程序及其对应指令说明：

(1)请绘制该程序的程序控制流图；

(2)该程序中是否存在循环，包括哪些基本块？

(3)请说明该程序的功能。

```
        la      $t0, array      #加载数组起始地址到t0
        addi    $t3,$zero, 0    #计数器$t3 = 0
out:    add     $t1, 0, $t0     #$t1=0+$t0
        slti    $s0, $t3, 10    #比较$t3是否小于10
        beqz    $s0, print      #不小于则跳转到print
        addi    $t4, $t3, -1    #$t4 = $t3 - 1
in:     slti    $s0, $t4, 0     #比较$t4 是否小于0
        bnez    $s0, exitIn     #小于则跳转到exitIn
        sll     $t5, $t4, 2     #$t5 = $t4 * 4
        add     $t5, $t1, $t5   #$t5 = 数组起始地址 + $t4 * 4
        lw      $t6, 0($t5)     #加载数组元素到$t6
        lw      $t7, 4($t5)     #加载数组元素到$t7
        slt     $s0, $t6, $t7   #比较$t6是否小于$t7
        bnez    $s0, swap       #小于则跳转到swap
        j       exitIn          #跳转到exitIn
swap:   sw      $t6, 4($t5)     #写入数组元素
        sw      $t7, 0($t5)     #写入数组元素
        addi    $t4, $t4, -1    #$t4 = $t4 - 1
        j       in              #跳转到in
exitIn: addi    $t3, $t3, 1     #$t3 = $t3 + 1
        j       out             #跳转到out
print:
```

0 1 2 3 4 5 6 7 8

8.2 局部优化

8.2.1 基本块定义与划分

8.2.2 程序的控制流图

8.2.3 基本块的DAG表示与应用

程序的控制流图(简称流图)

具有惟一首结点的有向图。流图G表示为

$$G = (N, E, n_0)$$

其中：

N： 流图的所有结点组成的集合。

流图中的一个结点为一个基本块。

n_0： 流图的首结点。

程序的首条语句所在的基本块。

程序的第一个基本块。

E： 流图的所有有向边组成的集合。

- 流图中的有向边：

如：结点i到结点k有边 

基本块i执行完后接下来可能执行的基本块为k

条件：

① 基本块k在代码中的位置紧跟在基本块i之后且i的出口语句不是无条件转移或停语句；

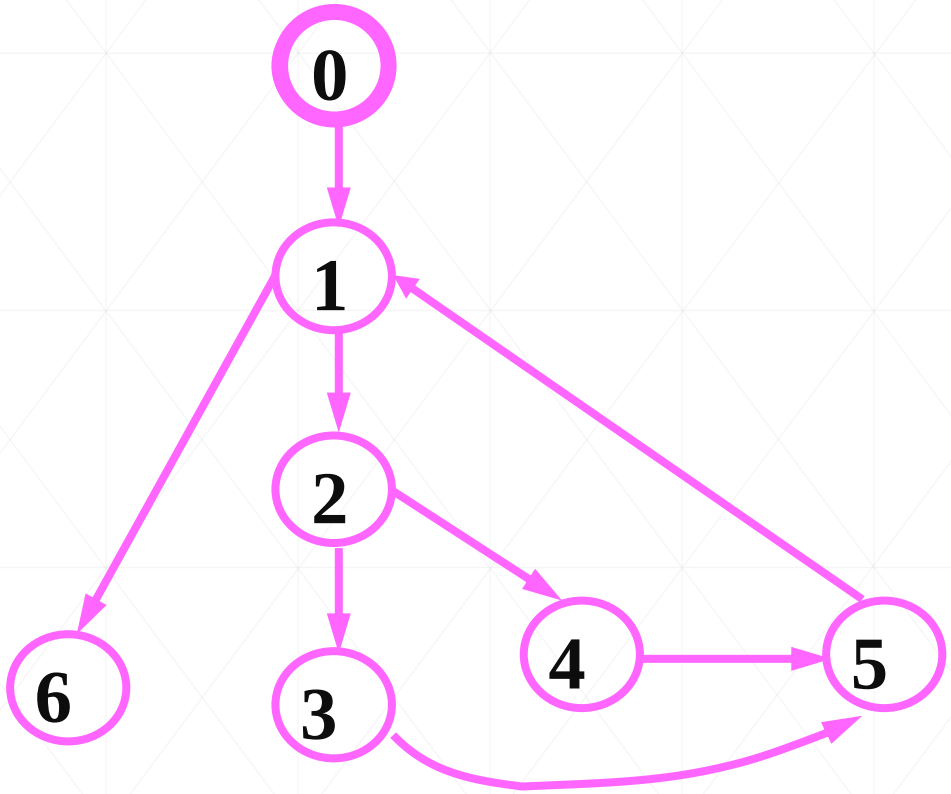
或② 基本块i的出口语句是跳转语句且跳转到的语句是基本块k的入口语句。

每个基本块射出的有向边数可为0或1或2：
根据其出口语句确定

例 给出如下程序的流图。

```
(1) read ( limit );
(2) i=1;
(3) if  (i>limit)  goto (11);
(4) read(j)
(5) if  (i=1) goto (8);
(6) sum=sum+j;
(7) goto (9);
(8) sum=j;
(9) i + +;
(10) goto (3);
(11) write(sum);
```

0 1 2 3 4 5 6



例 对如下程序段划分基本块，给出流图。

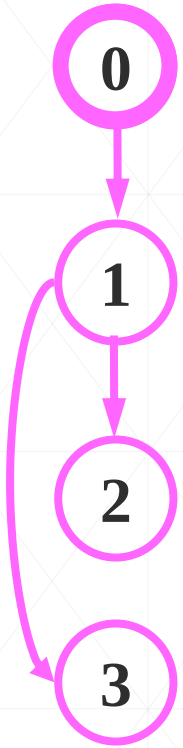
```
① read  x
② read  y
③ R=x / y
④  if R=0  goto  9
⑤  x = y
⑥  y = R
⑦  stop
⑧  goto 3
⑨ write y
⑩ halt
```

0

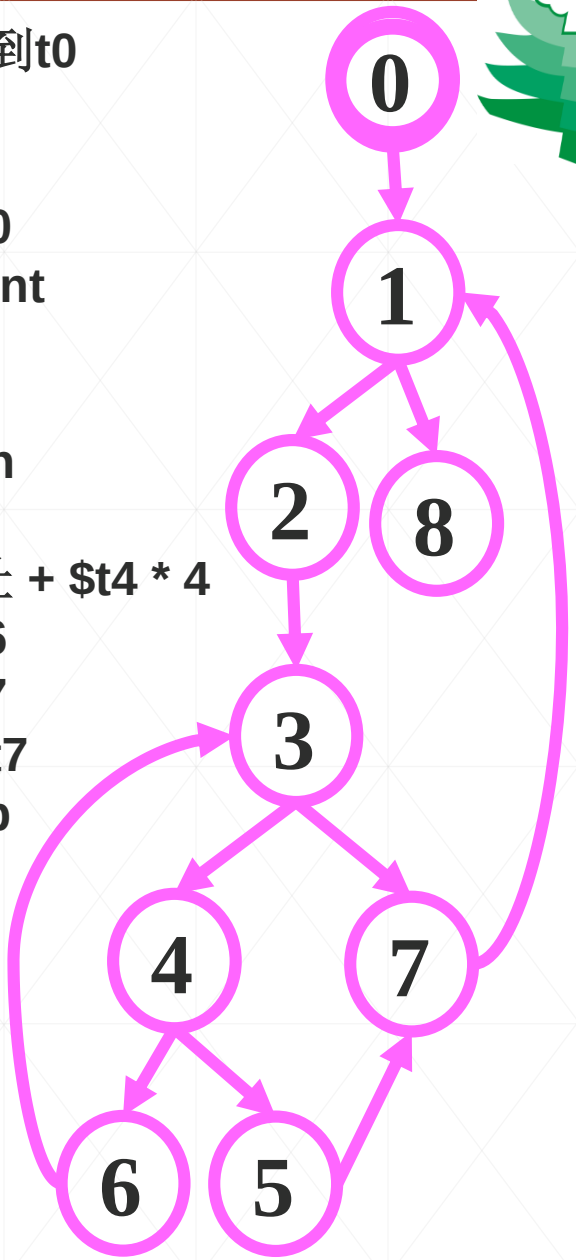
1

2

3



```
        la      $t0, array      0   #加载数组起始地址到t0
        addi    $t3,$zero, 0        #计数器$t3 = 0
out:    add     $t1, 0, $t0     1   #$t1=0+$t0
        slti    $s0, $t3, 10        #比较$t3是否小于10
        beqz    $s0, print          #不小于则跳转到print
        addi    $t4, $t3, -1    2   #$t4 = $t3 - 1
in:     slti    $s0, $t4, 0     3   #比较$t4 是否小于0
        bnez    $s0, exitIn         #小于则跳转到exitIn
        sll     $t5, $t4, 2     4   #$t5 = $t4 * 4
        add     $t5, $t1, $t5       #$t5 = 数组起始地址 + $t4 * 4
        lw      $t6, 0($t5)         #加载数组元素到$t6
        lw      $t7, 4($t5)         #加载数组元素到$t7
        slt     $s0, $t6, $t7       #比较$t6是否小于$t7
        bnez    $s0, swap           #小于则跳转到swap
        j       exitIn          5   #跳转到exitIn
swap:   sw      $t6, 4($t5)     6   #写入数组元素
        sw      $t7, 0($t5)         #写入数组元素
        addi    $t4, $t4, -1        #$t4 = $t4 - 1
        j       in                  #跳转到in
exitIn: addi    $t3, $t3, 1     7   #$t3 = $t3 + 1
        j       out                 #跳转到out
print:                          8
```



8.2 局部优化

8.2.1 基本块定义与划分

8.2.2 程序的控制流图

8.2.3 基本块的DAG表示与应用

(基本块优化）

DAG（Directed Acyclic Graph）（复习)

无环路的有向图。

定义

设G是由若干结点构成的有向图，从结点n_i到结点n_j的有向边用$n_i \rightarrow n_j$表示。

① 若存在有向边序列$n_{i1} \rightarrow n_{i2} \rightarrow \ldots \rightarrow n_{im}$，则称结点$n_{i1}$与结点$n_{im}$之间存在一条路径，或称$n_{i1}$与$n_{im}$是连通的。

② 如果存在一条路径，其起始和终止于同一个结点，则称该路径是一个环路；

③ 如果有向图G中任一条路径都不是环路，则称G为无环路有向图。

基本块的DAG表示

结点之间关系类似于有向树

基本块的DAG是结点上带有下列标记的DAG

① 叶结点用标识符或常量作为其标记，当叶结点是标识符时，代表其初值，标识符加下标0；

② 内部结点用运算符标记，表示后继结点施加该运算后的值；

③ 各结点上可以附加一个或多个标识符，附加在同一结点上的多个标识符具有相同的值。

- 常见四元式分类：

0型 → 定值语句

=，B，　，A

1型 → 单目运算且定值

op，B，　，A

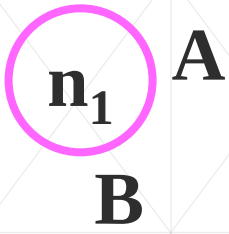
2型 → 双目运算、取数组元素且定值

op，B，C，A

=[]，B，C，A

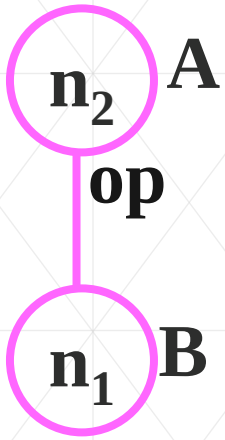
四元式与DAG对应关系

0型： (=，B，　，A)



一个结点

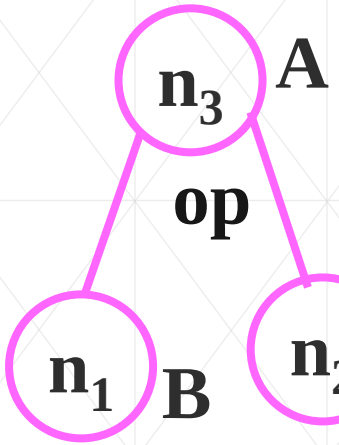
1型： (op，B，　，A)

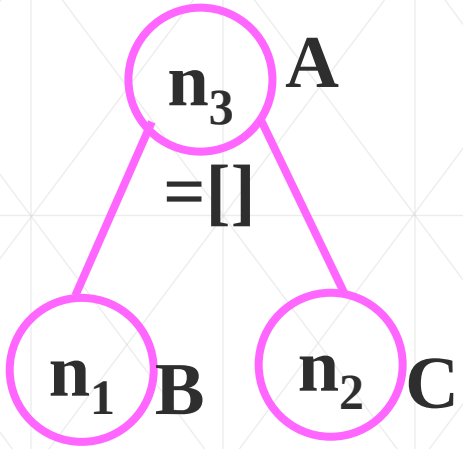


两个结点

2型： (op，B，C，A)

(=[]，B，C，A)





三个结点

- 算法(基本块的DAG的构造算法)

//初始化，置DAG为空。仅考虑0型、1型和2型①

输入：一个基本块i

输出：含有下列信息的基本块i的DAG：

适当数据结构存放的节点信息表。

叶结点、内部结点按统一标记；

标识符与结点的对应关系表（用NODE函数表示）；

0型：(=，B，，A)
1型：(op，B，，A)
2型：(op，B，C，A)

算法：

对基本块中每一四元式依次执行以下步骤

1. 构造操作数结点；

操作符处理

2. 捕捉已知量，合并常数； //删除原常数结点
3. 捕捉公共子表达式； //删除冗余的公共子表达式

赋值处理

4. 捕捉可能的无用赋值； //删除无用赋值

[1] //构造操作数结点

如果NODE(B)无定义，则增加该结点。令NODE(B)=n。

① 对0型，转[4]；

② 对1型，转[2]的①；

③ 对2型，若NODE(C)无定义，则增加该结点，并转[2]的②；

0型：(=，B，，A)
1型：(op，B，，A)
2型：(op，B，C，A)

[2] //常量合并

①如果NODE(B)是常数结点，则转[2]的③，否则转[3]的①； // 1型

②如果NODE(B)和NODE(C)都是常数结点，则转[2]的④，否则转[3]的②； // 2型

③ 执行op B,设得到的新常数为P。若NODE(P)无定义，则增加该结点，令NODE(P)=n，若NODE(B)是处理当前四元式时新建立的结点，则删除它。转[4] 。// 1型

④ 执行B op C，设得到的新常数为P。若NODE(P)无定义，则增加该结点，令NODE(P)=n。若NODE(B)或 NODE(C)是处理当前四元式时新建立的结点，则删除它。转[4] 。// 2型

[3]//捕捉并删除公共子表达式

① 检查DAG中是否有结点：标记为OP且唯一后继为NODE(B) 。若没有，增加该结点。令NODE(OP)=n；转[4] 。 // 1型

②检查DAG中是否有结点：标记为OP，其左后继为NODE(B)，右后继为 NODE(C)。若没有，增加该结点。令NODE(OP)=n；转[4] 。//对2型

[4] //捕捉并删除无用赋值

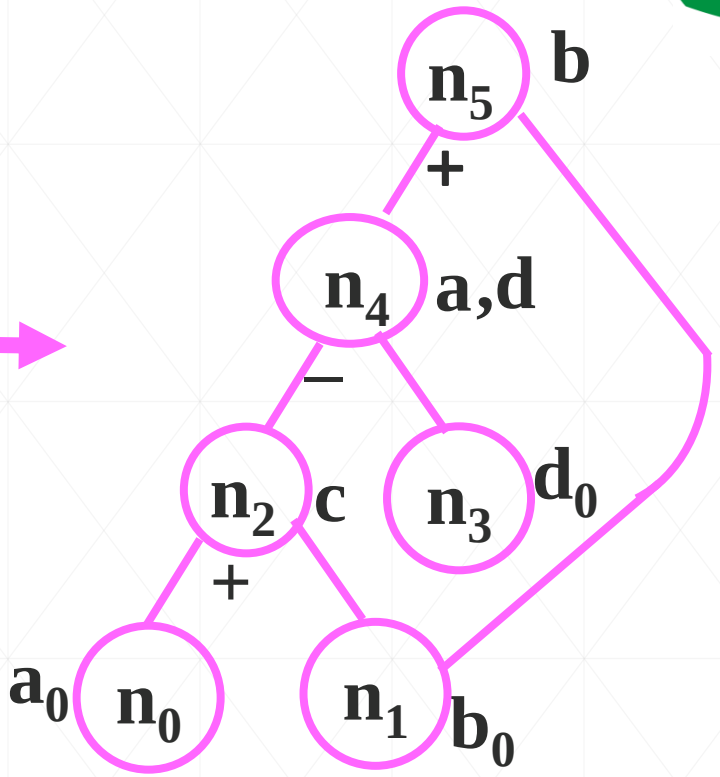
如果NODE(A)无定义，则令NODE(A)=n；否则，先把A从NODE(A)结点的附加标识符集中删除(如果NODE(A)是叶子结点，则其标记A不删除)，令NODE(A)=n。转处理下一四元式。

0型：(=，B，，A)
1型：(op，B，，A)
2型：(op，B，C，A)

例：设有基本块如下

```
+ a b c
– c d a
+ a b b
– c d d
```

DAG



删除了公共子表达式

按照DAG的结点顺序，重写代码，就是经过局部优化后的代码。

```
+ a b c
– c d a
= a   d
+ a b b
```

例：一基本块的语句序列如下，构造其DAG

(1) =, 3.14, , T_0

(2) *, 2, T_0, T_1

(3) +, R, r, T_2

(4) *, T_1, T_2, A

(5) =, A, , B

(6) *, 2, T_0, T_3

(7) +, R, r, T_4

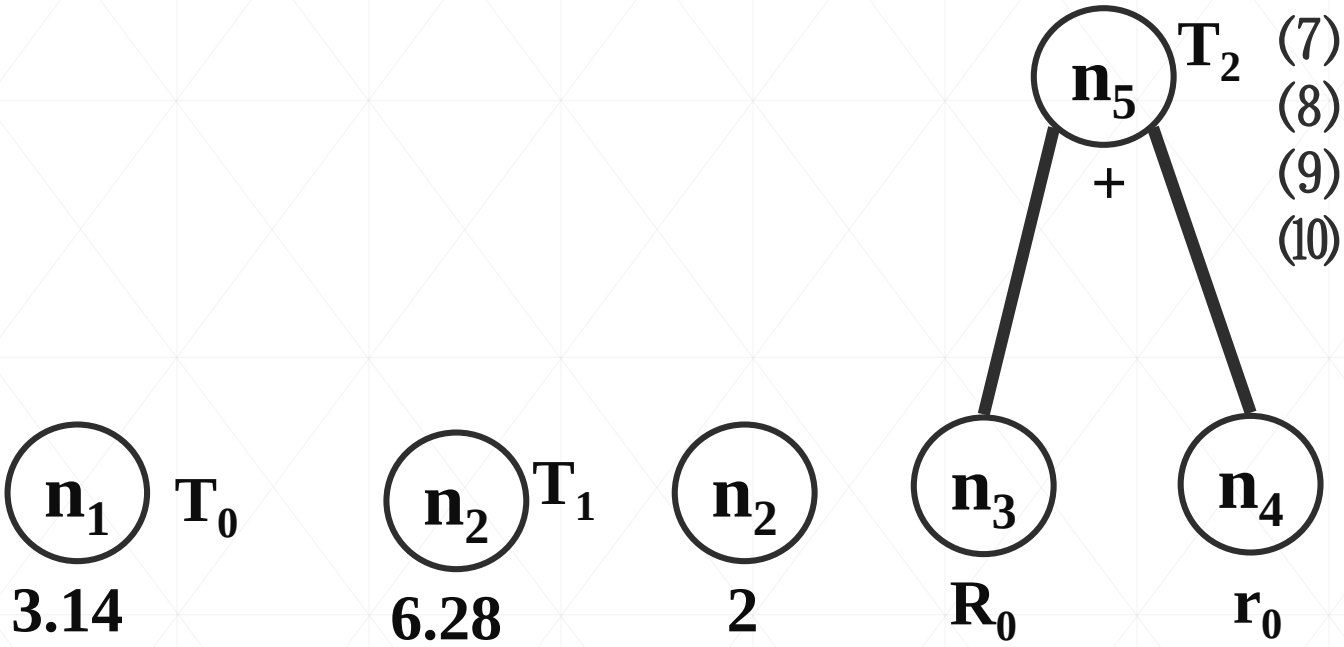
(8) *, T_3, T_4, T_5

(9) -, R, r, T_6

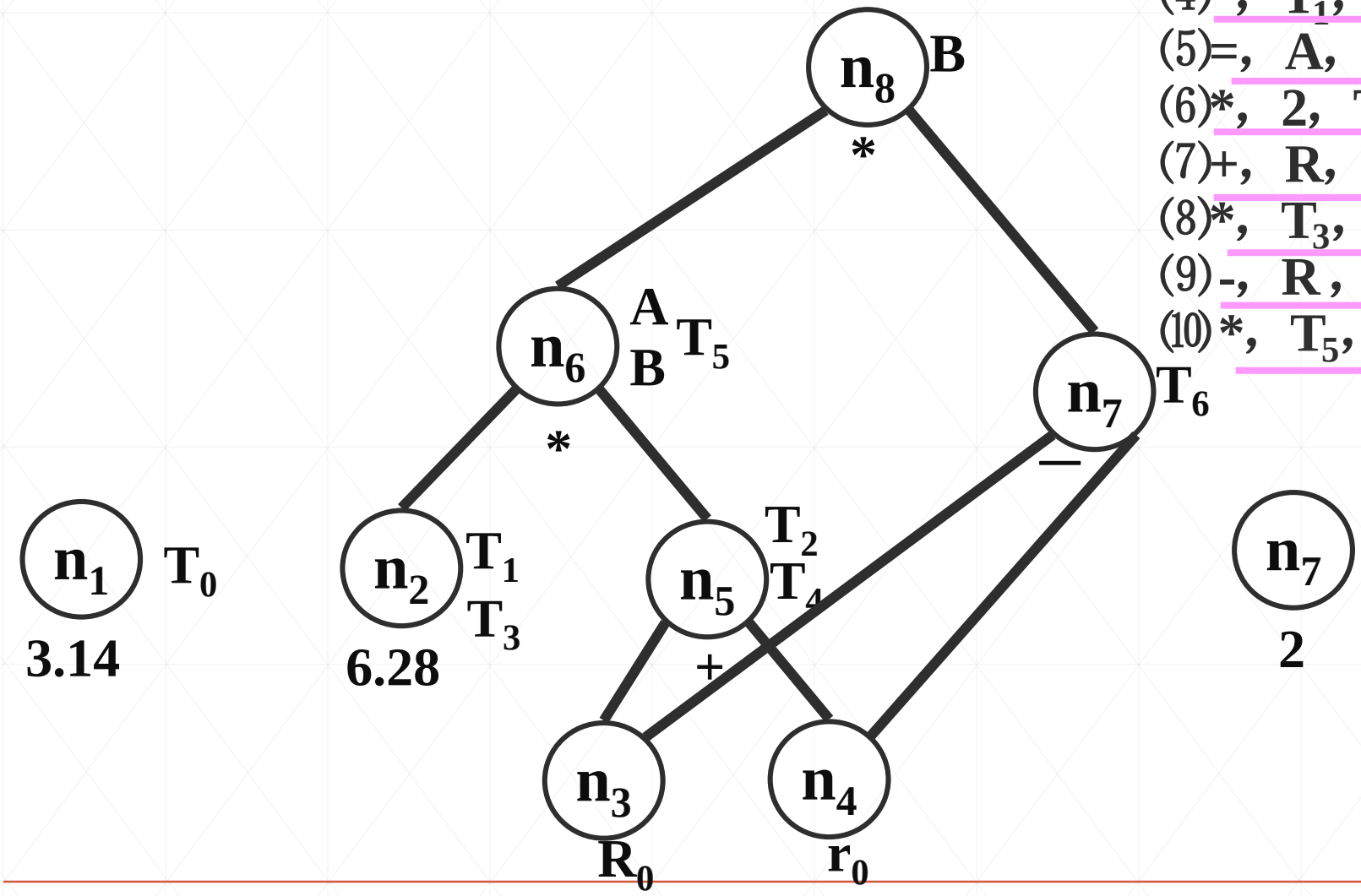
(10) *, T_5, T_6, B

解：构造DAG的过程如下：

(1)=， 3.14， ， T_0
(2)*， 2， T_0 ， T_1
(3)+， R， r， T_2
(4)*， T_1， T_2， A
(5)=， A， ， B
(6)*， 2， T_0， T_3
(7)+， R， r， T_4
(8)*， T_3， T_4， T_5
(9) -， R ， r， T_6
(10) *， T_5， T_6， B

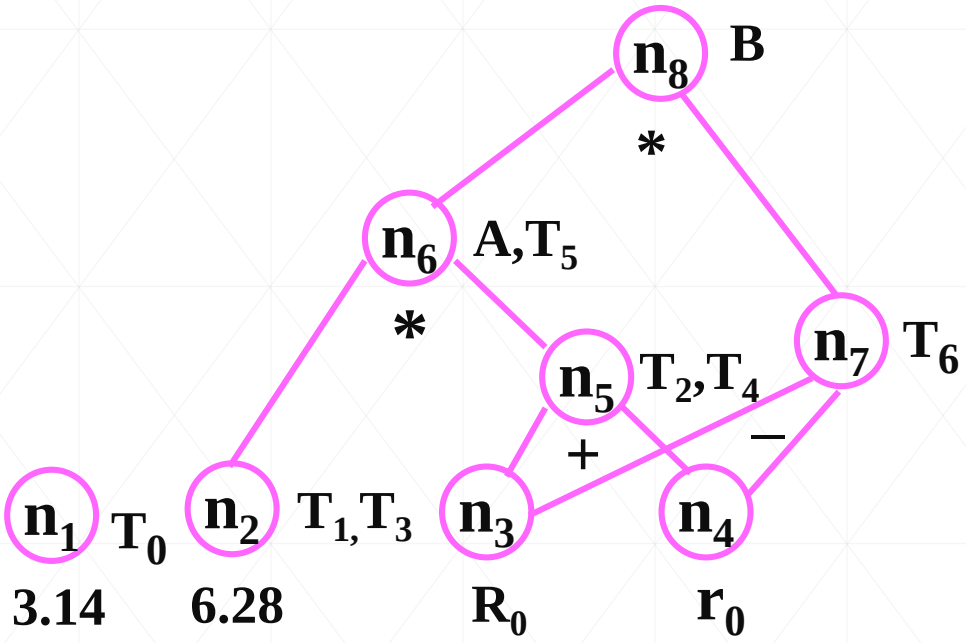


解：构造DAG的过程如下：



(1)=, 3.14, , T_0
(2)*, 2, T_0, T_1
(3)+, R, r, T_2
(4)*, T_1, T_2, A
(5)=, A, , B
(6)*, 2, T_0, T_3
(7)+, R, r, T_4
(8)*, T_3, T_4, T_5
(9)-, R, r, T_6
(10)*, T_5, T_6, B

(1)=, 3.14, , T_0 (2)*, 2, T_0, T_1 (3)+, R, r, T_2 (4)*, T_1, T_2, A
(5)=, A, , B (6)*, 2, T_0, T_3 (7)+, R, r, T_4 (8)*, T_3, T_4, T_5
(9)-, R, r, T_6 (10)*, T_5, T_6, B

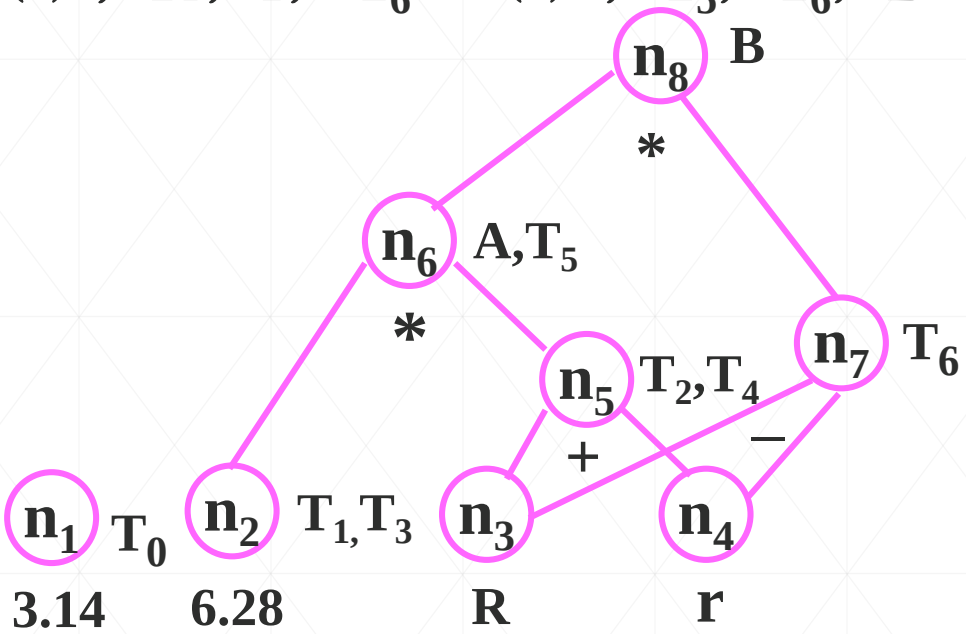


(1) =, 3.14, , T_0
(2) =, 6.28, , T_1
(3) =, 6.28, , T_3
(4) +, R, r, T_2
(5) =, T_2, , T_4
(6)*, 6.28, T_2, A
(7) =, A, , T_5
(8) -, R, r, T_6
(9)*, A, T_6, B

按照DAG的结点顺序，重写代码。

所做优化：常量合并、公共子表达式删除、无用赋值删除

(1)=, 3.14, , T_0 (2)*, 2, T_0, T_1 (3)+, R, r, T_2 (4)*, T_1, T_2, A
(5)=, A, , B (6)*, 2, T_0, T_3 (7)+, R, r, T_4 (8)*, T_3, T_4, T_5
(9)-, R, r, T_6 (10)*, T_5, T_6, B



如果知道某些变量在此基本块出口处不活跃(之后不再被引用)，其值就不用再“复制”。

若T3、T4在出口处不活跃。

(1) =, 3.14, , T_0
(2) =, 6.28, , T_1
(3) =, 6.28, , T_3
(4) +, R, r, T_2
(5) =, T_2, , T_4
(6)*, 6.28, T_2, A
(7) =, A, , T_5
(8) -, R, r, T_6
(9)*, A, T_6, B

注意：

流图的一个结点是一个基本块，基本块可用DAG表示。

流图确认的是基本块之间的关系，

DAG确认的是基本块内各四元式间的关系。

第 8 章 代码优化（optimization）

引入本节的原因

- **循环优化的重要性**：循环是程序中反复执行的代码序列，实施循环优化，将高效提高目标代码质量。

- **循环优化的技术准备**：**循环查找**；控制流和数据流分析。

通过控制流分析查找循环。

构成循环条件

具有下列性质的结点序列构成一个循环：

1. 强连通性。

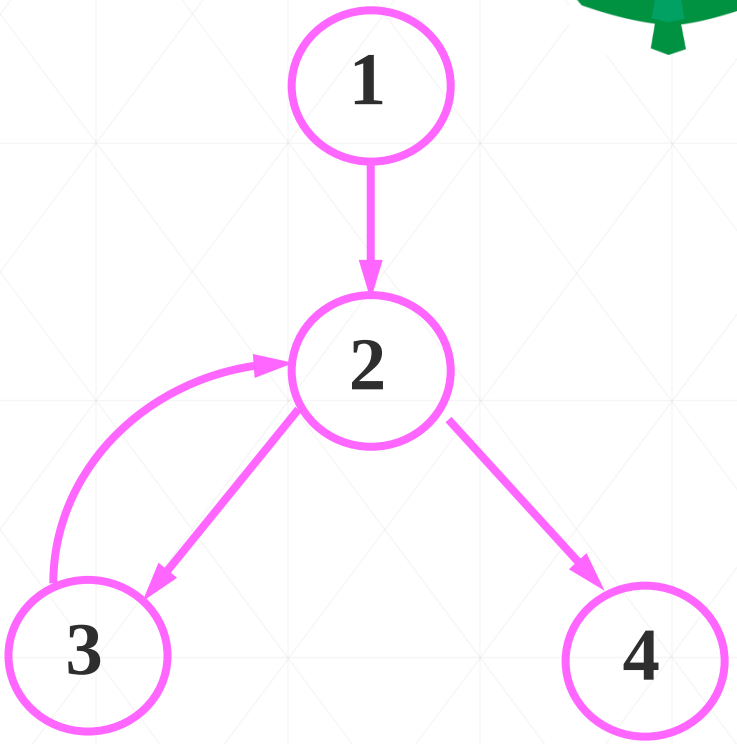
任意两个结点之间必有一条通路，且通路上的任何结点都属于该序列。

2. 入口惟一。

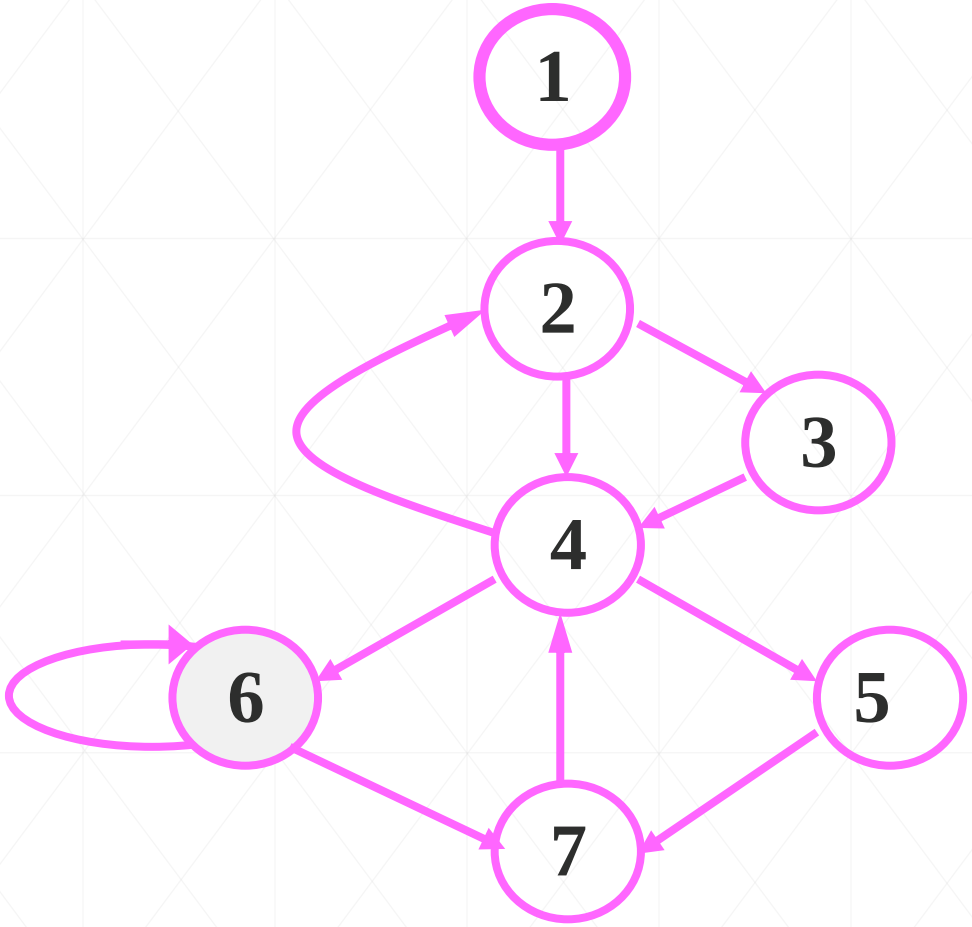
入口：流图的首结点或结点序列外某结点有一条有向边引到的结点。

例： 如右图，

{2，3}是循环
- 强连通性成立
- 惟一入口结点2



例如下图，



循环：

{6} 强连通/入口6

{4,5,6,7} 强连通/入口4

{2,3,4,5,6,7} 强连通/入口2

非循环：

{2,4} 强连通/入口2,4

{2,3,4} 强连通/入口2,4

{4,6,7} 强连通/入口4,7

必经结点、必经结点集与回边

- 定义（必经结点）

在程序流图G中，n_i和n_j为任意结点。若从n_0出发，到达n_j的任何一条通路都必经过n_i，则称n_i是n_j的必经结点，记作n_i DOM n_j。

- 定义（必经结点集）

在程序流图G中，结点n的全部必经结点，称为结点n的必经结点集，记作D(n)。

- **DOM是流图结点集上一个偏序关系：**

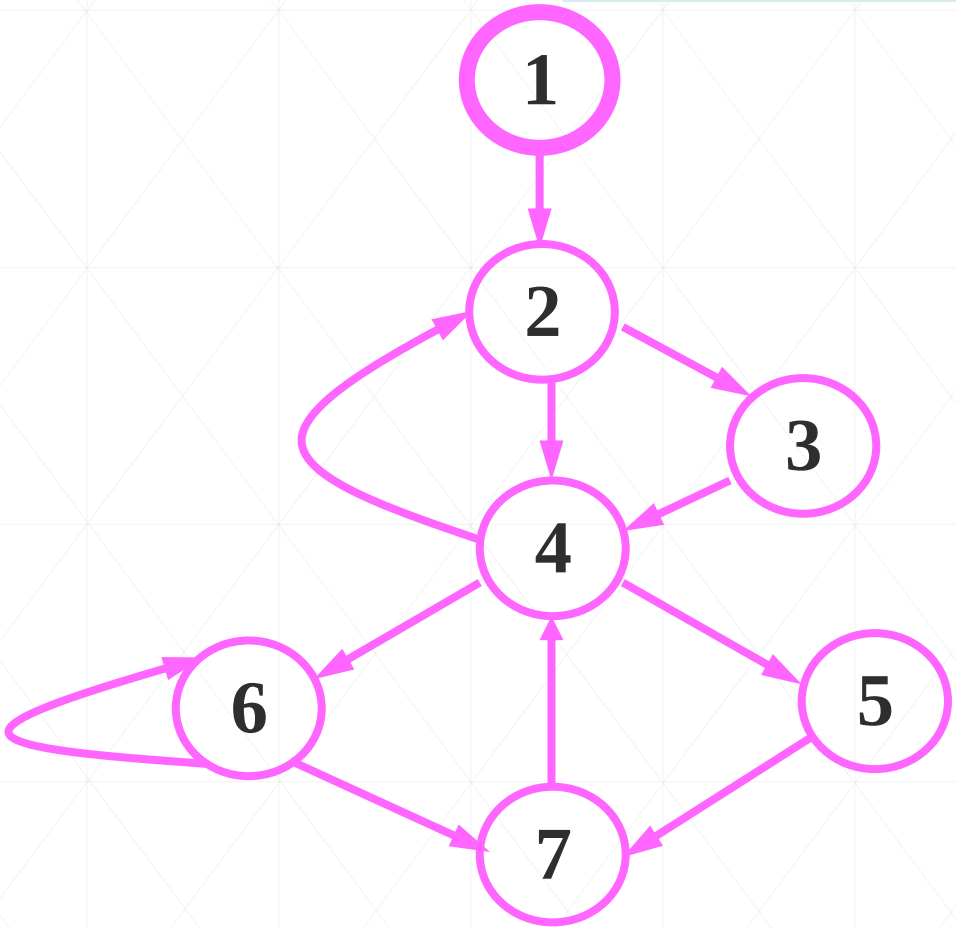
(1) 自反性： a DOM a

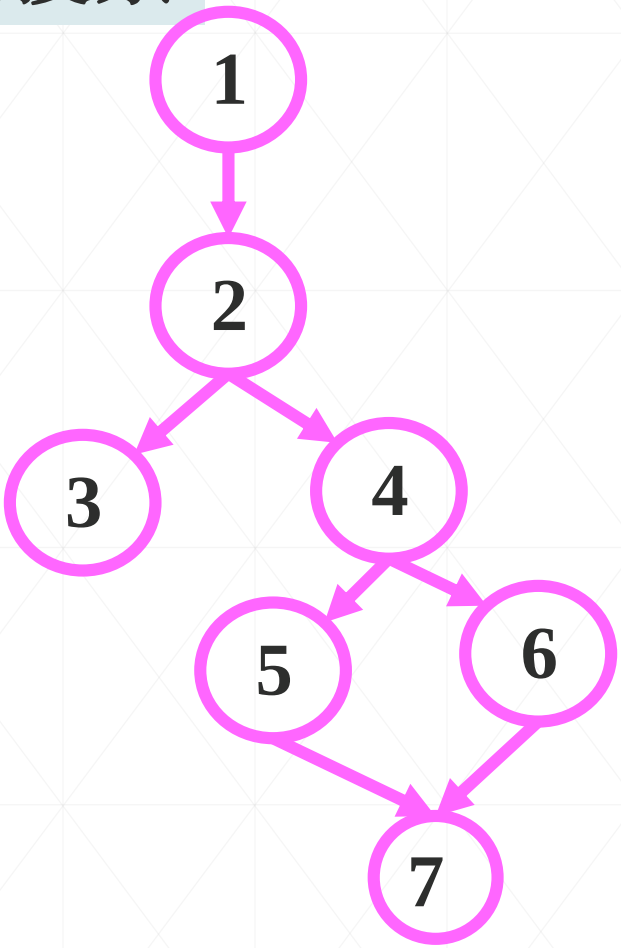
(2) 传递性：如果a DOM b, b DOM c，
则有： a DOM c 。

(3) 反对称性：若有 a DOM b, b DOM a,
则有： a = b。

例:设有如下流图,求出所有结点的必经结点集

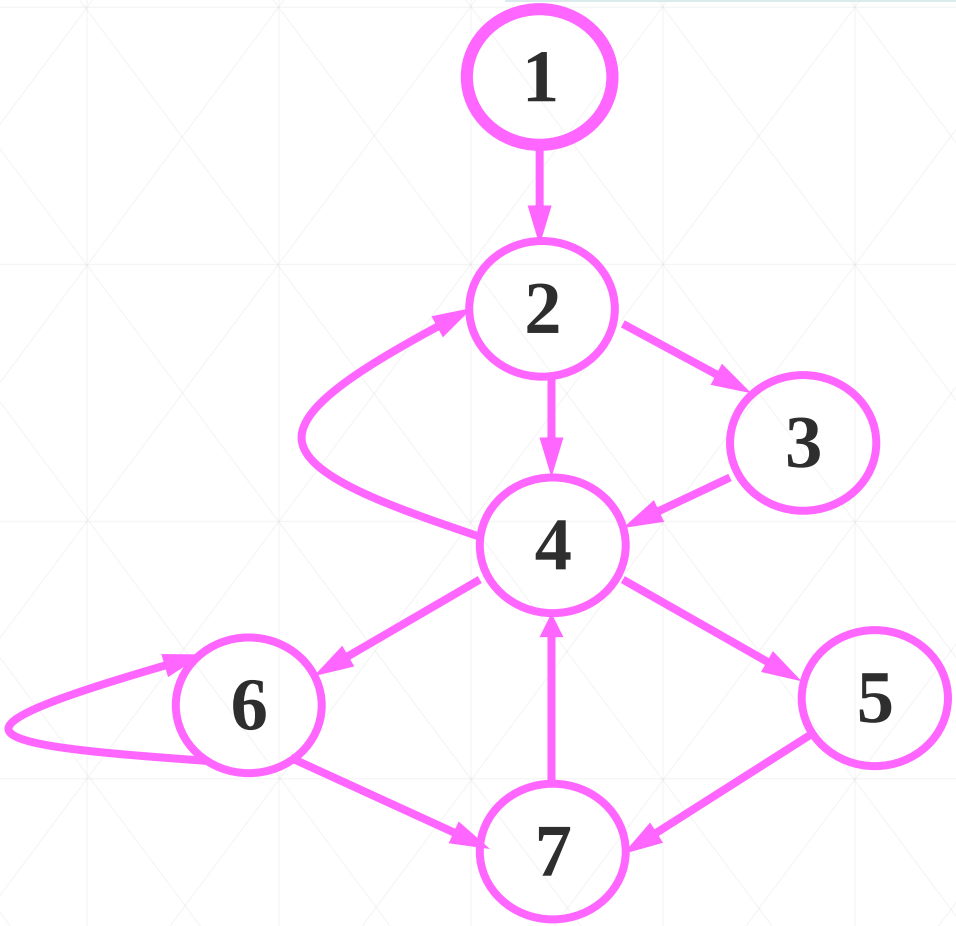
流图做广度优先搜索





例:设有如下流图,求出所有结点的必经结点集

根据广度优先搜索序列写出必经结点集



D(1)={1}

D(2)=D(1)∪{2}= {1,2}

D(3)=D(2)∪{3}={1, 2, 3}

D(4)=D(2)∪{4}={1, 2, 4}

D(5)=D(4)∪{5}={1, 2, 4, 5}

D(6)=D(4)∪{6}={1, 2, 4, 6}

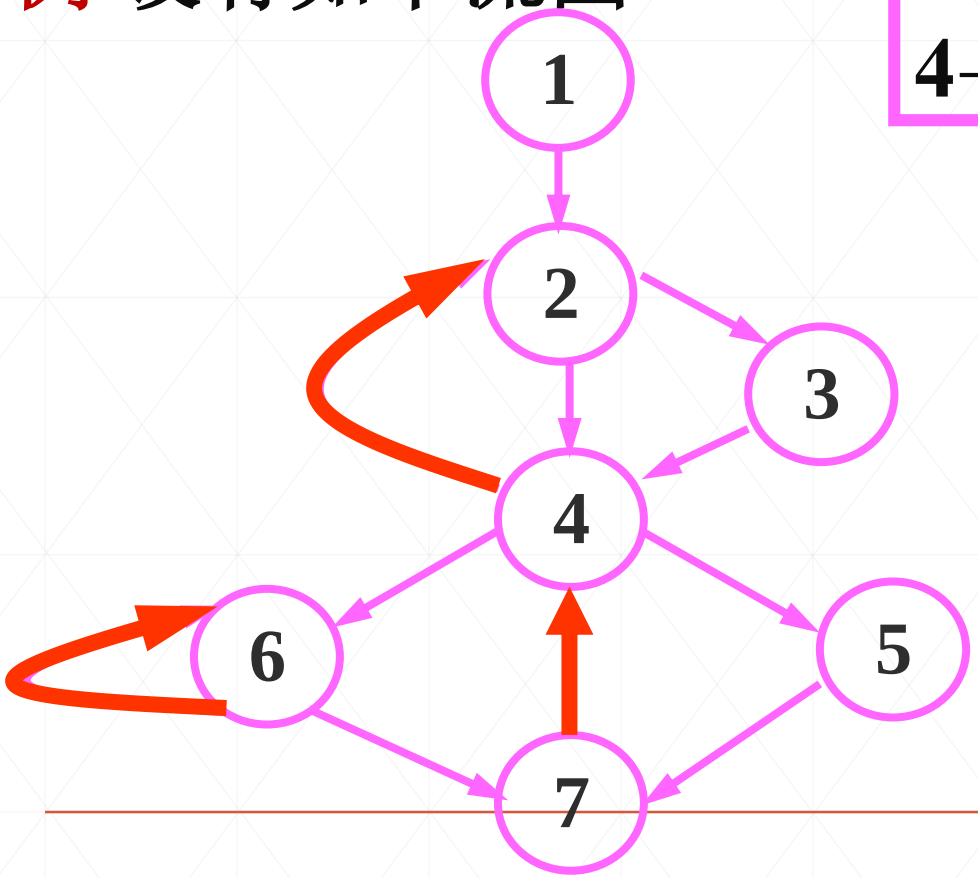
D(7)=(D(5)∩D(6))∪{7}

={1, 2, 4, 7}

定义（回边）

a→b是流图G中一条有向边，如果b DOM a，则称a→b是流图G中的一条回边。记作<a, b>。

例 设有如下流图



流图中的回边有3条：
4→2，6→6和7→4 。

D(1) = {1}

D(2) = {1,2}

D(3) = {1, 2, 3}

D(4) = {1, 2, 4}

D(5) = {1, 2, 4, 5}

D(6) = {1, 2, 4, 6}

D(7) = {1, 2, 4, 7}

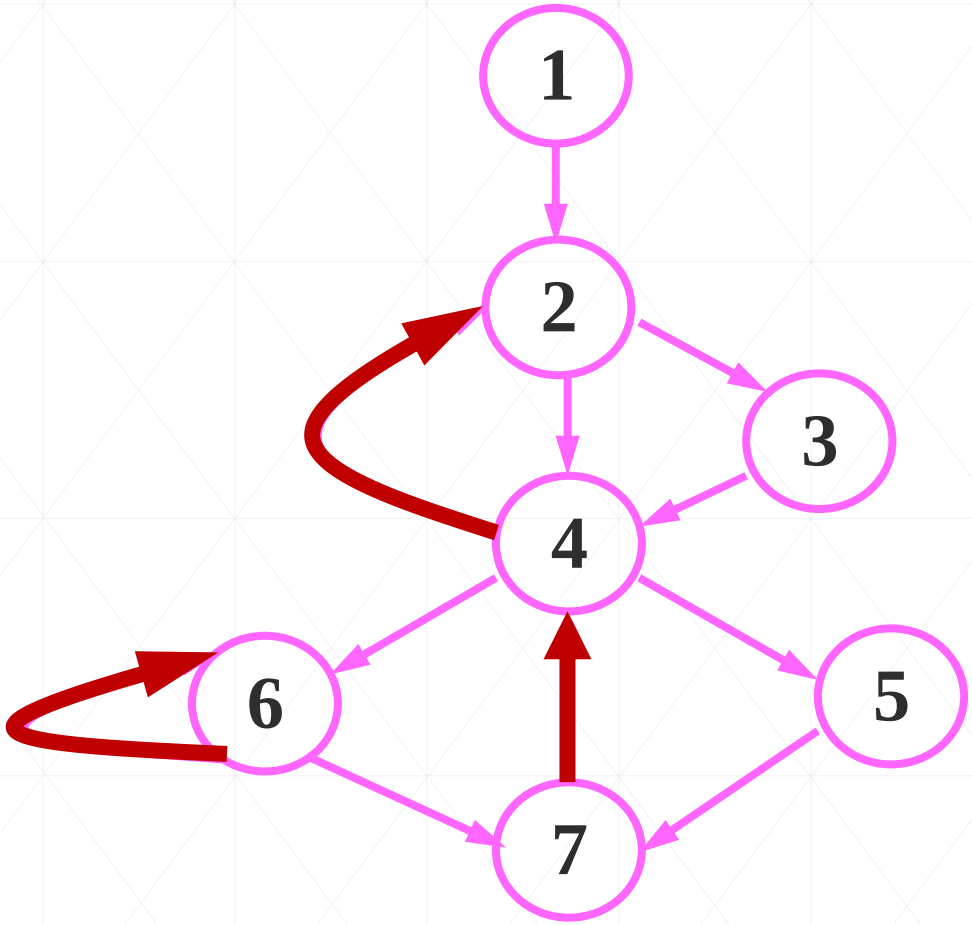
利用回边求出流图中的循环：

若<n, d>是一回边，则由结点d、结点n以及所有通路到达n而该通路不经过d的所有结点序列构成一个循环L，d是循环L的惟一入口。

求解算法：

```
loop＝{d};
{if n is not in loop
    {loop=loop∪{n};push n onto stack;}}
while stack is not empty
{pop m;for each predecessor p of m do
  {if p is not in loop
    {loop=loop∪{p};push p onto stack;}}}
```

例：设有如下流图



流图中的循环：

<4, 2> loop

= { 2 , 4 , 3, 7 , 5, 6 }

<6, 6> loop

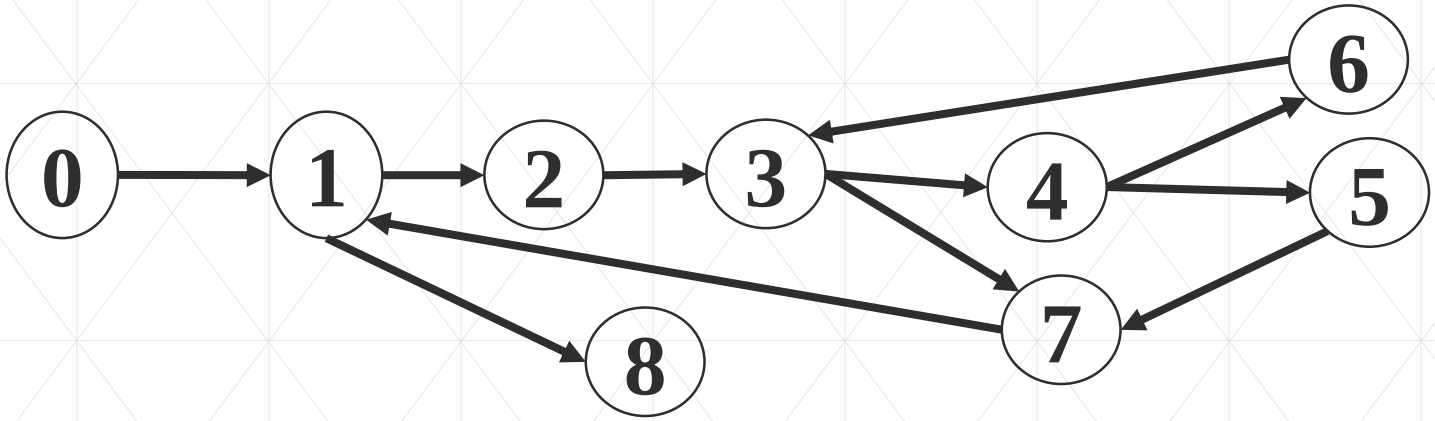
= { 6}

<7, 4> loop

= { 4 , 7 , 5, 6}

设置

如下图所示的程序控制流图中有【 】个循环



A 1

B 2

C 3

D 4

提交

➢ summary (查找循环步骤)

1. 确定G的所有节点的D(n)；

2. 由D(n)找回边；

3. 通过回边确定循环。

第 8 章 代码优化（optimization）

一. 数据流分析基础

▪ 数据流分析

涉及多个基本块范围的优化，

编译程序需要知道相关基本块中的数据如何

沿着执行路径流动的信息，

此信息叫数据流信息，

收集信息的工作称为数据流分析。

- 点：数据信息采集位置

一个中间语言语句在代码序列中的位置。

或语句在流图基本块中的位置。

基本块入口点：第一个中间代码的前位置

基本块出口点：最后一个中间代码后位置

例如

d_i: → x=a*b;

d_{i+1}: → read x ;

d_{i+2}: →

- **定值点：**

变量x获得值的中间代码的位置d，称为x的定值点。

例如，d_i: x=a*b+c; d_j: read x ;

定值方式：
- 赋值语句
- 输入语句
- 函数调用的形参与实参结合

- **引用点：**

  引用变量x的中间代码的位置d，称为x的引用点。

如，　d_j：　i = i+1

定值　　引用

如，　d_k：　i ++

引用/定值

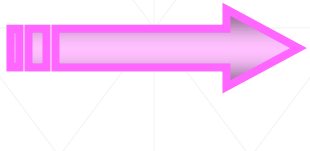
- 到达 — 定值：

在流图G，从点d有一通路到达点p且该通路上没有对变量A的再定值，则称变量A在点d的定值到达点p。

约定：　<A> — 对变量A的引用；

A — 对变量A的定值；

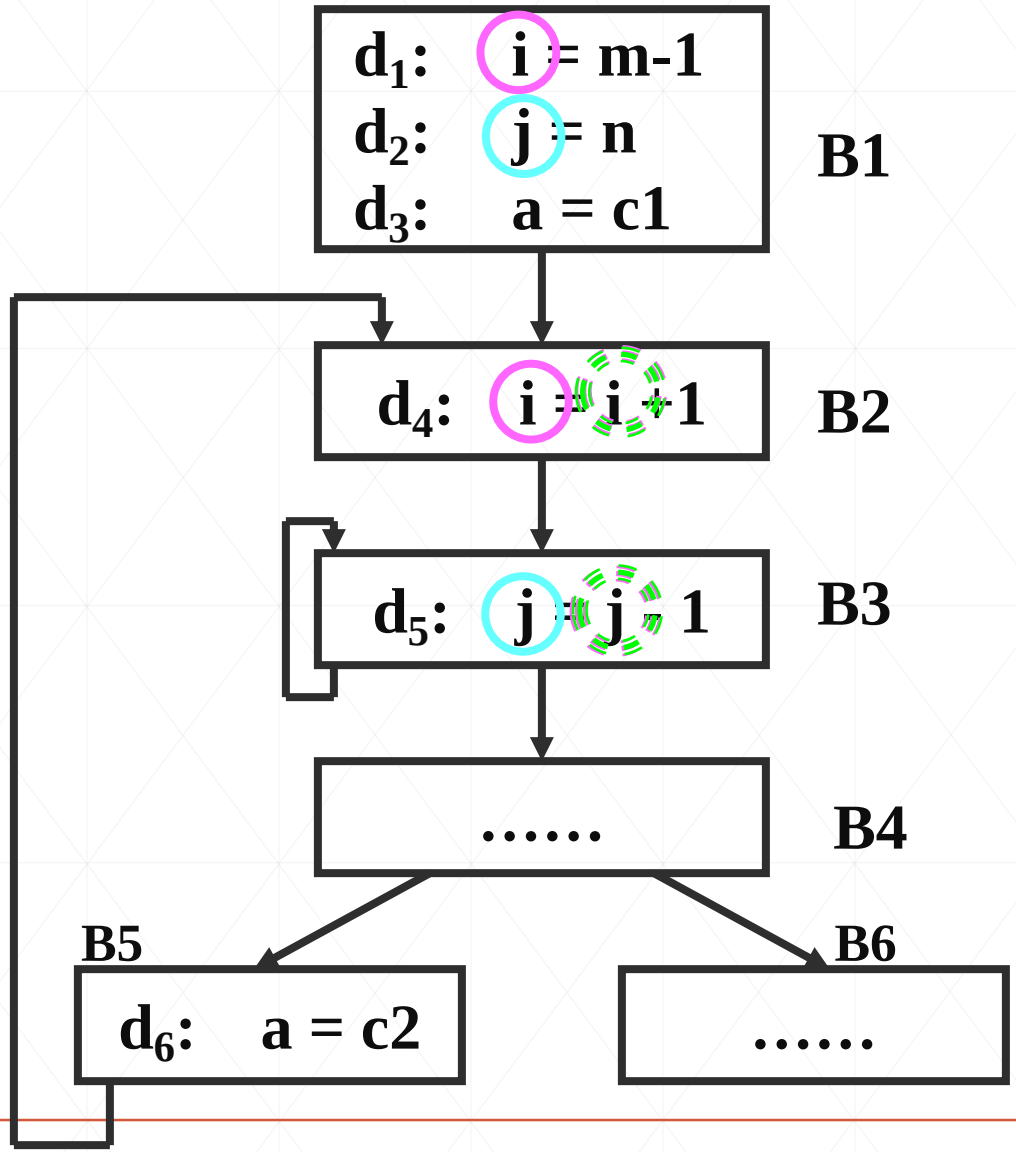
d: A



有此路径，且无对变量A的其他定值

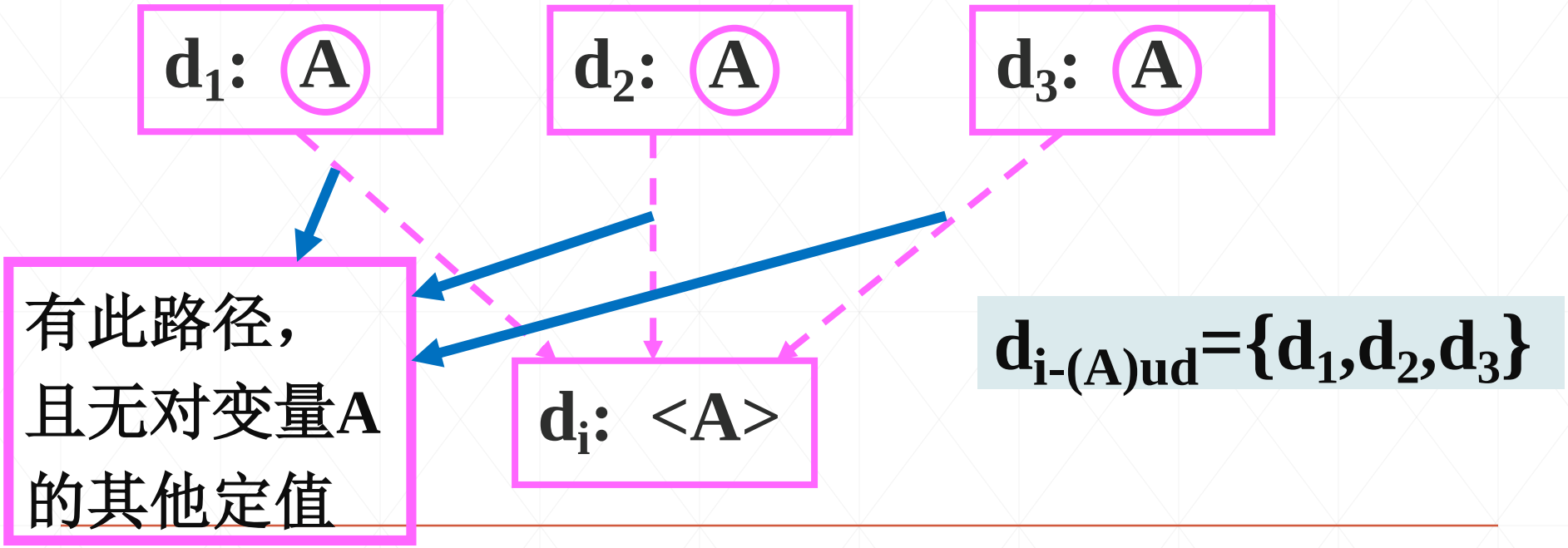
p: …

例 设有如下流图



定义（ud链）

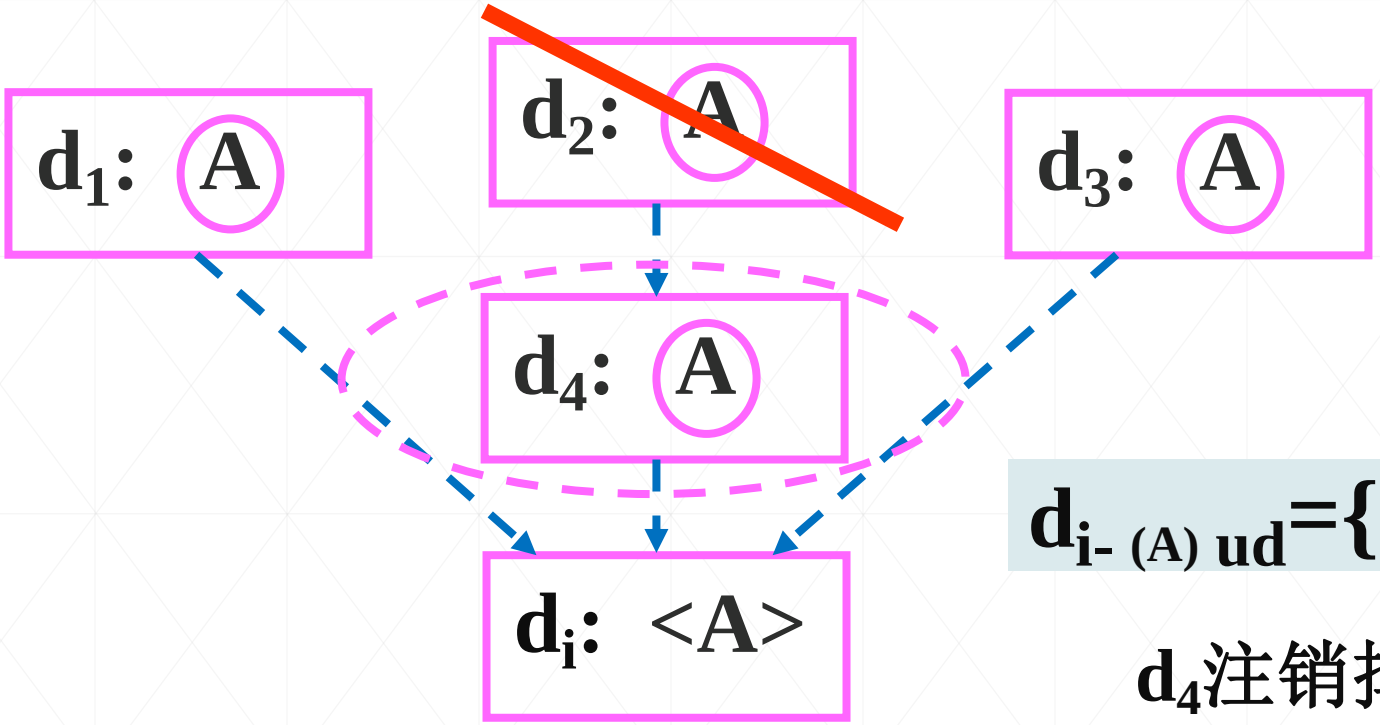
假设在程序中某点P引用了变量A的值，则把流图中能到达P的A的定值点的全体，称为A在引用点P的引用—定值链（即ud链）。



☺ ud链是相对于引用点的定值情况。

变量A在点d的引用的ud链:

所有能到达d点的A的定值点。



$d_{i-(A)\,ud}=\{d_1,d_4,d_3\}$

d_4注销掉d_2

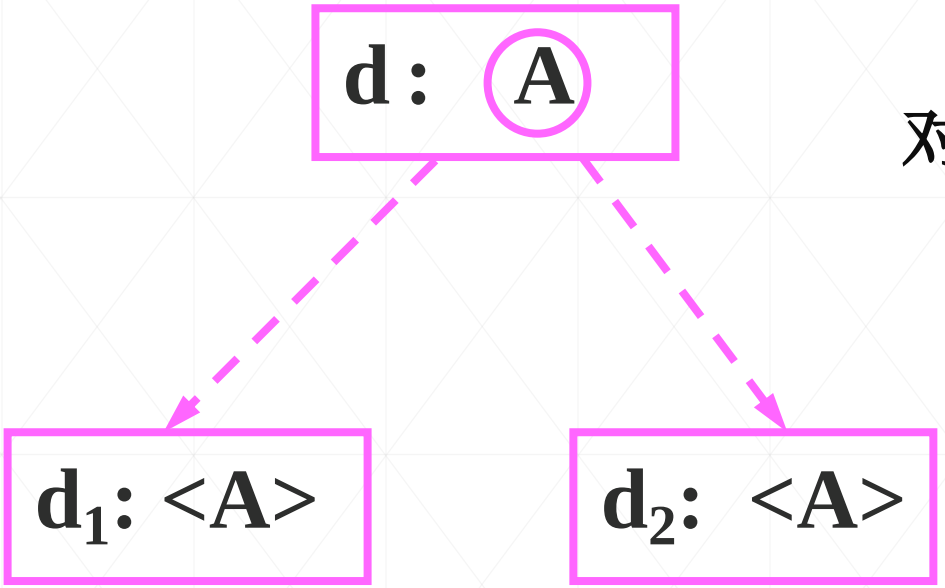
- 定义(du链)

假设在程序中某点P对一个变量A定值，则把该定值能到达的A的引用点的全体，称为A在定值点P的定值—引用链（即du链）。

☺ du链是相对于定值点的引用情况。

变量A在点d的定值的du链，
定值到达的所有引用点。

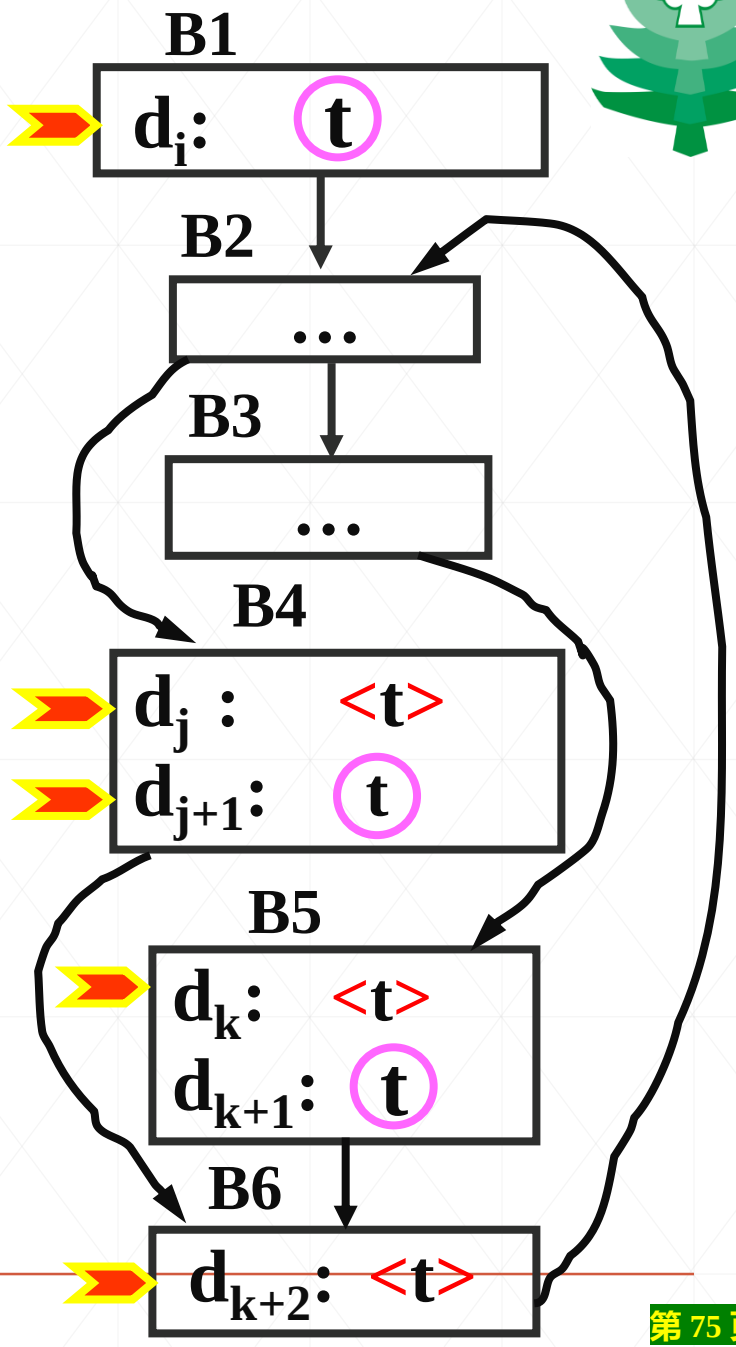
- du链



对变量A：

$d\text{-}_{(A)du} = \{d_1, d_2\}$

例 设有流图



t在点d_{k+2}的ud链 $=\{d_{j+1},d_{k+1}\}$

t在点d_k的ud链$=\{d_i,d_{j+1},d_{k+1}\}$

t在点d_j的ud链$=\{d_i,d_{j+1},d_{k+1}\}$

t在点d_i的du链$=\{d_j\ ,\ d_k\ \}$

t在点d_{j+1}的du链

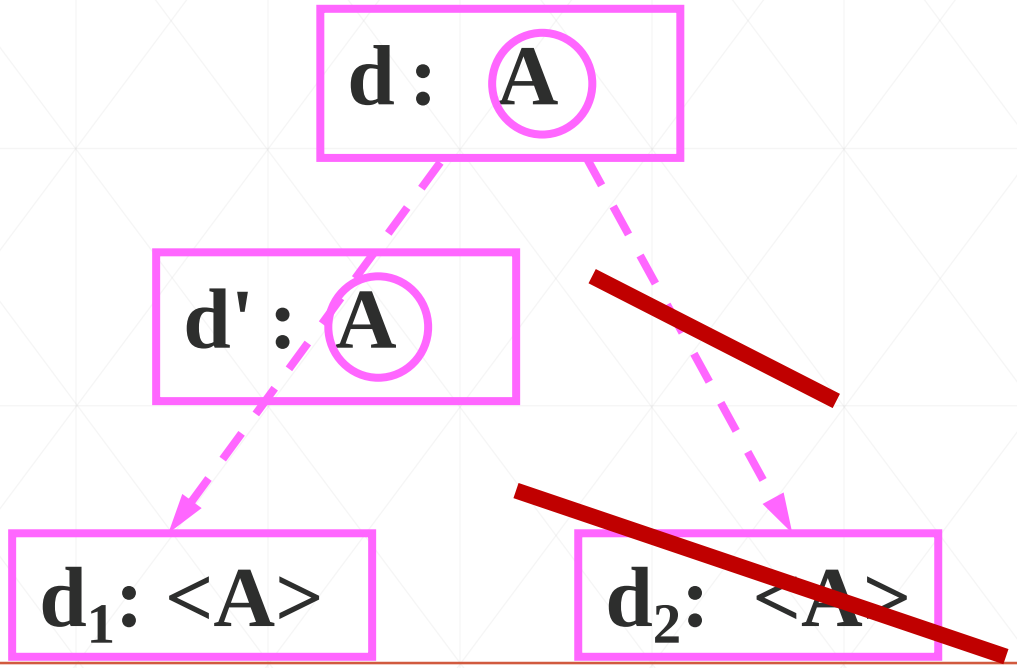
$=\{\ d_{k+2}\ ,d_k,d_j\}$

t在点d_{k+1}的du链

$=\{\ d_{k+2}\ ,d_k,d_j\}$

活跃变量：

在程序中对某变量A和某点P，如果存在一条从P开始的通路，其中引用了A在点P的值，则称A在点P是活跃的，否则称A在点P是死亡的。



A在点d活跃

A在点d, d'活跃

A在点d'活跃，
在点d死亡

二. 重要数据流方程

编译器把程序的一部分或全部看作一个整体来收集信息，并把收集的信息分配给流图中的各个基本块。

- 到达定值信息 —求解ud链信息；常数合并
- 活跃变量信息 —求解du链信息；删除无用赋值；
- 公共子表达式信息 — 删除冗余运算。

- 典型的数据流方程

前 → 后　　数据流控制流方向一致

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

后 → 前　　数据流控制流方向相反

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

信息采集单位B：流图中某个基本块，或语句

出来的信息是产生(gen)信息加上没有被注销(kill)的进去的信息；

使用数据流方程的注意事项

1. 产生、注销的概念依赖所需要的信息

2. 进入的信息依赖数据流方向

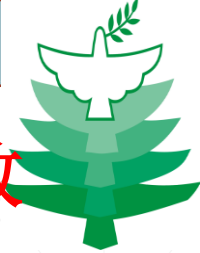
   前 → 后　　数据流控制流方向一致

   in[B]由前驱基本块的信息决定

   后 → 前　　数据流控制流方向相反

   out[B]由后继基本块的信息决定

3. 由于数据沿流图的控制路径流动，故数据流分析受程序控制结构影响。



- 到达—定值数据流方程 数据流控制流方向一致

采集程序中变量的定值情况的数据流分析

(即到达点P的各变量的全部定值点信息)。

$in(B_i)$: 能到达基本块B_i入口点的各个变量的所有定值点集。

$out(B_i)$: 能到达基本块B_i出口点的各变量所有定值点的集合。

$gen(B_i)$: 在B_i中定值且能到达B_i出口点的所有定值点集。

$kill(B_i)$: 在基本块B_i中定值的变量在所有其它基本块的定值点的集合。

到达—定值方程

$$\begin{cases} out(B) = gen(B) \cup (in(B) - kill(B)) \\ in(B) = \cup out(P) \quad P \in Pred(B) \end{cases}$$

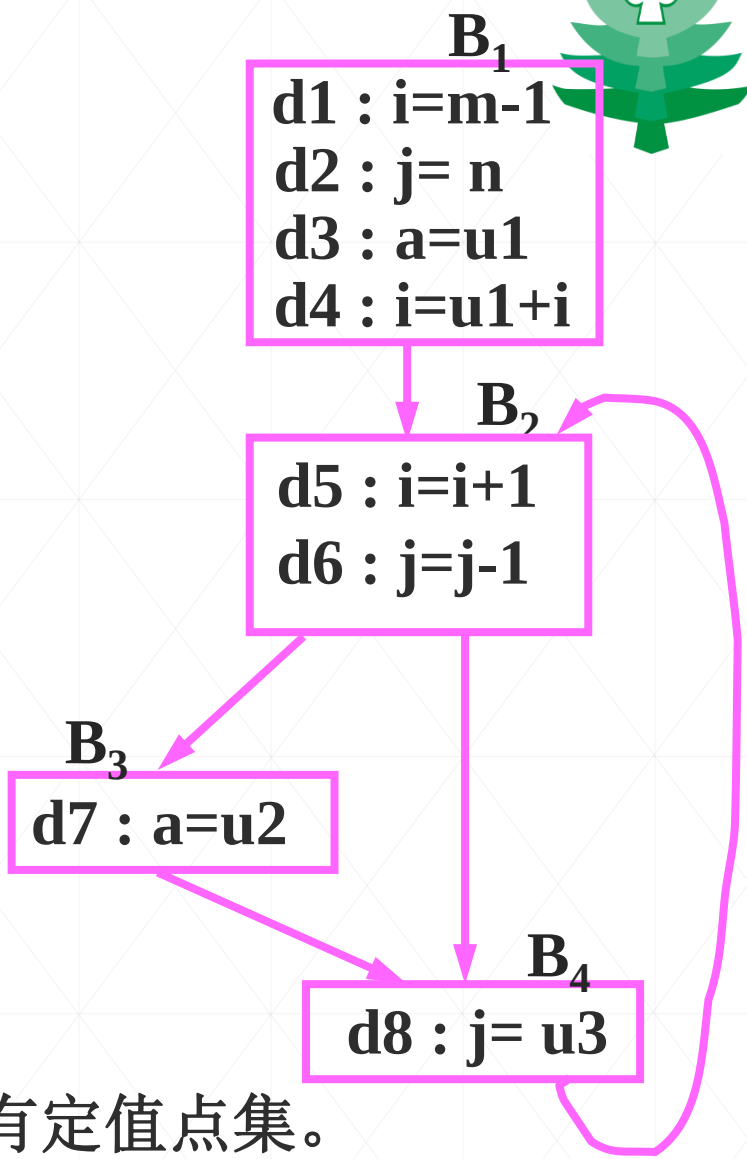
其中：

gen(B)和kill(B)从其定义出发，直接从给定的代码求出。

Pred(B)表示B的前驱基本块。

例：设有流图

	gen(B)	kill(B)
B_1	{d2(j) ,d3(a) , d4(i) }	{d5(i),d6(j), d7(a),d8(j) }
B_2	{d5(i), d6(j)}	{d1(i), d2(j), d4(i),d8(j)}
B_3	{d7(a) }	{d3(a) }
B_4	{d8(j)}	{d2(j), d6(j) }



gen(B_i): 在B_i中定值且能到达B_i出口的所有定值点集。

kill(B_i):在基本块B_i中定值的变量在其它基本块的定值点的集合。

对out(B)，可由以下条件得到：

① 如果某定值点d在in(B)中，而且在d定值的变量在B中未被重新定值，则d也在out(B)中；

② 如果定值点d在gen(B)中,则它一定在out(B)中；

③ 除以上两种情况外，没有其它定值点d∈out(B)。

对in(B)：

某定值点d到达基本块B的入口点，当且仅当它到达B的某一前驱基本块的出口点。

即in(B)是B的所有前驱基本块的out之并。

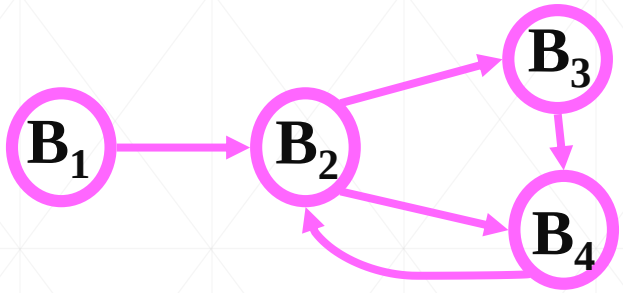
算法 （到达—定值）

输入：G及G中每个基本块B的kill[B]和gen[B]

输出：G中每个基本块B的in[B]和out[B]

```
{
 for (i=1;i<=N;i++)
     {
     in[Bi]=Φ; out[Bi]=gen[Bi];          /* in[Bi]和out[Bi]的迭代初值 */
     }
     change="真"; /*change记录相继2次迭代所得的in[Bi]之值.不等则为"真"，
     while (change)  需要继续迭代；若相等，则迭代过程结束，其值为"假" */
            { change= "假";
              for (i=1;i<=N;i++)
              {  NEWIN=∪out[P];     /*   P∈Pred(Bi) */
                 if ( NEWIN != in[Bi] ) /* NEWIN记录每次迭代后IN[Bi] 的新值  */
                    {  change= "真";
                       in[Bi]= NEWIN;
                       out[Bi]=gen[Bi]∪( in[Bi]-kill[Bi]);
                     }
               }
            }
}
```

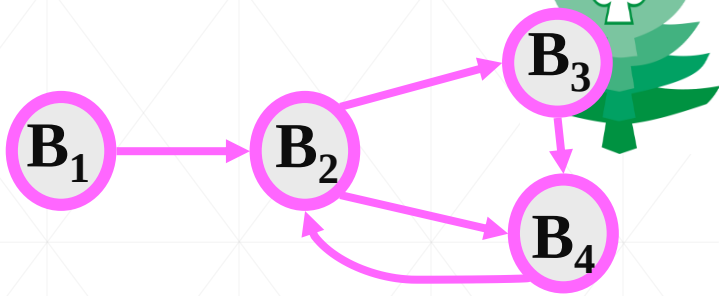
例： 设有程序的流图及每个流图的数据信息，求每个基本块的到达——定值信息



	gen(B)	kill(B)
B_1	{d2(i), d3(j) ,d4(a) }	{d5(i),d6(j), d7(a),d8(j) }
B_2	{d5(i), d6(j)}	{ d1(i) , d2(j) , d4(a) ,d8(j) }
B_3	{d7(a) }	{d3(a) }
B_4	{d8 (j)}	{d2(j), d6(j) }

in(B1)	Φ
out(B1)	{d2(i), d3(j) ,d4(a) }
in(B2)	Φ
out(B2)	{d5(i), d6(j)}
in(B3)	Φ
out(B3)	{d7(a) }
in(B4)	Φ
out(B4)	{d8(j)}

	gen(B)	kill(B)
B_1	{d2(i), d3(j),d4(a)}	{d5(i),d6(j),d7(a),d8(j)}
B_2	{d5(i),d6(j)}	{d1(i),d2(i),d4(j),d8(j)}
B_3	{d7(a) }	{d3(a)}
B_4	{d8 (j)}	{d2(j), d6(j)}



in(B1)	Φ
out(B1)	{d2(i), d3(j) ,d4(a)}
in(B2)	Φ
out(B2)	{d5(i), d6(j)}
in(B3)	Φ
out(B3)	{d7(a) }
in(B4)	Φ
out(B4)	{d8(j)}

in(B1)	Φ
out(B1)	{d2(i),d3(j) ,d4(a)}
in(B2)	{d2(i), d3(j),d4(a),d8(j)}
out(B2)	{d3(j), d5(i),d6(j)}
in(B3)	{d3(j), d5(i),d6(j)}
out(B3)	{d5(i), d6(j), d7(a)}
in(B4)	{d3(j), d5(i), d6(j),d7(a)}
out(B4)	{d3(j),d5(i), d7(a), d8(j)}

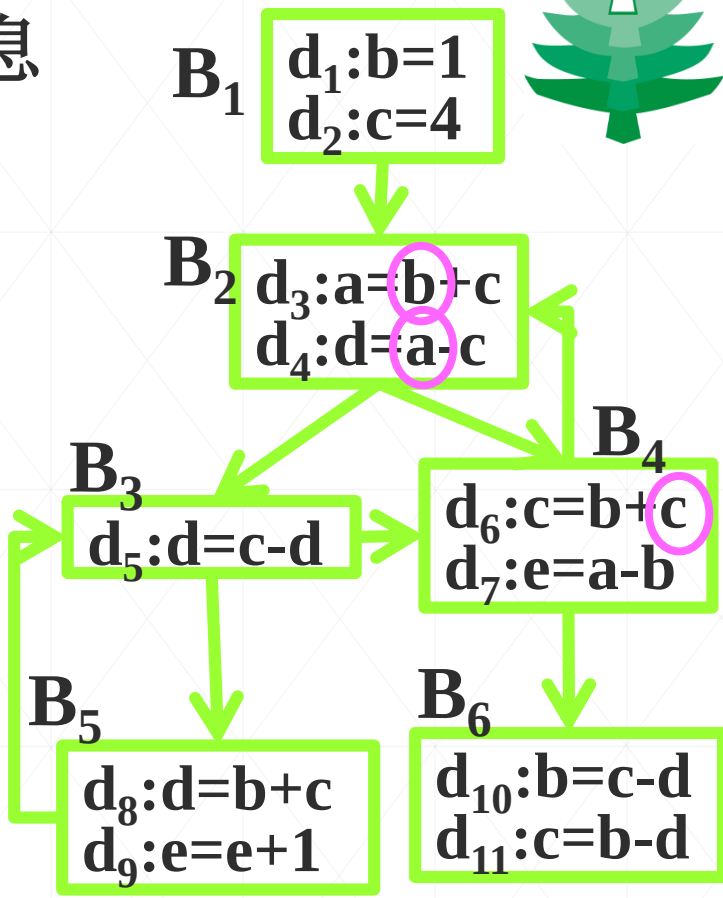
利用到达—定值信息计算ud链

(1) 若在基本块B中，某变量A的引用点u之前有A的定值点d，且A的定值点d能到达点u, 则A在u点的ud链为{d}；

(2) 若在基本块B中，某变量A的引用点u之前无A的定值点，则包含在IN[B]中的全部A的定值点均可到达点u, 所以in[B]中的这些A的定值点组成A在u点的ud链。

例：如图所示的流图的到达定值信息

in(B1)	Φ
out(B1)	1(b),2(c)
in(B2)	1(b),2(c),3(a),4(d),5(d),6(c),7(e)
out(B2)	1(b),2(c),3(a),4(d),6(c),7(e)
in(B3)	1(b),2(c),3(a),4(d),6(c),7(e),8(d),9(e)
out(B3)	1(b),2(c),3(a),5(d),6(c),7(e),9(e)
in(B4)	1(b),2(c),3(a),4(d),5(d),6(c),7(e),9(e)
out(B4)	1(b),3(a),4(d),5(d),6(c),7(e)
in(B5)	1(b),2(c),3(a),5(d),6(c),7(e),9(e)
out(B5)	1(b),2(c),3(a), 6(c), 8(d),9(e)
in(B6)	1(b),3(a),4(d),5(d),6(c),7(e)
out(B6)	3(a),4(d),5(d), 7(e), 10(b), 11(c)



$d_{3-(b)ud} = \{d_1\}$

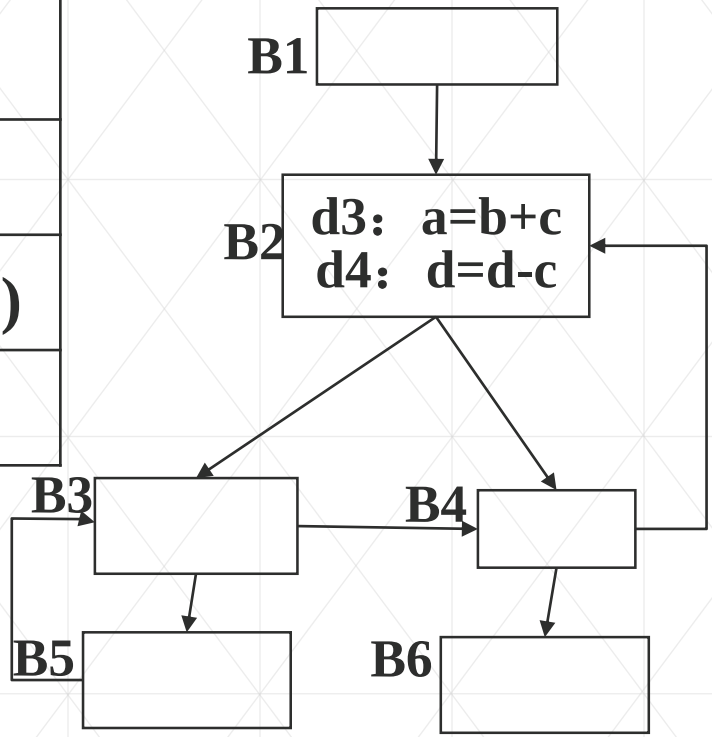
$d_{4-(a)ud} = \{d_3\}$

$d_{6-(c)ud} = \{d_2, d_6\}$

设置

下表为右图所示流图中部分基本块的到达定值信息，则变量d在d4点的ud链为

in(B1)	Φ
out(B1)	1(d),2(c)
in(B2)	1(d),2(c),3(a),4(d),5(d),6(c),7(e)
out(B2)	2(c),3(a),4(d),6(c),7(e)



A d1

B d2

C d3

D d4

E d5

F d6

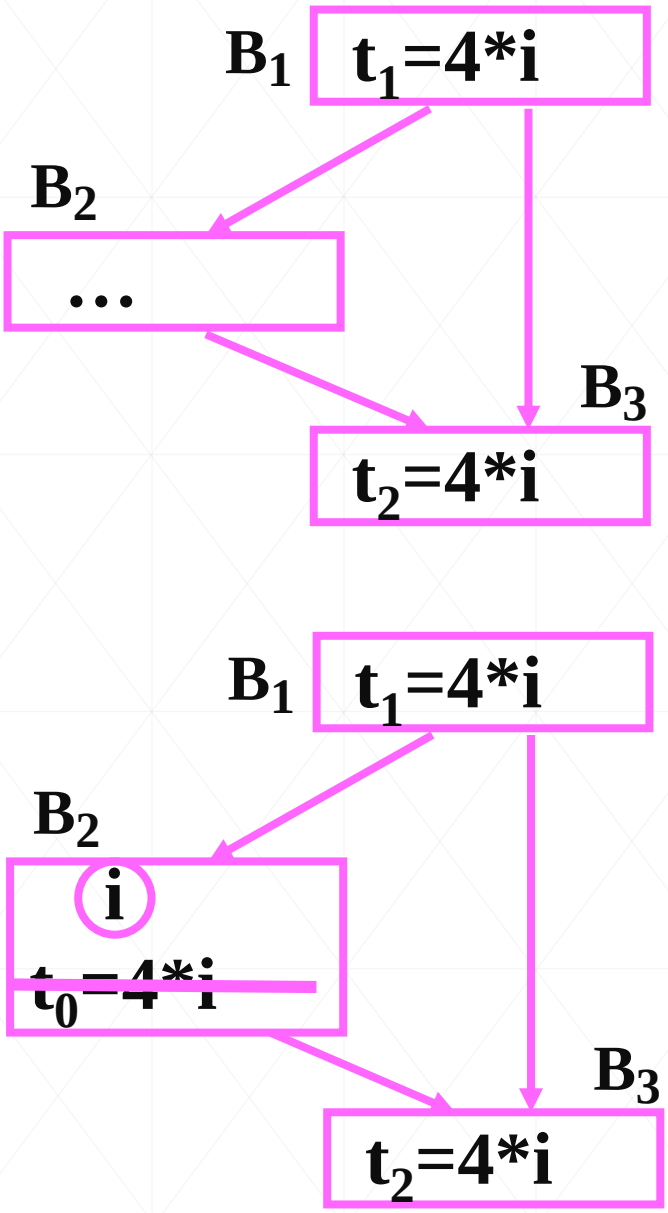
提交

可用表达式数据流方程

表达式x+y在点P可用：

如果从首结点到P的每条路径上都计算x+y，并且在最后一次计算x+y到P之间未对x或y定值，则表达式x+y在点P可用。

若有对x或y的定值，则可用的x+y被注销。



B_2中没有对变量i的定值，则B_1中的4*i 在B_3开始点可用。

B_2中对变量i定值后又计算4*i，则表达式4*i 在B_3开始点可用。

B_2中对变量i定值后不计算4*i，则表达式4*i 在B_3开始点不可用。

可用表达式数据流方程

数据流控制流方向一致

$$
\begin{cases}
out[B] = (in[B] - kill[B]) \cup gen[B] \\
in(B) = \bigcap_{P是B的前驱} out[P] & (B不是首基本块) \\
in(B_1) = \Phi & (B_1是首基本块)
\end{cases}
$$

** 与到达—定值数据流方程的区别：

(1)首基本块的处理特殊；

首基本块无任何表达式可用

(2) 算符∩；

一个表达式在块的开始点可用，

只有当它在该块的所有前趋块的出口可用时才行

活跃变量数据流方程

数据流控制流方向相反

$$\begin{cases} in_L(B) = (out_L(B) - def_L(B)) \cup use_L(B) \\ out_L(B) = \bigcup_{S \in Succ(B)} in_L(S) \end{cases}$$

$in_L(B)$：在基本块B入口点的活跃变量的集合。

$out_L(B)$：在基本块B出口点的活跃变量的集合。

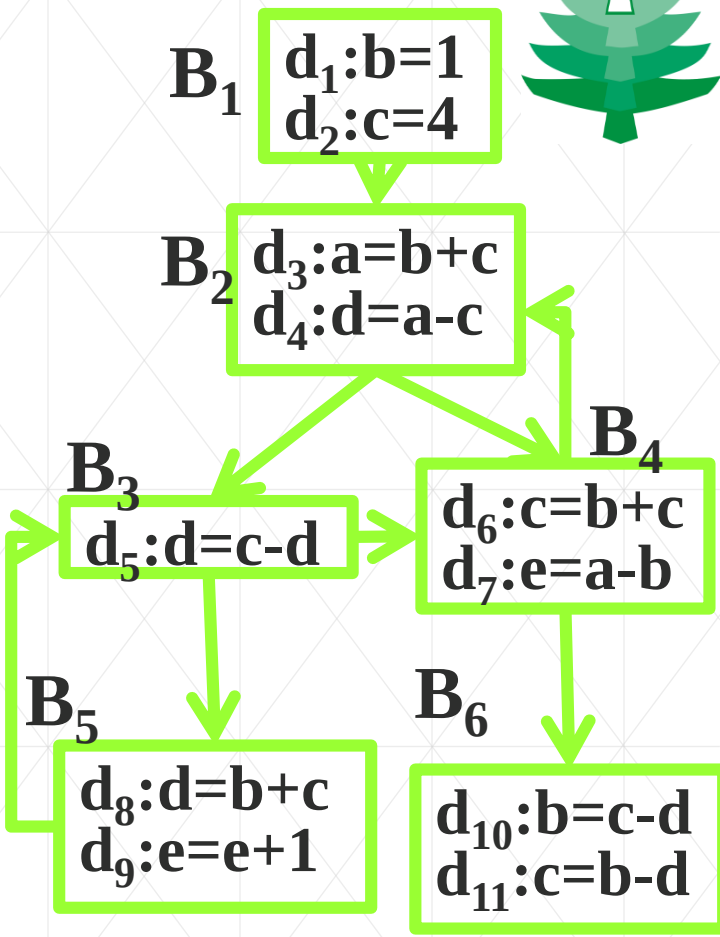
$def_L(B)$：在基本块B中定值的变量的集合。

$use_L(B)$：在基本块B中引用的，但在引用前未曾在B中定值的变量集。

Succ(B)：表示B的后继基本块。

例：如图所示的流图

	DEF	USE
B1	b,c	
B2	a,d	b,c
B3	d	c,d
B4	c,e	c,b,a
B5	d,e	b,c,e
B6	b,c	c,d



$def_L(B)$:在基本块B中定值的变量集合。

$use_L(B)$:在基本块B中引用的，但在引用前未曾在B中定值的变量集。

对$in_L(B)$，可由以下条件得到：

① 如果某变量a在$out_L(B)$中，而且变量a在B中未被重新定值，则a也在$in_L(B)$中；

② 如果变量a在$use_L(B)$中,则它一定在$in_L(B)$中；

③ 除以上两种情况外，没有其它变量a∈ $in_L(B)$。

对$out_L(B)$：

某变量a在基本块B的出口点是活跃变量，当且仅当a在B的某一后继基本块的入口点活跃。

即$out_L(B)$是B的所有后继基本块的in_L之并。

算法 （活跃变量）

输入：G及G中每个基本块B的$def_L[B]$和$use_L[B]$

输出：G中每个基本块B的$in_L[B]$和$out_L[B]$

```
{
 for (i=1;i<=N;i++)
     {
     in[Bi]= useL[Bi]; out[Bi]=Φ;          /* in[Bi]和out[Bi]的迭代初值 */
     }
     change="真"；/*change记录相继2次迭代所得的out[Bi]之值.不等则为"真"，
     while (change)  需要继续迭代；若相等，则迭代过程结束，其值为"假" */
            { change= "假"；
              for (i=N;i>=1;i--)
              { NEWOUT=∪in[S];     /*  S∈Succ(Bi) */
                if ( NEWOUT != out[Bi] ) /* NEWOUT记录每次迭代后OUT[Bi] 的
                   {  change= "真"；              新值  */
                      out[Bi]= NEWOUT;
                      in[Bi]= useL[Bi]∪(outL[Bi] - defL[Bi]);
                   }
              }
            }
}
```

包含活跃点的活跃变量数据流方程

$$\begin{cases} in_L(B) = (out_L(B) - def_L(B)) \cup use_L(B) \\ out_L(B) = \bigcup_{S \in Succ(B)} in_L(S) \end{cases}$$

$in_L(B)$：在基本块B入口点的活跃变量活跃点的集合。

$out_L(B)$：在基本块B出口点的活跃变量活跃点的集合。

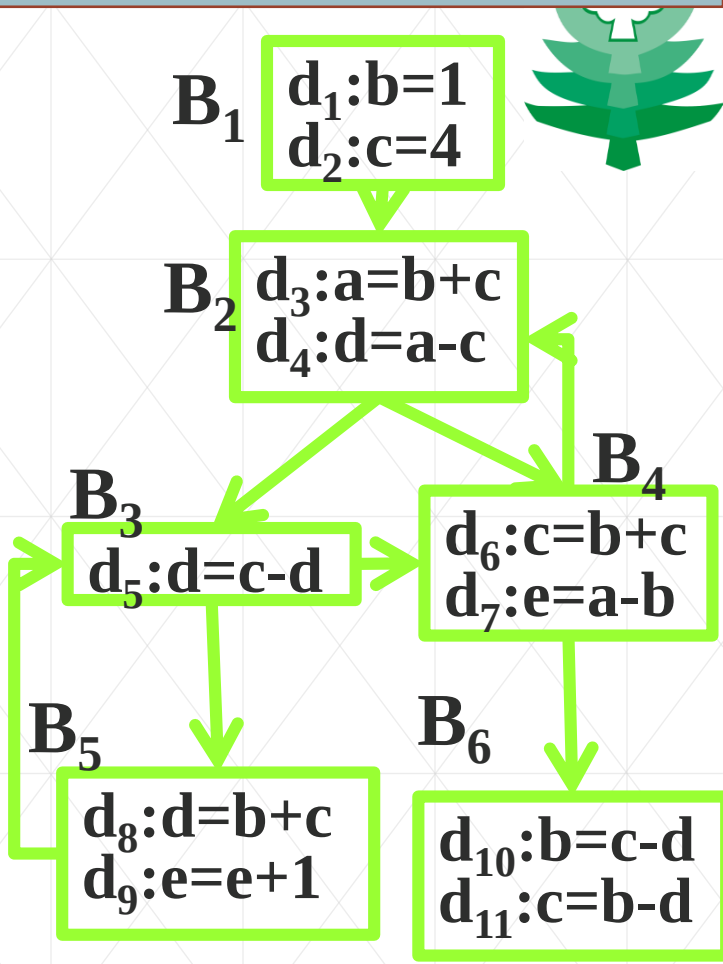
$def_L(B)$：在基本块B中定值的变量在其它基本块的引用点的集合。

$use_L(B)$：在基本块B中引用的，但在引用前未曾在B中定值的变量引用点的集合。

Succ(B)：表示B的后继基本块。

例：如图所示的流图

	DEF	USE
B1	b(3,6,7,8,11), c(3,4,5,6,8,10)	
B2	a(7),d(5,10,11)	b(3),c(3,4)
B3	d(10,11)	c(5),d(5)
B4	c(3,4,5,8,10), e(9)	c(6),b(6, 7), a(7)
B5	d(5,10,11)	b(8),c(8),e(9)
B6	b(3,6,7,8), c(3,4,5,6,8)	c(10),d(10,11)



$def_L(B)$：在基本块B中定值。点为块外引用点

$use_L(B)$：在基本块B中引用的，但在引用前未曾在B中定值的变量集。点为引用点。

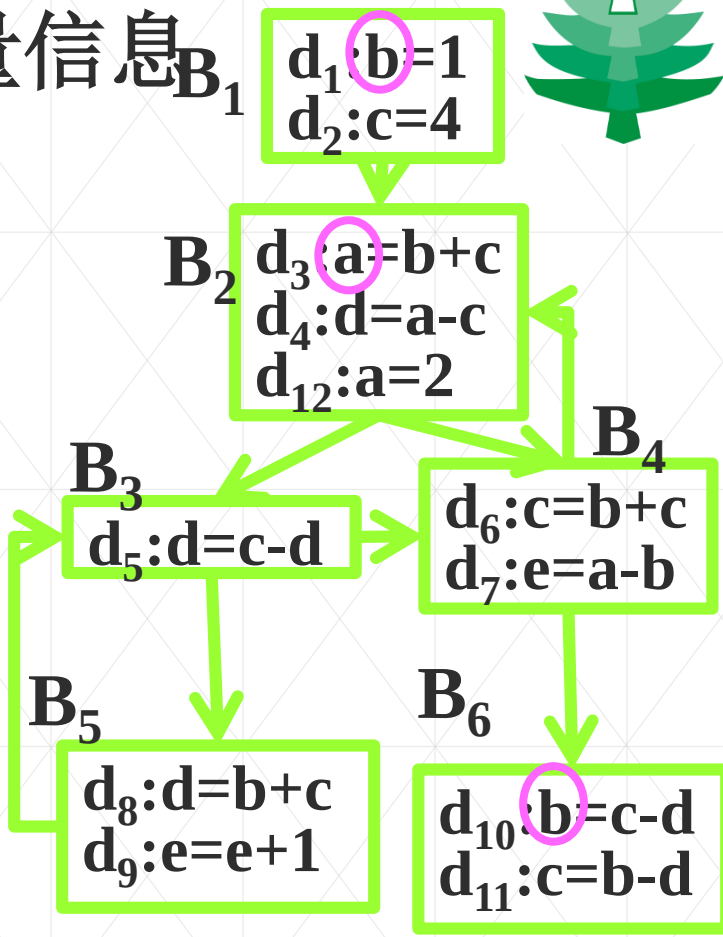
利用包含活跃点的活跃变量信息计算du链

(1)若在基本块B中，某变量A的定值点d之后有A的定值点p，则从点d到与d相距最近的那个A的定值点p之间的对A的所有引用点，即为A在定值点d的du链。

(2)若在基本块B中，某变量A的定值点d之后无A的定值点，则B中点d之后的A的所有引用点加上$OUT_L(B)$中变量A的所有活跃点即为A在点d的du链。

例：如图所示的流图的活跃变量信息

in(B1)	e(9)
out(B1)	b(3,6,7,8),c(3,4,5,6,8), e(9)
in(B2)	b(3,6,7,8),c(3,4,5,6,8), e(9)
out(B2)	c(5,6,8),d(5,10,11), b(3,6,7,8), a(7), e(9)
in(B3)	c(5,6,8),d(5), b(3,6,7,8), a(7), e(9)
out(B3)	c(5,6,8),b(3,6,7,8),a(7), d(10,11), e(9)
in(B4)	c(5,6),b(3,6,7,8),a(7), d(10,11)
out(B4)	b(3,6,7,8),c(3,4,5,6,8,10),d(10,11), e(9)
in(B5)	b(3,6,7,8),c(5,6,8),e(9), a(7)
out(B5)	c(5,6,8),d(5), b(3,6,7,8), a(7), e(9)
in(B6)	c(10),d(10,11)
out(B6)	∅



$d_{3-(a)du}=\{d_4\}$

$d_{1-(b)du}=\{d_3,d_6,d_7,d_8\}$

$d_{10-(b)du}=\{d_{11}\}$

第 8 章 代码优化（optimization）

循环优化准备

1. 循环查找；
2. 涉及循环的所有基本块的数据流分析：

量的定值——引用情况信息

ud链　　du链、活跃变量

可实施的循环优化：
- 代码外提 (频度削弱)
- 强度削弱
- 删除归纳变量

- 循环的前置结点

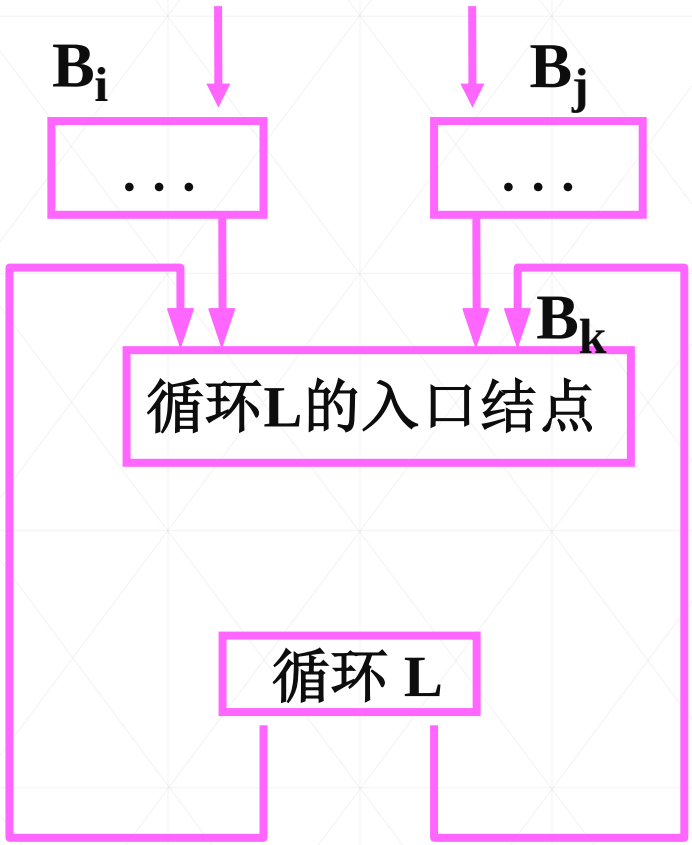
在循环的入口结点前加的一个新结点(基本块)
循环的入口结点为其惟一后继，
原程序流图中从循环外到循环入口结点的有向边修改为到循环前置结点。

∵ 循环的入口惟一

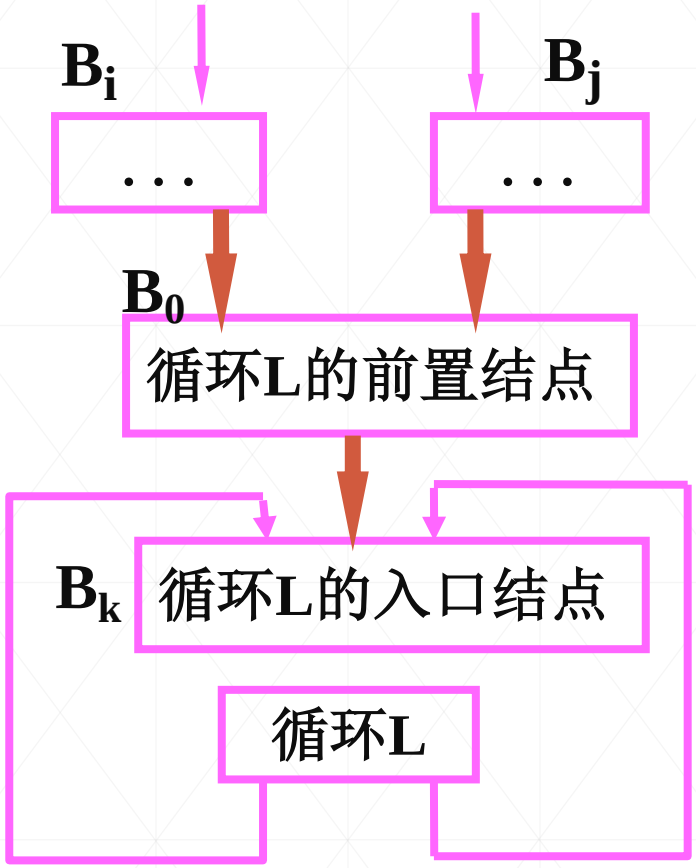
∴ 前置结点惟一

应用

例：设有流图



建立循环L的
前置结点B_0前的流图



建立循环L的
前置结点B_0后的流图

一．代码外提

将循环中的不变运算提到循环的前置结点中。

不变运算:与循环执行次数无关的运算或

不受循环控制变量影响的那些运算。

例如:

循环L中有语句A=B op C

B和C是常数，

B和C虽然是变量，但到达B和C的定值点皆在循环L外，

则在循环中每次计算出的B op C的值始终不变。

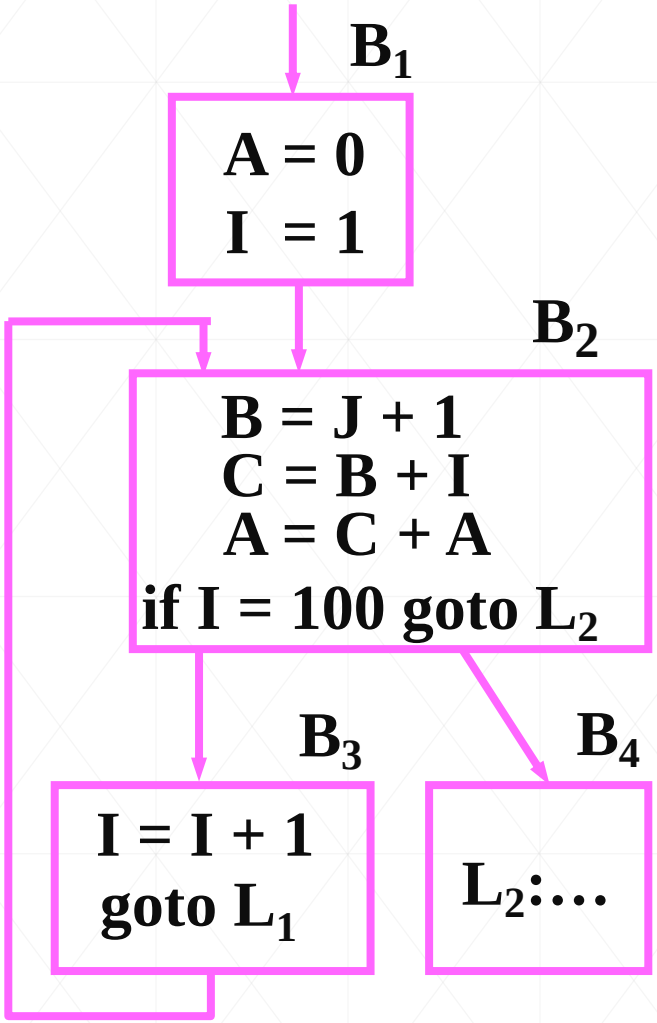
例：给出以下源程序

循环中：

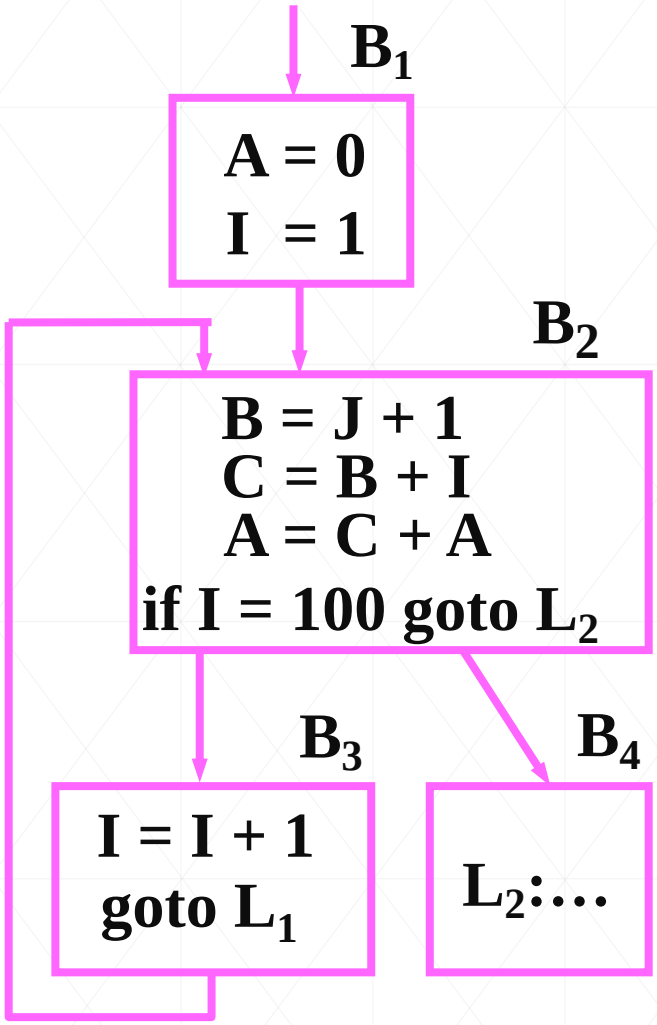
基本块B_2中的语句

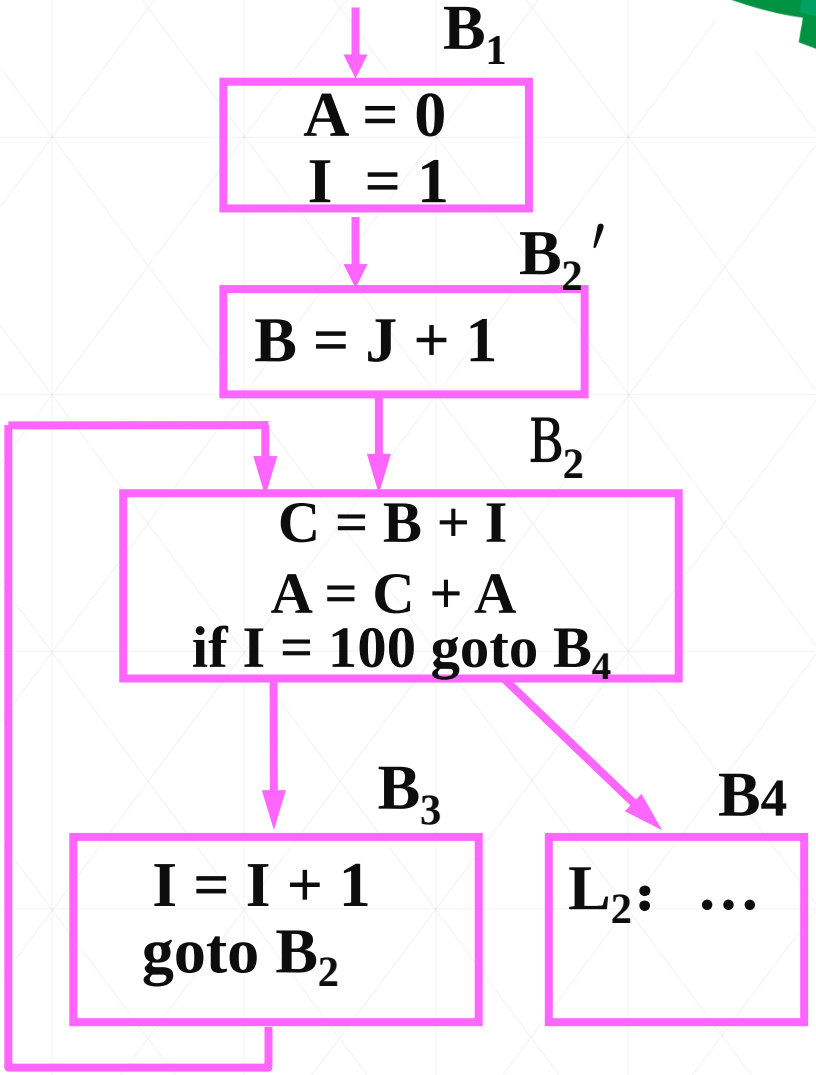
$$B = J+1,$$

循环中没有对J的定值点，对J引用的定值点都在循环外，所以它是循环的不变运算，可提到循环的前置结点B_2'中。



$<B_3, B_2>$ loop $= \{ B_2, B_3 \}$


B1
A = 0
I = 1
B2
B = J + 1
C = B + I
A = C + A
if I = 100 goto L2
B3
I = I + 1
goto L1
B4
L2:…


B1
A = 0
I = 1
B2′
B = J + 1
B2
C = B + I
A = C + A
if I = 100 goto B4
B3
I = I + 1
goto B2
B4
L2: …

- 代码外提算法的设计

(1) 查找循环中的不变运算； （X1）

(2) 实施代码外提； （X2）

- 算法 （ X1:查找循环中不变运算）

设有循环L

输入： 循环L；L中的所有变量引用点的ud链信息；

输出： 标识了所有“不变运算”的循环L；

方法：

(1)查看L中各基本块的每个语句，如果其中的每个运算对象为常数或定值点在L外（据ud链判断），将该语句标记为“不变运算”；

(2)重复第(3)步，直至没有新的语句被标记为不变运算”为止；

(3)依次查看未被标记为“不变运算”的语句，如果其运算对象为常数

或定值点在L外，

或只有一个到达-定值点，但该点上的语句已被标记为“不变运算”，

则将被查看的语句标为“不变运算” 。

- 算法 （X2: 代码外提）

输入： 循环L；ud链信息和必经结点$D(n_i)$信息；活跃变量信息

输出： L'；(加前置块，已经外提“不变运算”后的循环L)

方法：

(1) 求出循环L中所有不变运算。(call X1)

(2) 对(1)求出的每一不变运算

d：A=B op C 或 A=op B 或 A=B，

检查是否满足如下条件之一：

①(i)点d所在的结点是L的所有出口结点的必经结点；

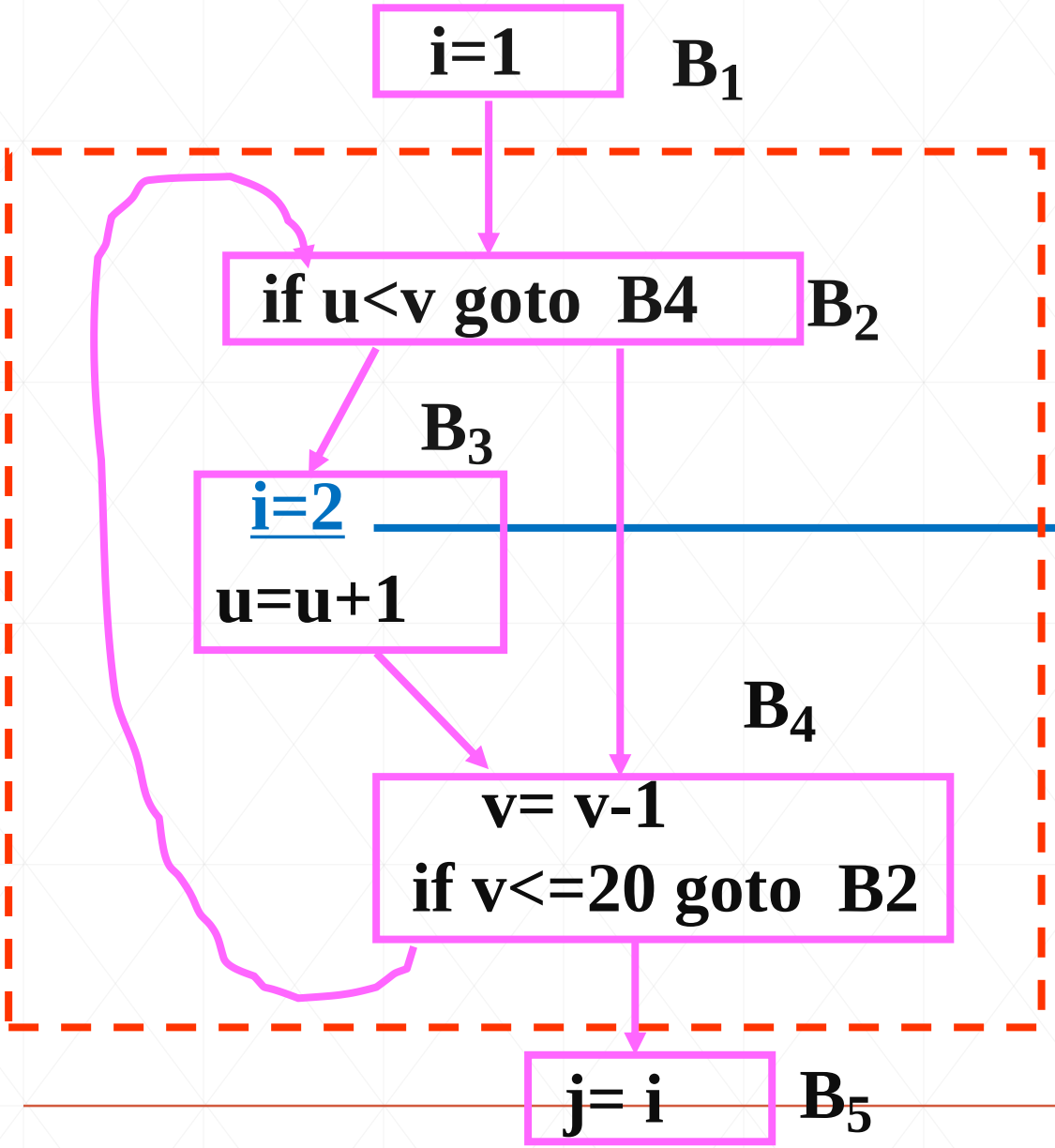
(ii)A在L中其它地方未再定值；

(iii)L中所有A的引用点只有点d对A的定值才能到达。

② A在离开L后不再是活跃的(A在L的任何出口结点的出口处不是活跃的)，且条件①的(ii)和(iii)成立。

(3) 按第 (1)步找出的不变运算的顺序，依次把符合(2)的条件之一的不变运算外提到L的前置结点中。若点d的运算对象(B或C)是在L中定值的，那么，只有当这些定值语句都提到前置结点中后，才可把点d的运算外提。

说明外提的限制条件：



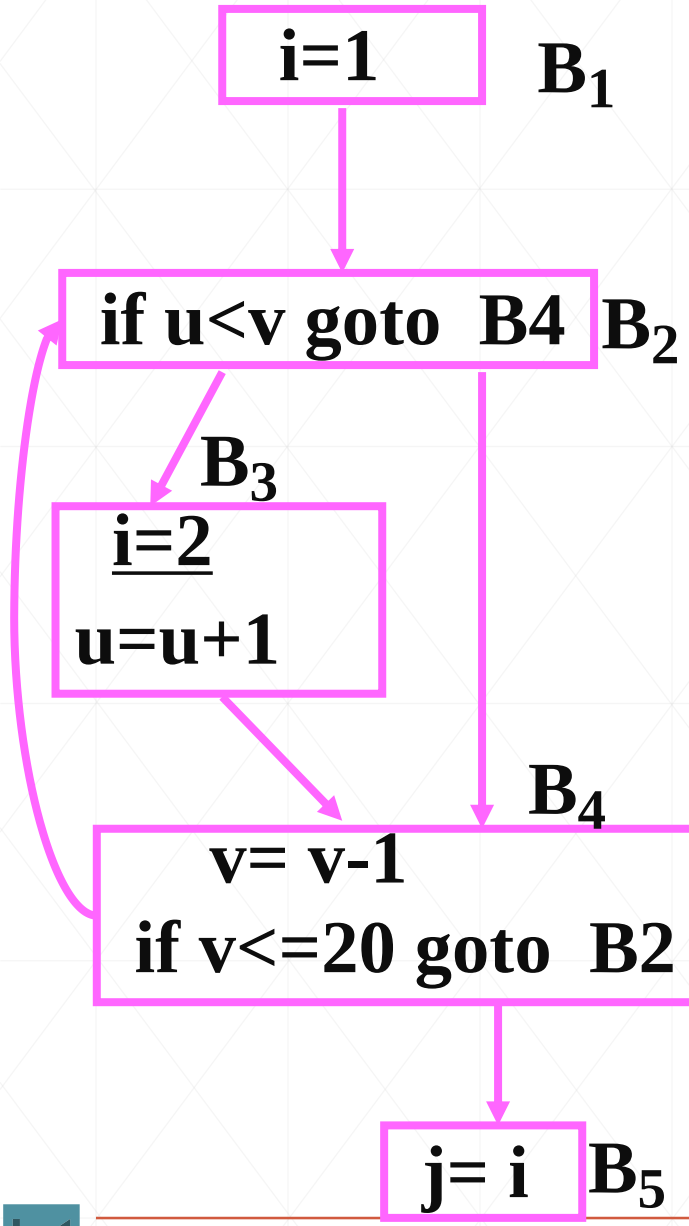
关于条件(1)中的(i)的举例：

点d所在的结点是L的所有出口结点的必经结点；

不变运算

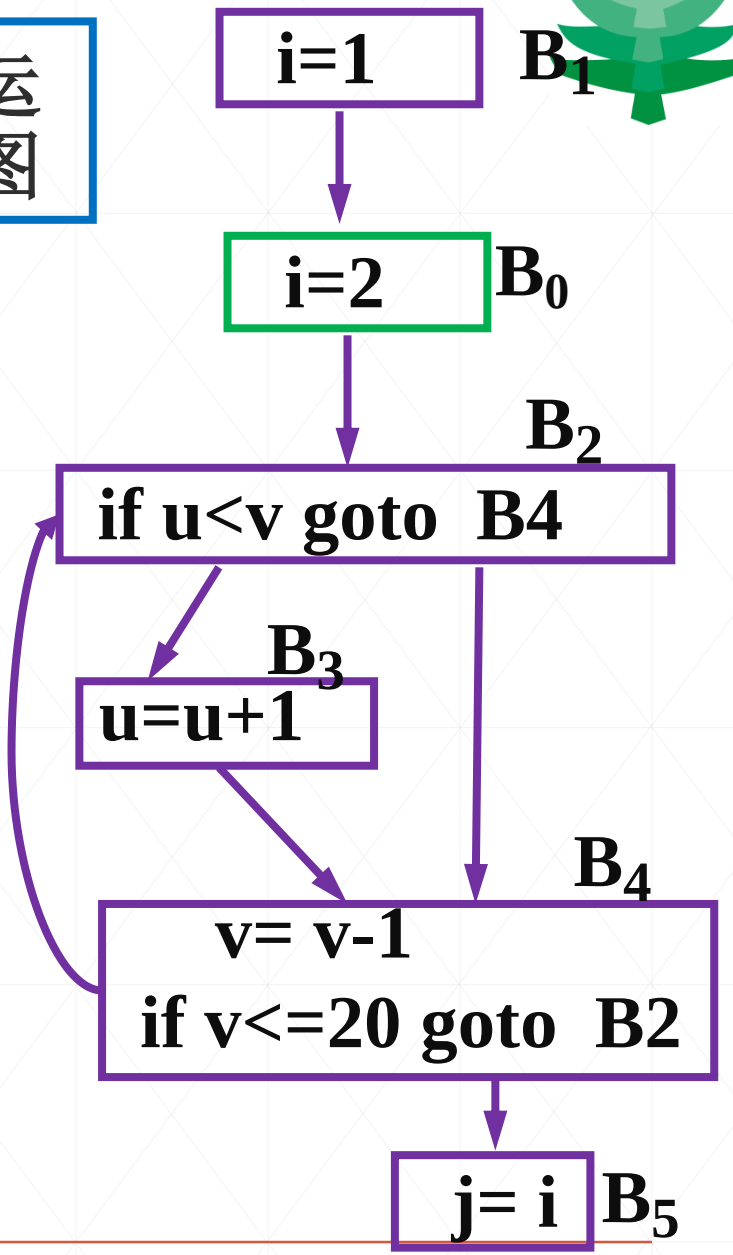
$<B4,B2>_{Loop}$

$=\{ B2, B3, B4 \}$



外提不变运算后的流图

将“不变运算”外提后，会改变程序的计算结果。



说明外提的限制条件：

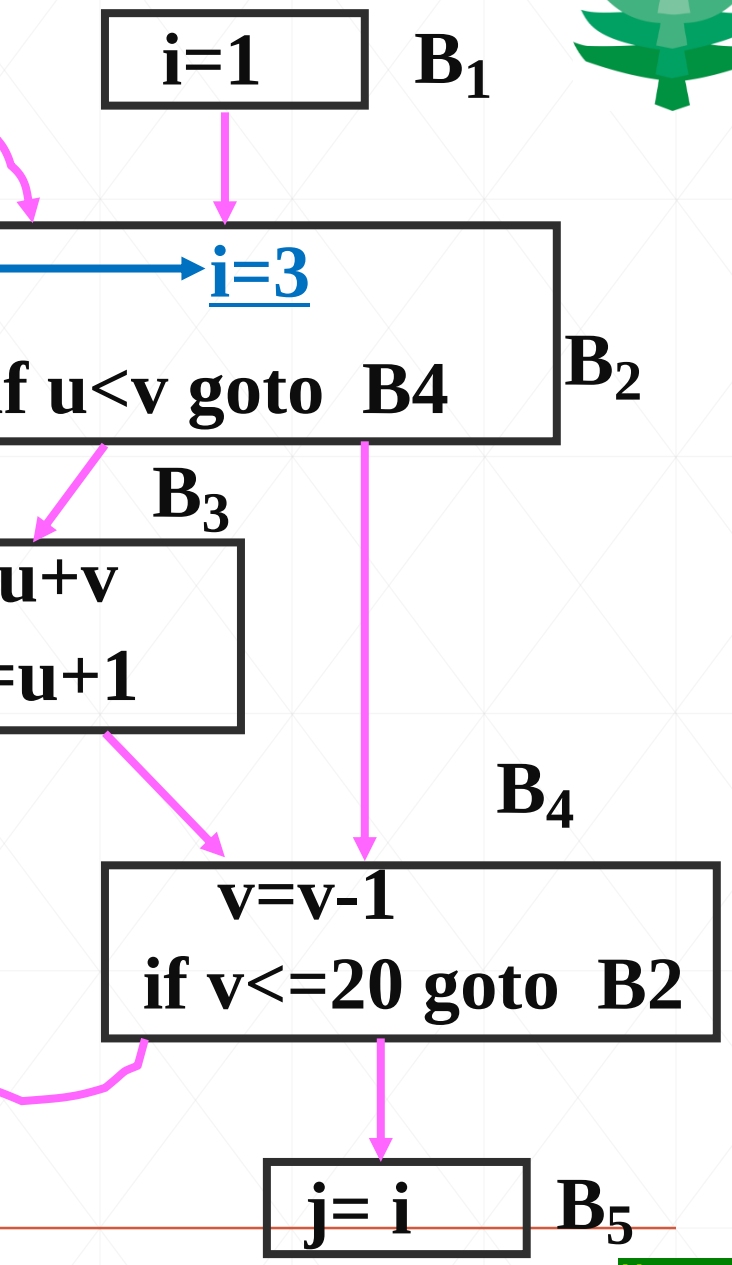
不变运算

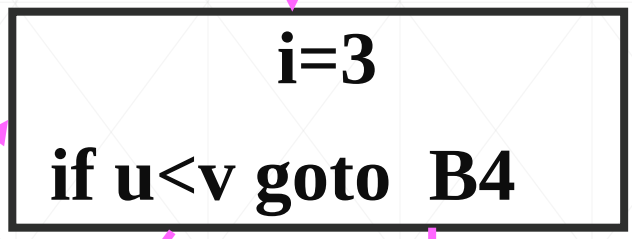
条件(i)不能

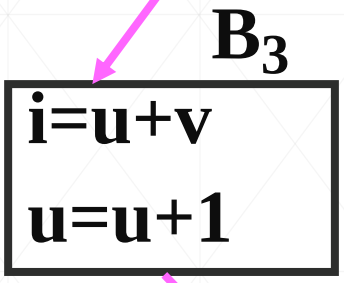
阻挡将i=3外提

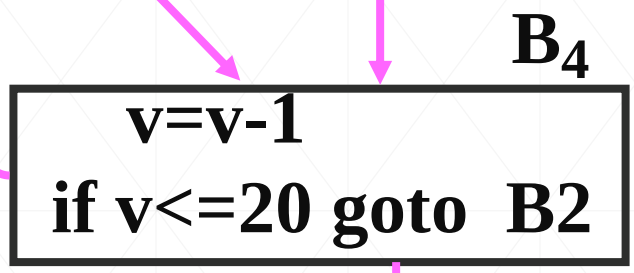
关于条件(1)中的(ii)的举例：

A在L中其它地方未再定值；



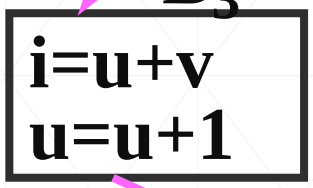


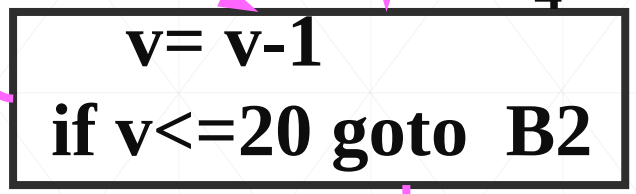


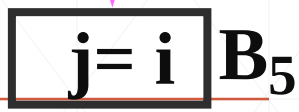


外提不变运算后的流图

将“不变运算”外提后，会改变程序的计算结果。



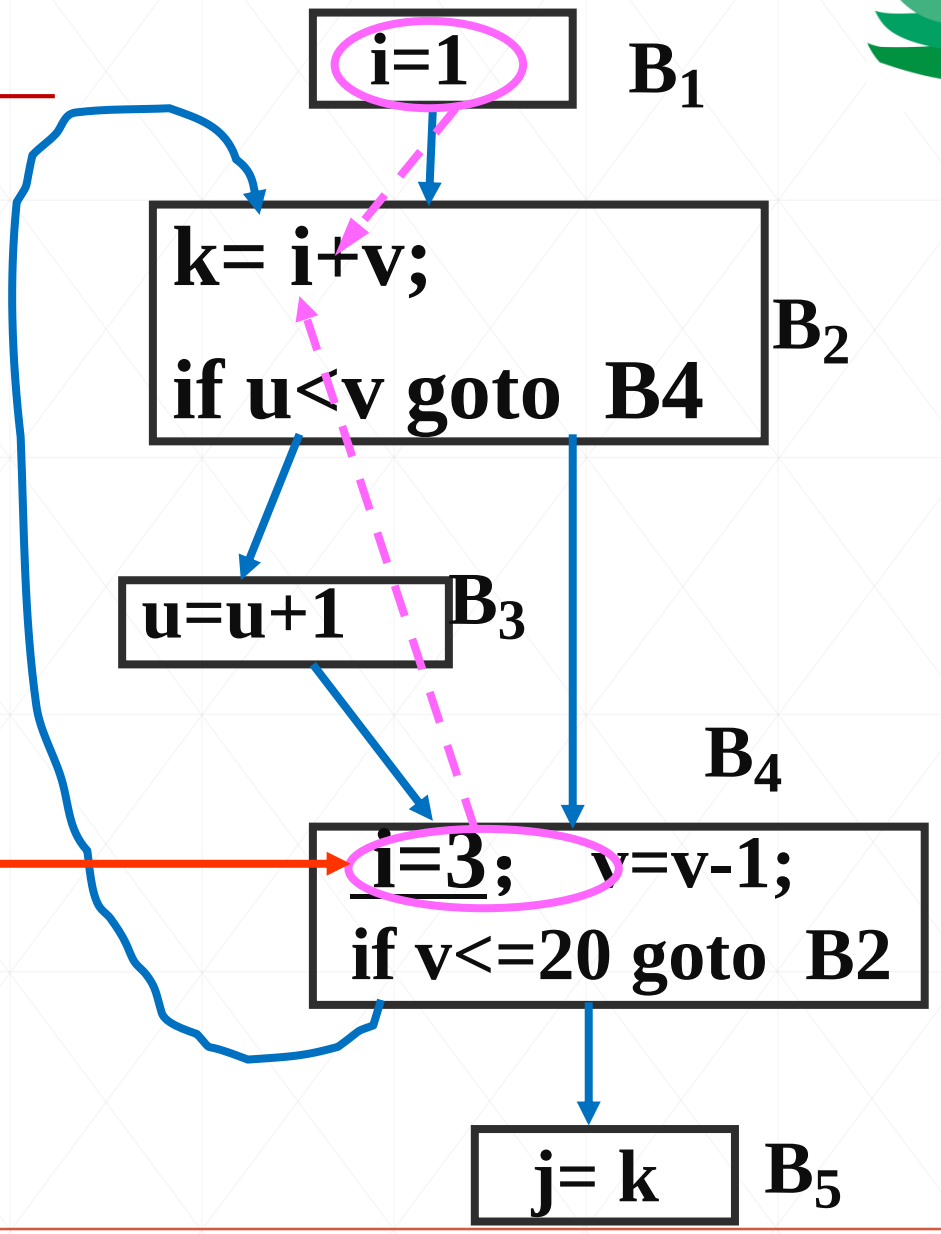




说明外提的限制条件：

关于条件(1)中的(iii)的举例：

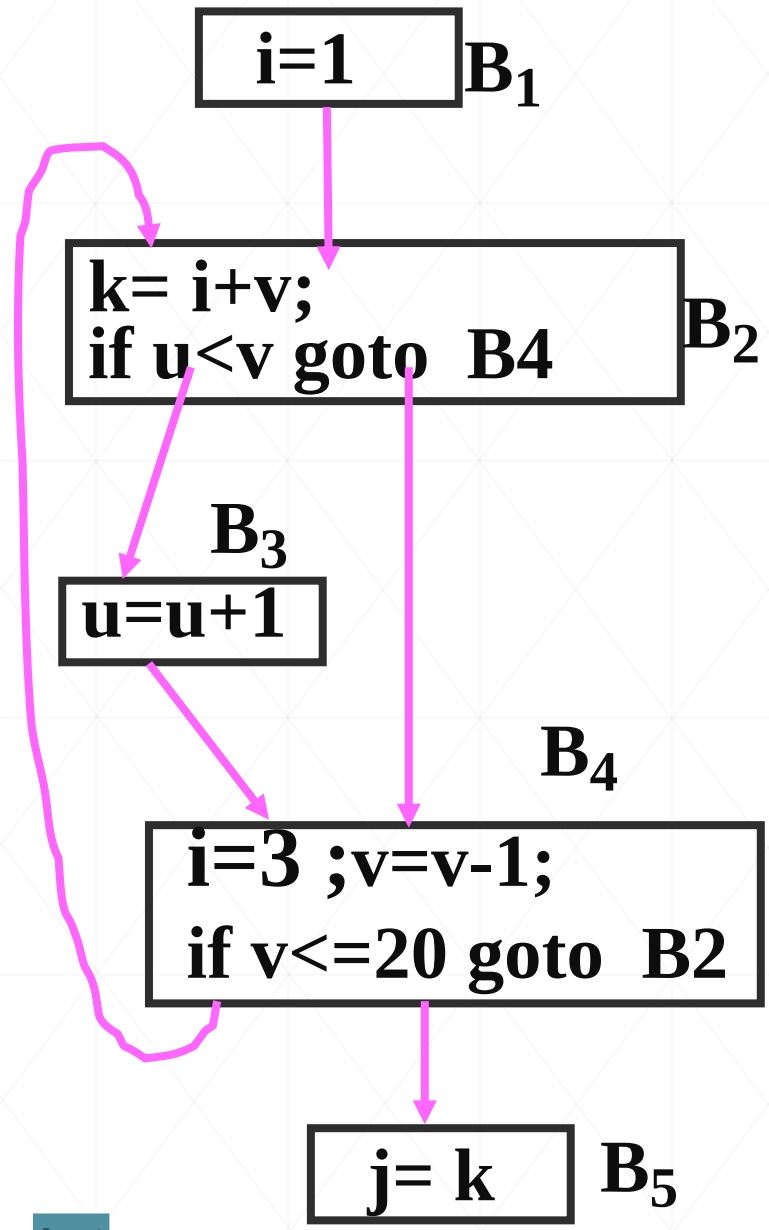
L中所有A的引用点只有点d对A的定值才能到达；



不变运算

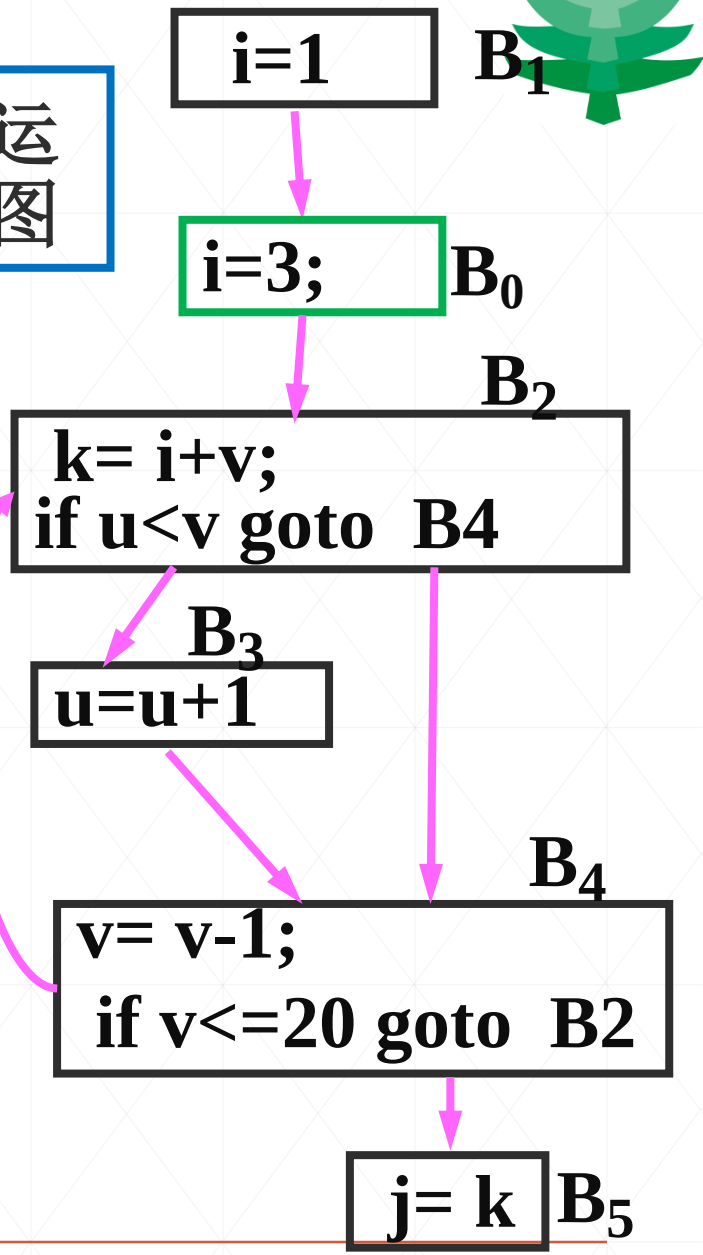
条件(i)和(ii)都不能阻挡将i=3外提

返回



外提不变运算后的流图

将“不变运算”外提后，会改变程序的计算结果。



- 算法 （X2: 代码外提）

输入： 循环L；ud链信息和必经结点$D(n_i)$信息；活跃变量信息

输出： L'；(加前置块，已经外提“不变运算”后的循环L)

方法：

(1) 求出循环L中所有不变运算。(call X1)

(2) 对(1)求出的每一不变运算

d：A=B op C 或 A=op B 或 A=B，

检查是否满足如下条件之一：

①(i)点d所在的结点是L的所有出口结点的必经结点；

(ii)A在L中其它地方未再定值；

(iii)L中所有A的引用点只有点d对A的定值才能到达。

② A在离开L后不再是活跃的(A在L的任何出口结点的出口处不是活跃的)，且条件①的(ii)和(iii)成立。

(3) 按第 (1)步找出的不变运算的顺序，依次把符合(2)的条件之一的不变运算外提到L的前置结点中。若点d的运算对象(B或C)是在L中定值的，那么，只有当这些定值语句都提到前置结点中后，才可把点d的运算外提。

二．强度削弱与删除归纳变量

强度削弱是将程序中强度高的运算使用强度低的运算替代，以便使程序运行时间缩短。

▲ 一般情况

循环L中存在 $I=I\pm C_1$

且L中存在 $T=K*I\pm C_2$

呈线性函数

（T是归纳变量

I是基本归纳变量）

求出递增(减)量K1，用±替代*

$T = T \pm K1$

- 定义（基本归纳变量/归纳变量）

如果循环中变量I仅有惟一的 I=I±C 形式的赋值，其中C为循环不变量，则称I为循环中基本归纳变量。

如果I是循环中一基本归纳变量，变量J在循环中的定值总可化为I的某一线性函数的形式：$J= C_1*I±C_2$，其中C_1，C_2是循环不变量，则称J是归纳变量，并称J与I同族。

循环优化中强度削弱和删除归纳变量

有次序且相关

(寻找归纳变量) (强度削弱) (删除归纳变量)

算法 （W1：查找归纳变量）

输入：带有到达—定值信息和循环不变运算信息的循环L

输出：查找循环L中的一组归纳变量

方法：

step1：扫描L，找出所有基本归纳变量；（I=I±C）

step2：寻找L中只有一个定值的K(归纳变量)，其形式为：

K=J*C； K=C*J； K=J/C； K=J ± C； K=C ± J

(其中：C为循环不变量；J为基本归纳变量或归纳变量；)

1) 若J是基本归纳变量，K在J族中；{K、J同族}

2) 若J是归纳变量，J∈I族， K、J、I同族的附加要求：

a) 在L中对J的惟一定值和对K的定值间没有对I的定值；

b) L外没有J的定值可到达K；

找出一族归纳变量，可以变换计算归纳变量的指令 (*→+)

算法（W2：强度削弱）

输入：带有到达—定值信息的L和归纳变量族

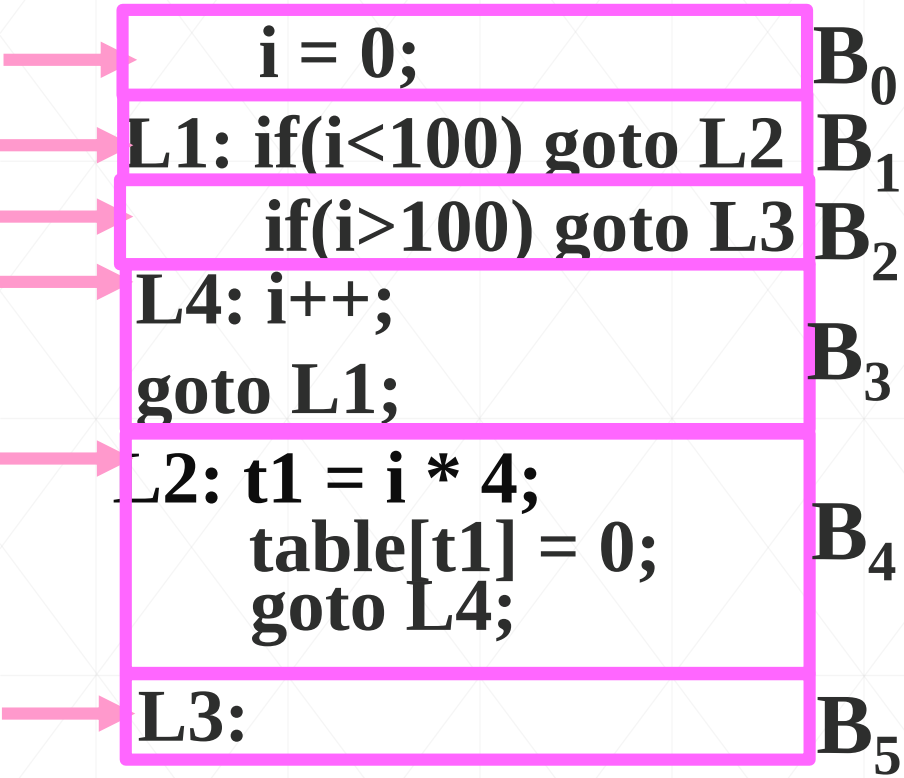
输出：进行强度削弱优化后的L

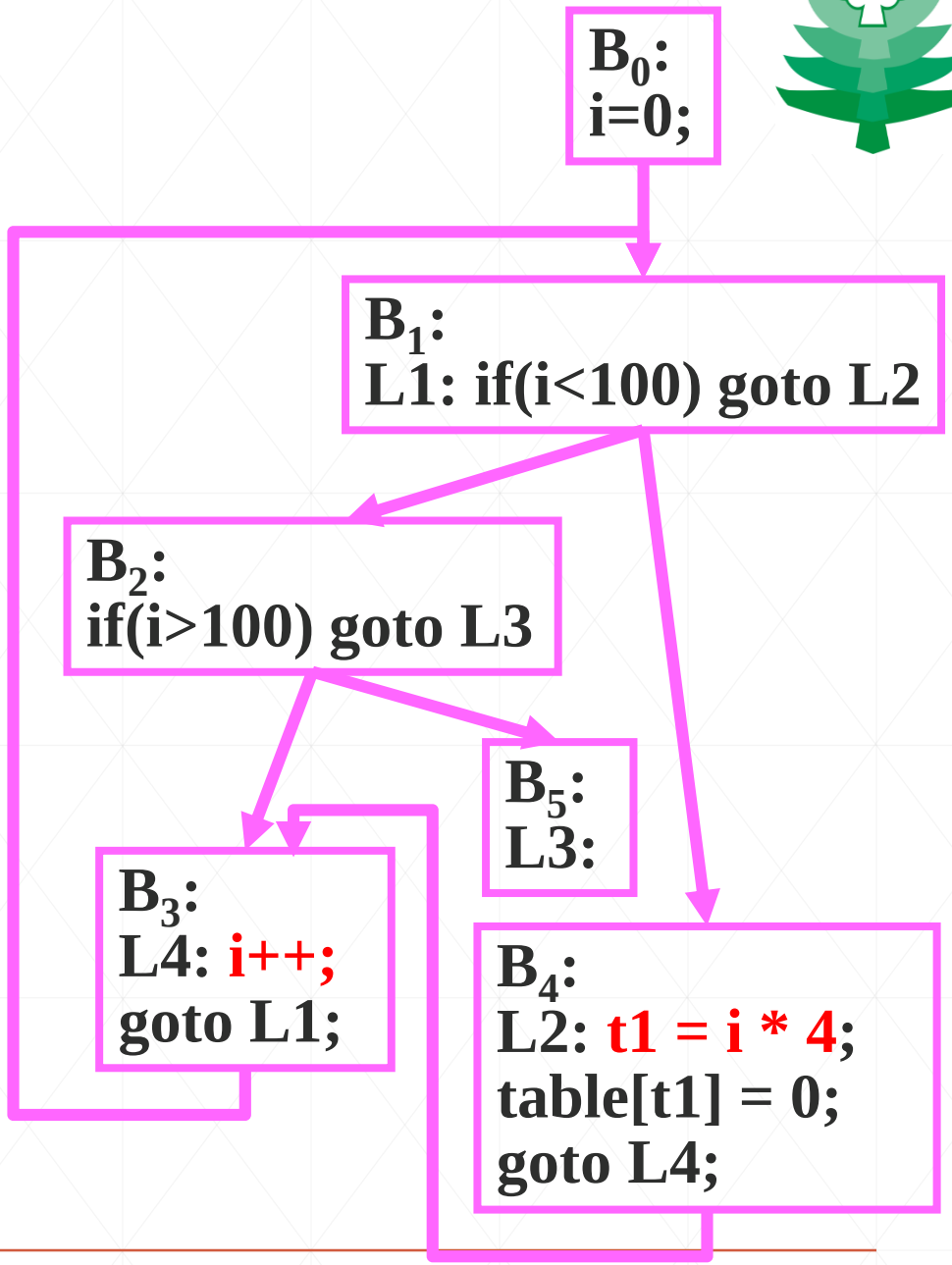
方法：依次考察基本归纳变量I，对每个形如J=C*I ±d的四元式：

step1: 在L中每个I=I+n(n为常量)的四元式后加上J=J+C*n；

step2: 在循环的前置基本块中添加J=C*I ± d；

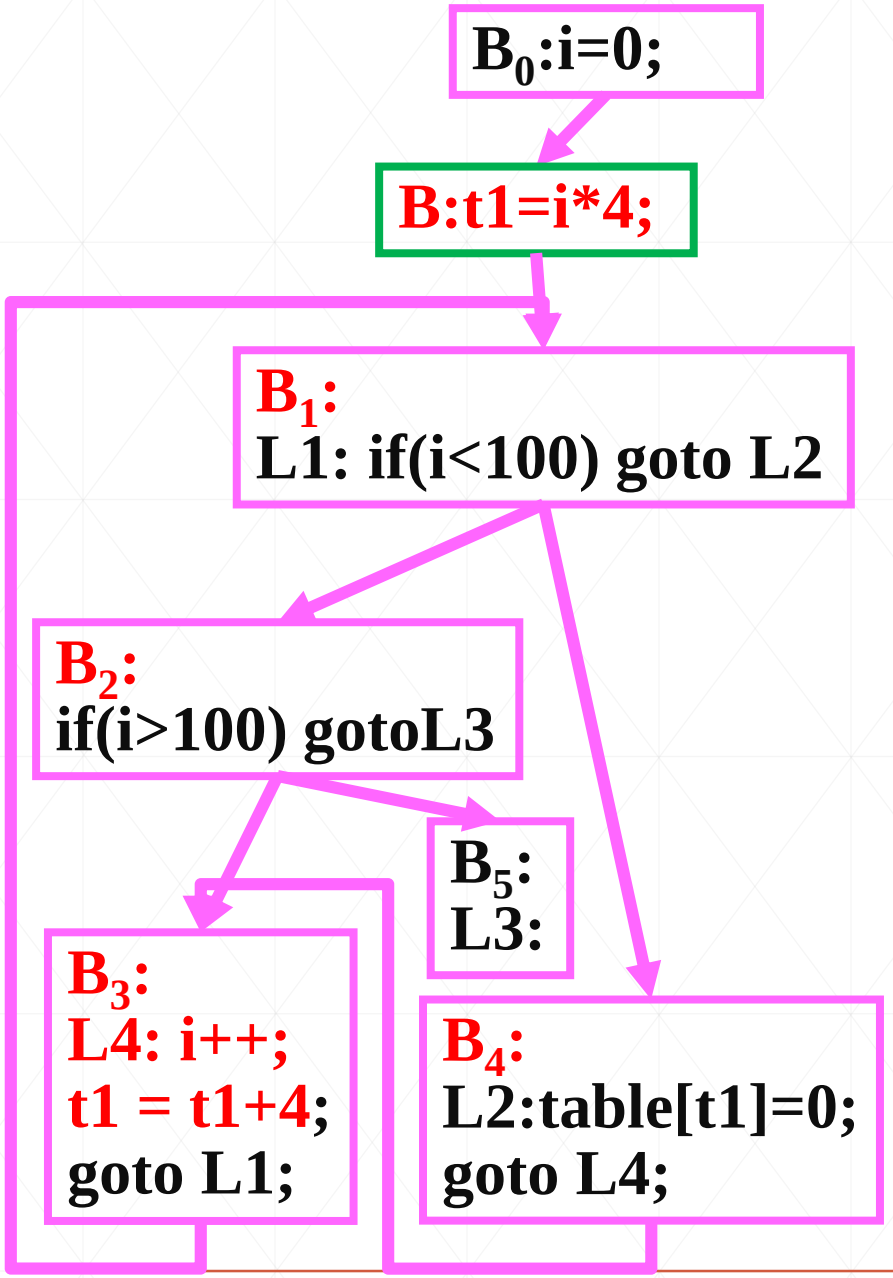
step3: 删除原来的J=C*I ±d

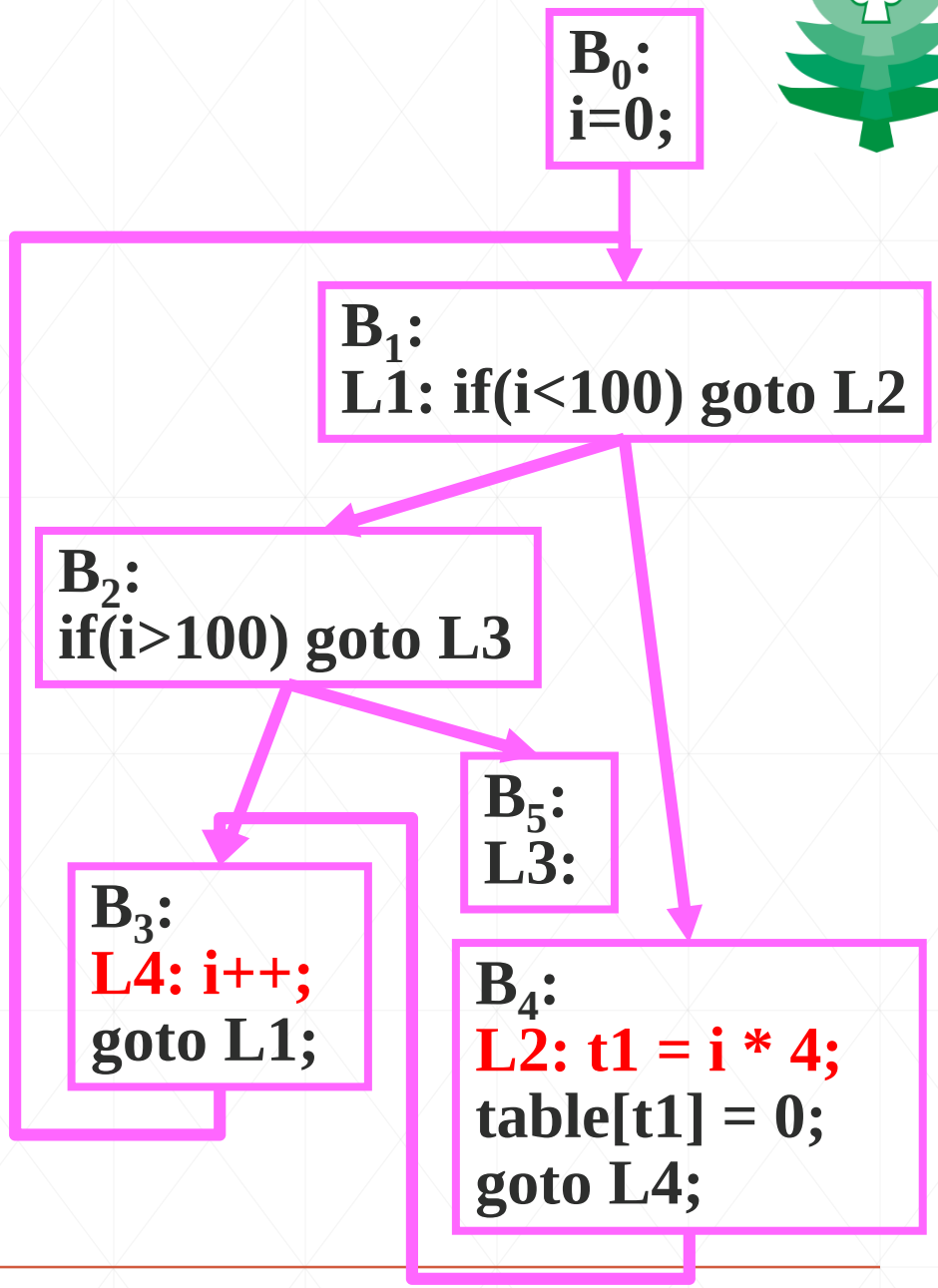




<B3, B1> loop

= { B1, B3, B2, B4}


B0:i=0;
B:t1=i*4;
B1:
L1: if(i<100) goto L2
B2:
if(i>100) gotoL3
B3:
L4: i++;
t1 = t1+4;
goto L1;
B5:
L3:
B4:
L2:table[t1]=0;
goto L4;


B0:
i=0;
B1:
L1: if(i<100) goto L2
B2:
if(i>100) goto L3
B3:
L4: i++;
goto L1;
B5:
L3:
B4:
L2: t1 = i * 4;
table[t1] = 0;
goto L4;

算法 （W3：删除归纳变量）

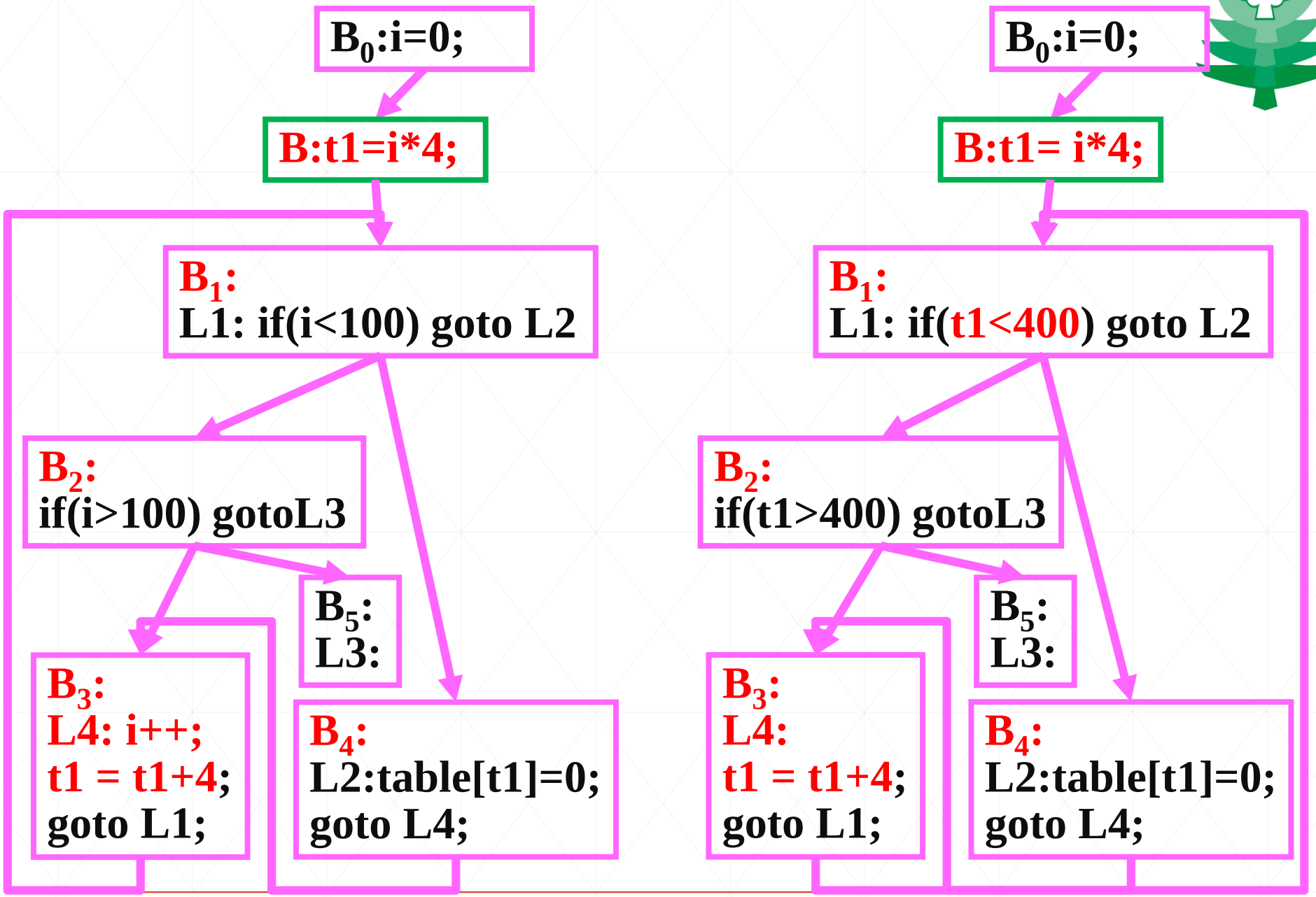
输入：带有到达—定值信息、循环不变运算信息和活跃变量信息的L

输出：删除归纳变量优化后的L

方法： 考察每个仅用于计算同族中其它归纳变量和条件分支的归纳变量I，取I族的一个归纳变量J，将含I的测试改为用J代替。

据du链信息

替代后的I不再引用时，从L中删去对I定值的语句


B0:i=0;
B:t1=i*4;
B1:
L1: if(i<100) goto L2
B2:
if(i>100) gotoL3
B5:
L3:
B3:
L4: i++;
t1 = t1+4;
goto L1;
B4:
L2:table[t1]=0;
goto L4;
B0:i=0;
B:t1= i*4;
B1:
L1: if(t1<400) goto L2
B2:
if(t1>400) gotoL3
B5:
L3:
B3:
L4:
t1 = t1+4;
goto L1;
B4:
L2:table[t1]=0;
goto L4;

例： P247 — 例8.6

设计算大小为20的两个向量(一维数组表示)a与b的内积公式为

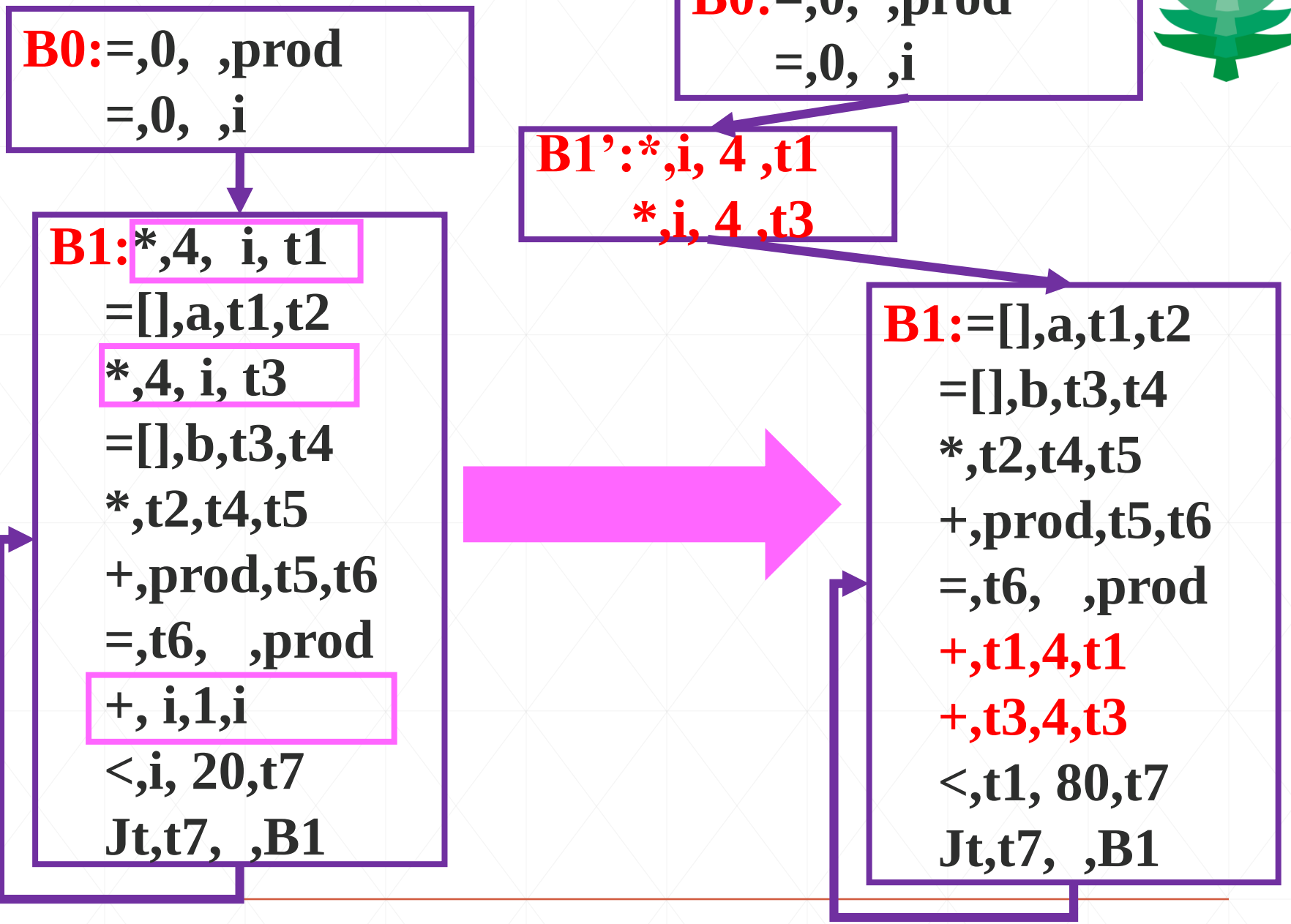
prod=a1×b1+a2×b2+ … +a20×b20

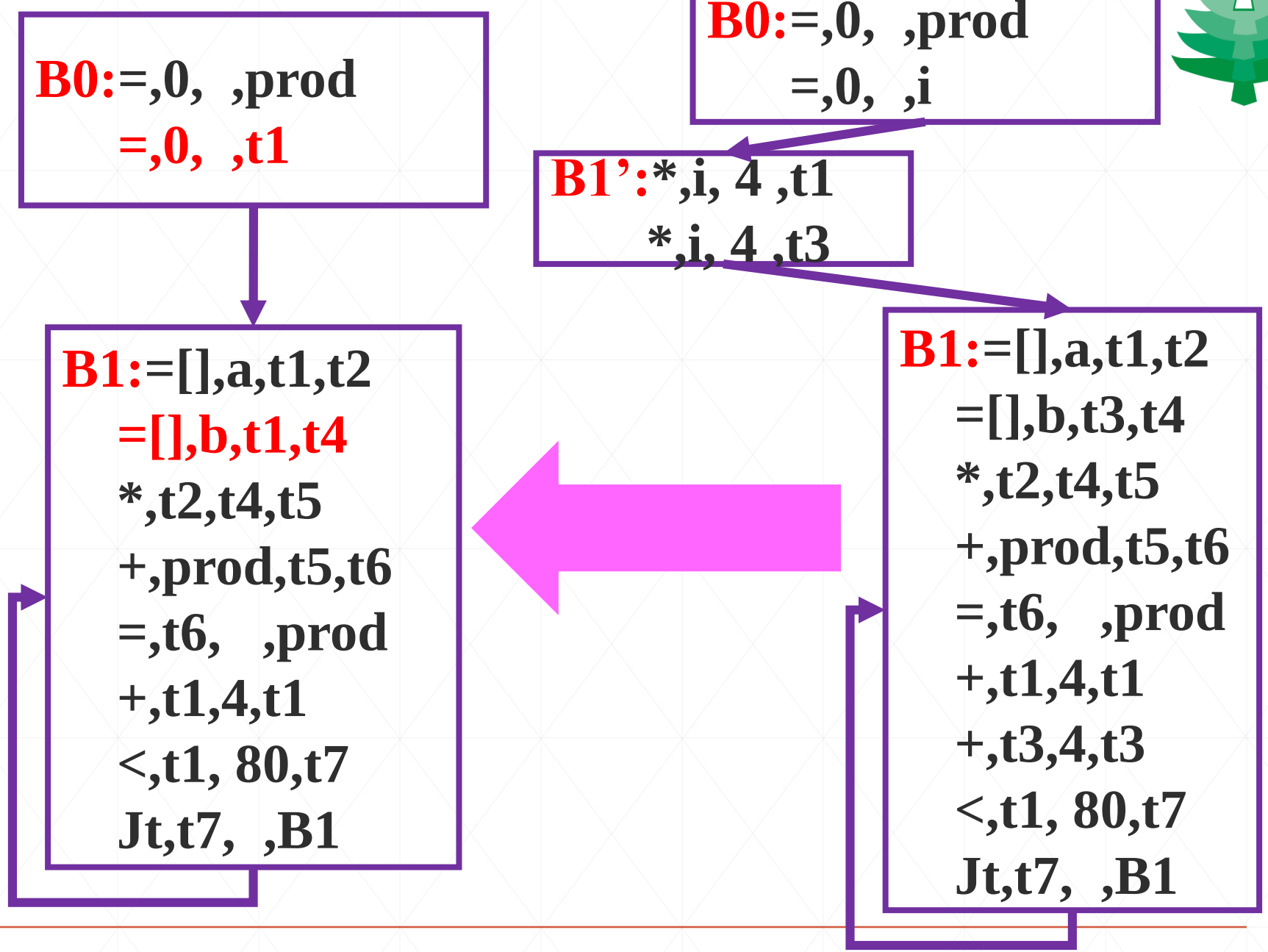
实现计算的源程序片段如下：

```
prod=0;  i=0;
do{
    prod=prod+a[i]*b[i];
    i++;}
while(i>20)
```

```
1.=,0,  ,prod
2.=,0,  ,i        B0
3.*,4,  i, t1
4.=[],a,t1,t2
5.*,4, i, t3
6.=[],b,t3,t4
7.*,t2,t4,t5
8.+,prod,t5,t6
9.=,t6,   ,prod
10.+, i,1,i
11.<,i, 20,t7
12.Jt,t7,  ,3   B1
```

```
prod=0;  i=0;
do{
prod=prod+a[i]*b[i];
i++;}
while(i<20)
```


B0:=,0, ,prod
=,0, ,i
B1:*,4, i, t1
=[],a,t1,t2
*,4, i, t3
=[],b,t3,t4
*,t2,t4,t5
+,prod,t5,t6
=,t6, ,prod
+, i,1,i
<,i, 20,t7
Jt,t7, ,B1
B0:=,0, ,prod
=,0, ,i
B1':*,i, 4 ,t1
*,i, 4 ,t3
B1:=[],a,t1,t2
=[],b,t3,t4
*,t2,t4,t5
+,prod,t5,t6
=,t6, ,prod
+,t1,4,t1
+,t3,4,t3
<,t1, 80,t7
Jt,t7, ,B1


B0:=,0, ,prod
=,0, ,t1
B1:=[],a,t1,t2
=[],b,t1,t4
*,t2,t4,t5
+,prod,t5,t6
=,t6, ,prod
+,t1,4,t1
<,t1, 80,t7
Jt,t7, ,B1
B0:=,0, ,prod
=,0, ,i
B1':*,i, 4 ,t1
*,i, 4 ,t3
B1:=[],a,t1,t2
=[],b,t3,t4
*,t2,t4,t5
+,prod,t5,t6
=,t6, ,prod
+,t1,4,t1
+,t3,4,t3
<,t1, 80,t7
Jt,t7, ,B1

实施优化的综合考虑

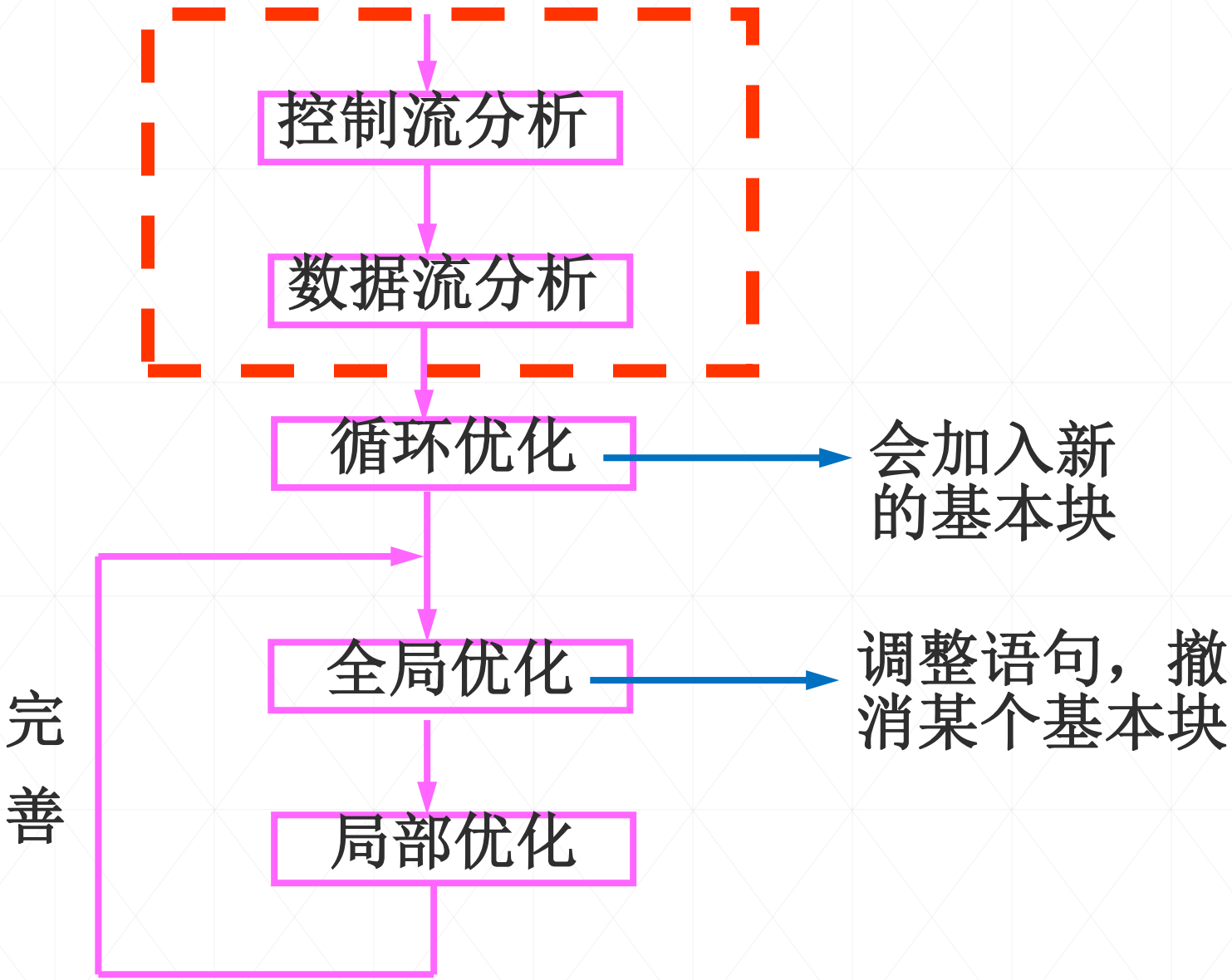
综合应用各类优化技术的共性应考虑的因素：

一. 中间代码的选择

1. 便于生成目标代码；
2. 便于优化；

二. 确定实施各类优化的内容、次序和具体技术

1. 内容：适合实施的具体优化工作；
2. 次序：对提高优化效率，减少优化代价很重要。


控制流分析
数据流分析
循环优化
会加入新的基本块
全局优化
调整语句，撤消某个基本块
局部优化
完善

三. 平衡提高优化效率、减少优化代价的矛盾

- 优化效率本身的矛盾： 代码执行时间的减少；
  存储空间占用的减少；
- 优化考虑严密、完善，不顾及完成优化所花费的代价，则会相对抵消整个编译程序的效率、质量甚至影响优化的实际效率；
- 策略：针对具体问题抓住主要矛盾，估计主要因素；如，
  - * 目标机环境；
  - * 循环优化：最内层优化；
  - * 数据流分析信息对优化的应用价值；
  - * 通用、专用性语言，库函数、包 …

✓代码优化的基本概念

✓优化技术的几种分类方法及类别

✓优化的技术基础：控制流分析（流图）和数据流分析

✓局部优化：基本块划分，构造DAG，优化实施

✓循环优化的技术基础：控制流分析（循环查找）基础上的数据流分析，数据流方程及其中引入的概念

✓ 循环优化：不变外提、强度消弱、删除归纳变量

结束