

✧ 目标代码生成及代码优化基础

- ✧ 二者在编译程序中的逻辑位置
- ✧ 基本块、流图和循环
- ✧ 数据流分析基础
- ✧ 基本块的 DAG 表示（局部优化技术）
- ✧ 目标代码生成技术
- ✧ 代码优化技术

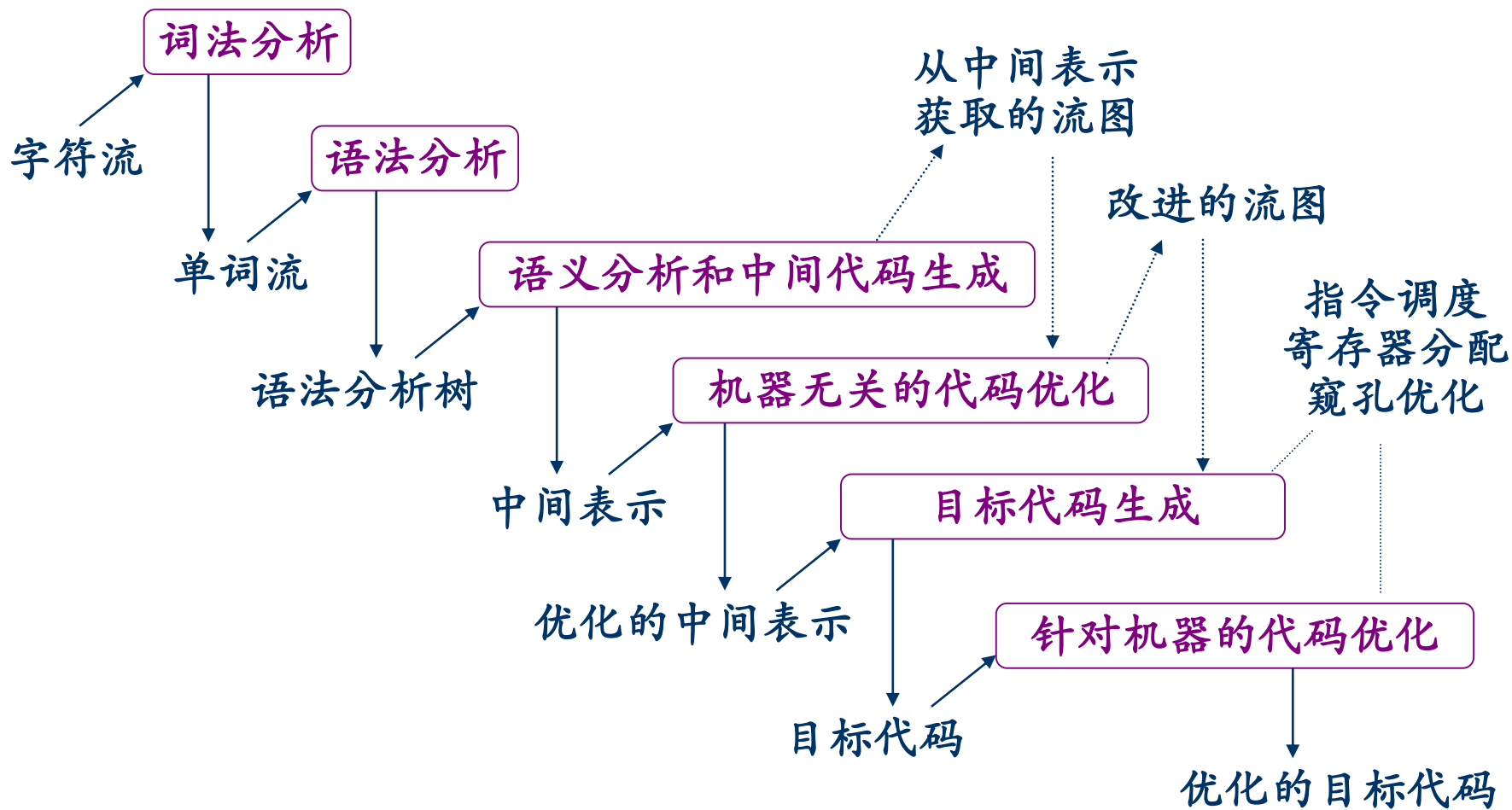
目标代码生成及代码优化基础



清华大学

《编译原理》

◇ 二者在编译程序中的逻辑位置



◇ 基本块 (*basic block*)

— 概念

- 程序中一个顺序执行的语句序列
- 只有一个入口语句和一个出口语句
- 除入口语句外其他语句均不可以带标号
- 除出口语句外其他语句均不可能是转移或停语句

— 入口语句

- 程序的第一个语句；或者
- 条件转移语句或无条件转移语句的转移目标语句；或者
- 紧跟在条件转移语句后面的语句

◇ 划分基本块的算法

— 针对三地址码 (TAC)

— 步骤

- 求出 TAC 程序之中各个基本块的入口语句
- 对每一入口语句，构造其所属的基本块。它是由该语句到下一入口语句（不包括下一入口语句），或到一转移语句（包括该转移语句），或到一停语句（包括该停语句）之间的语句序列组成的
- 凡未被纳入某一基本块的语句，都是程序中控制流程无法到达的语句，因而也是不会被执行到的语句，可以把它们删除

◇ 划分基本块的算法

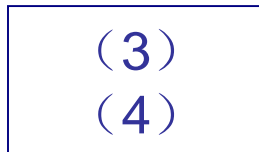
— 针对三地址码 (TAC)

— 举例 右边 TAC 程序可划分成 4 个基本块

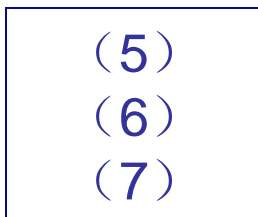
B1



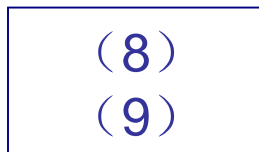
B2



B3



B4



*(1) read x

(2) read y

*(3) r:=x mod y

(4) if r=0 goto (8)

*(5) x:=y

(6) y:=r

(7) goto(3)

*(8) write y

(9) halt

✧ 流图 (flow graph)

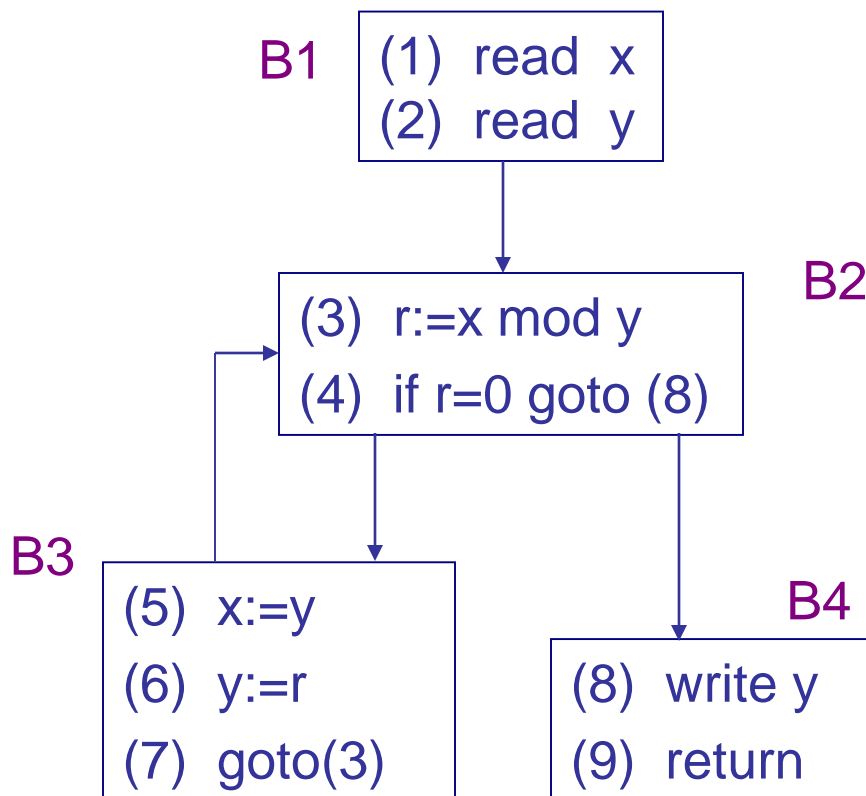
- 概念 可以为构成程序的基本块增加控制流信息，方法是构造一个有向图，称之为流图或控制流图 (CFG, Control-Flow Graph)

流图以基本块集为结点集；第一个结点为含有程序第一条语句的基本块；从基本块 i 到基本块 j 之间存在有向边，当且仅当

- 基本块 j 在程序的位置紧跟在 i 后,且 i 的出口语句不是转移 (可为条件转移)语句、停语句或者返回语句；或者
- i 的出口是 $\text{goto}(S)$ 或 $\text{if goto}(S)$, 而 (S) 是 j 的入口语句

◇ 流图

— 举例



*(1) read x
(2) read y
*(3) r:=x mod y
(4) if r=0 goto (8)
*(5) x:=y
(6) y:=r
(7) goto(3)
*(8) write y
(9) return

◇ 循环 (loop)

— 支配结点集 (dominators)

如果从流图的首结点出发,到达 n 的任意通路都要经过 m , 则称 m 支配 n , 或 m 是 n 的支配结点, 记为 $m \text{ DOM } n$ ($\forall a. a \text{ DOM } a$)

结点 n 的所有支配结点的集合, 称为结点 n 的支配结点集, 记为 $D(n)$.

◇ 循环 (loop)

— 支配结点集举例

$$D(1) = \{1\}$$

$$D(2) = \{1, 2\}$$

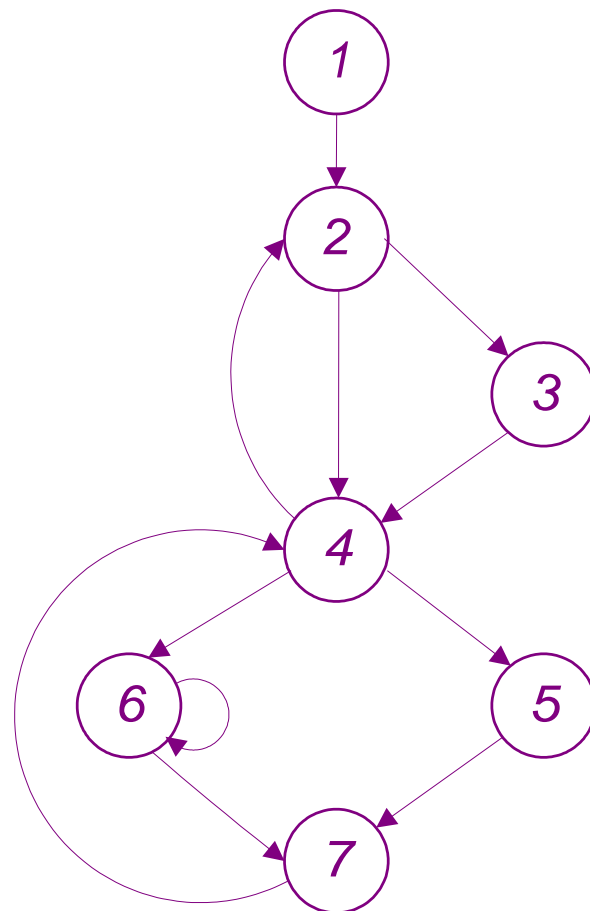
$$D(3) = \{1, 2, 3\}$$

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

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

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

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



◇ 循环 (loop)

— 自然循环 (natural loop)

假设 $n \rightarrow d$ 是流图中的一条有向边，如果 $d \text{ DOM } n$ 则称 $n \rightarrow d$ 是流图中的一条回边 (back edge)

有向边 $n \rightarrow d$ 是回边，它对应的自然循环是由结点 d ，结点 n 以及有通路到达 n 而该通路不经过 d 的所有结点组成，并且 d 是该循环的唯一入口结点

同时，因 d 是 n 的支配结点，所以 d 必可达该循环中任意结点

注：流图中的任何结点都是从首结点可达的

◇ 循环 (loop)

— 自然循环举例

对应回边 $6 \rightarrow 6$:

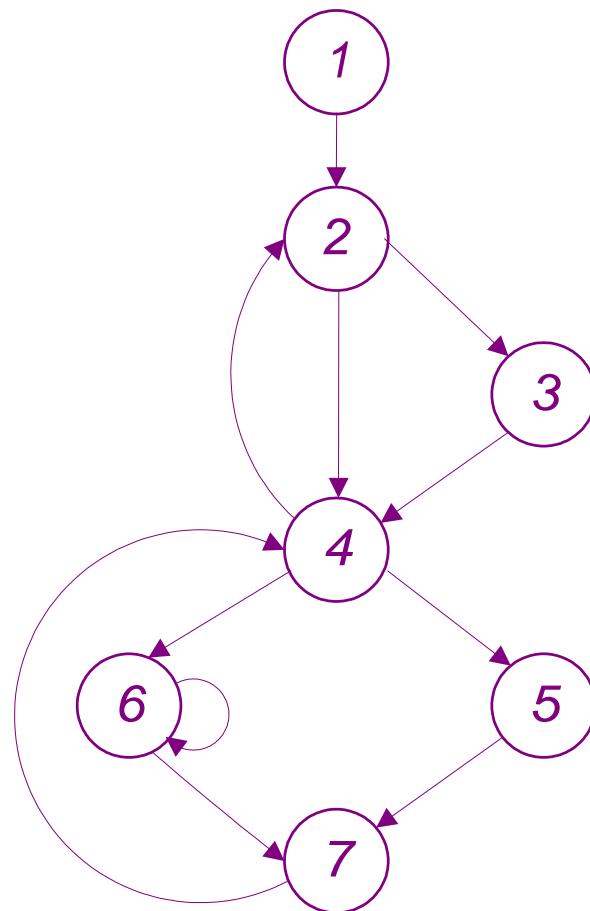
$\{ 6 \}$

对应回边 $7 \rightarrow 4$:

$\{ 4, 5, 6, 7 \}$

对应回边 $4 \rightarrow 2$:

$\{ 2, 3, 5, 6, 7, 4 \}$



◇ 数据流分析 (*data-flow analysis*)

— 作用与目的

为做好代码生成和代码优化工作，通常需要收集整个程序的一些特定信息，并把这些信息分配到流图中的语句单元（如基本块、循环、或单条语句等）中

称这些信息为数据流信息，上述过程为数据流分析

— 数据流信息收集的一种途径

- 建立和求解数据流方程 (*data-flow equation*)

◇ 数据流方程

— 典型的数据流方程举例

(以面向基本块的某种正向数据流为例)

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

其含义为：基本块 S 出口处的数据流信息 ($out[S]$) 或者是 S 内部产生的信息 ($gen[S]$)，或者是从 S 开始处进入 ($in[S]$) 但在穿过 S 的控制流时未被杀死 ($killed$) 的信息 (不在 $kill[S]$ 中)

S 还可以是：

其他语句块、编译区域 (*region*)、单条语句等

◇ 数据流分析举例

- 到达-定值 (*reaching definitions*) 数据流分析
- 活跃变量 (*live variables*) 数据流分析

◇ 到达-定值数据流分析

- 变量 A 的定值 (definition) 是一个 (TAC) 语句, 它赋值或可能赋值给 A

最普通的定值是对 A 的赋值或读值到 A 的语句, 该语句的位置 称作 A 的定值点

- 变量 A 的定值点 d 到达某点 p , 是指如果有路径从紧跟 d 的点到达 p , 并且在这条路径上 d 未被“杀死” (指该变量重新被定值) .

直观地说, 是指流图中从 d 有一条路径到达 p 且该通路上没有 A 的其它定值

◇ 到达-定值数据流分析

— 数据流方程

$$OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$$

$$IN[B] = \bigcup_{b \in P[B]} OUT[b]$$

其中, $P[B]$ 为 B 的所有前驱基本块;

$GEN[B]$ 为 B 中定值并可到达 B 出口处的所有定值点集合;

$KILL[B]$ 为 B 之外的能够到达 B 的入口处、且其定值的变量在 B 中又重新定值的那些定值点的集合;

$IN[B]$ 为到 B 入口处各变量的所有可到达的定值点的集合;

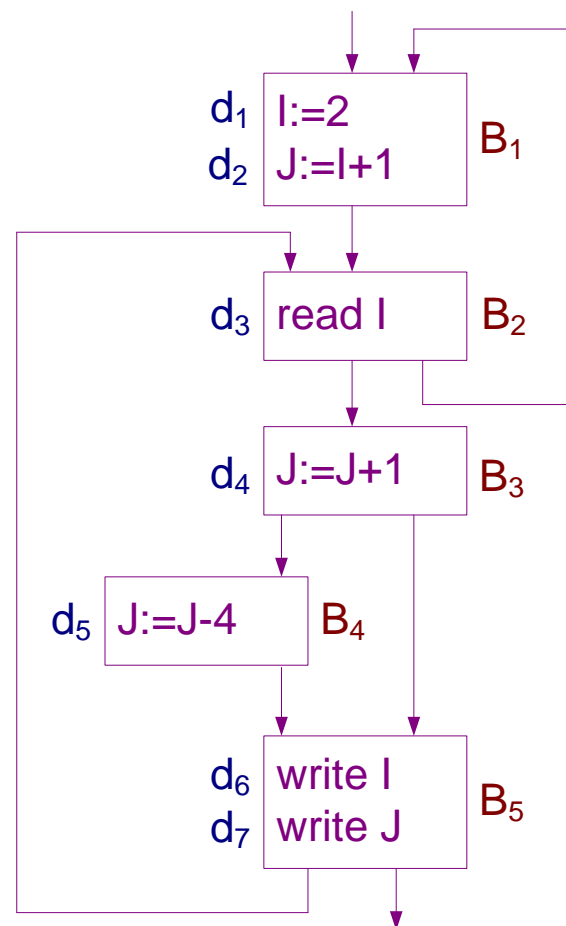
$OUT[B]$ 为到达 B 出口处各变量的所有可到达的定值点的集合

◇ 到达-定值数据流分析

— 数据流方程求解举例

对于右边的流图（略去了基本块中的一些跳转语句），如下是上页数据流方程的一个解：

	GEN	KILL	IN	OUT
B_1	$\{d_1, d_2\}$	$\{d_3, d_4, d_5\}$	$\{d_2, d_3, d_4, d_5\}$	$\{d_1, d_2\}$
B_2	$\{d_3\}$	$\{d_1\}$	$\{d_1, d_2, d_3, d_4, d_5\}$	$\{d_2, d_3, d_4, d_5\}$
B_3	$\{d_4\}$	$\{d_2, d_5\}$	$\{d_2, d_3, d_4, d_5\}$	$\{d_3, d_4\}$
B_4	$\{d_5\}$	$\{d_4\}$	$\{d_3, d_4\}$	$\{d_3, d_5\}$
B_5	$\{\}$	$\{\}$	$\{d_3, d_4, d_5\}$	$\{d_3, d_4, d_5\}$



◇ 到达-定值数据流分析

— 数据流方程求解算法（对 n 个结点的流图）

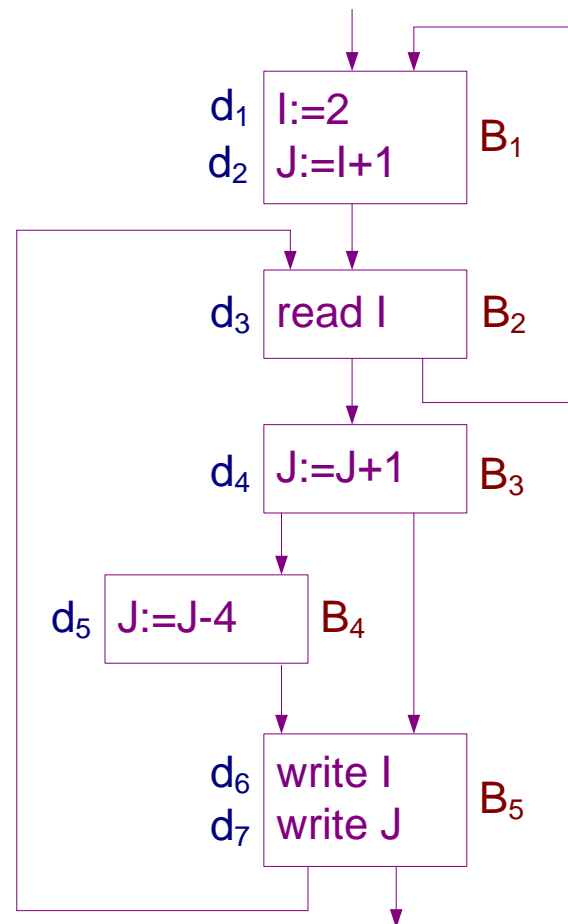
```
for  $i := 1$  to  $n$  {  $IN[B_i] := \emptyset$ ;  $OUT[B_i] := GEN[B_i]$ ; }  
change := true;  
while change {  
    change := false;  
    for  $i := 1$  to  $n$  {  
        newin :=  $\cup OUT[p]$ ;           //  $p \in P[B_i]$   
        if newin  $\neq IN[B_i]$  {  
            change := true;     $IN[B_i] := newin$ ;  
             $OUT[B_i] := (IN[B_i] - KILL[B_i]) \cup GEN[B_i]$   
        }  
    }  
}
```

◇ 到达-定值数据流分析

— 数据流方程求解过程举例

对于右边的流图，如下是上页数据流方程的一个求解过程：

	GEN	KILL	IN	OUT
B_1	$\{d_1, d_2\}$	$\{d_3, d_4, d_5\}$	$\{ \}$	$\{d_1, d_2\}$
B_2	$\{d_3\}$	$\{d_1\}$	$\{ \}$	$\{d_3\}$
B_3	$\{d_4\}$	$\{d_2, d_5\}$	$\{ \}$	$\{d_4\}$
B_4	$\{d_5\}$	$\{d_4\}$	$\{ \}$	$\{d_5\}$
B_5	$\{ \}$	$\{ \}$	$\{ \}$	$\{ \}$

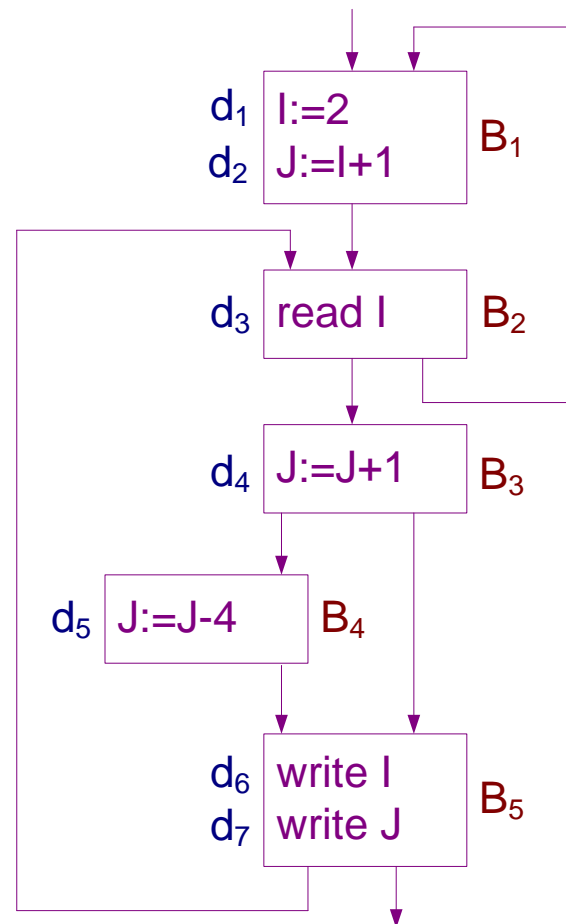


◇ 到达-定值数据流分析

— 数据流方程求解过程举例

对于右边的流图，如下是上页数据流方程的一个求解过程：

	GEN	KILL	IN	OUT
B_1	$\{d_1, d_2\}$	$\{d_3, d_4, d_5\}$	$\{d_3\}$	$\{d_1, d_2\}$
B_2	$\{d_3\}$	$\{d_1\}$	$\{ \}$	$\{d_3\}$
B_3	$\{d_4\}$	$\{d_2, d_5\}$	$\{ \}$	$\{d_4\}$
B_4	$\{d_5\}$	$\{d_4\}$	$\{ \}$	$\{d_5\}$
B_5	$\{ \}$	$\{ \}$	$\{ \}$	$\{ \}$

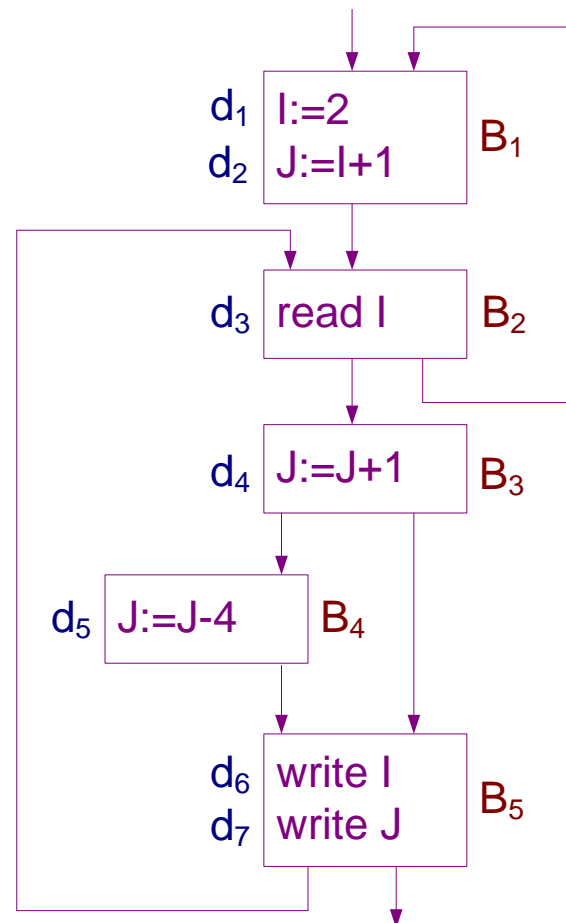


◇ 到达-定值数据流分析

— 数据流方程求解过程举例

对于右边的流图，如下是上页数据流方程的一个求解过程：

	GEN	KILL	IN	OUT
B_1	$\{d_1, d_2\}$	$\{d_3, d_4, d_5\}$	$\{d_3\}$	$\{d_1, d_2\}$
B_2	$\{d_3\}$	$\{d_1\}$	$\{d_1, d_2\}$	$\{d_2, d_3\}$
B_3	$\{d_4\}$	$\{d_2, d_5\}$	$\{ \}$	$\{d_4\}$
B_4	$\{d_5\}$	$\{d_4\}$	$\{ \}$	$\{d_5\}$
B_5	$\{ \}$	$\{ \}$	$\{ \}$	$\{ \}$

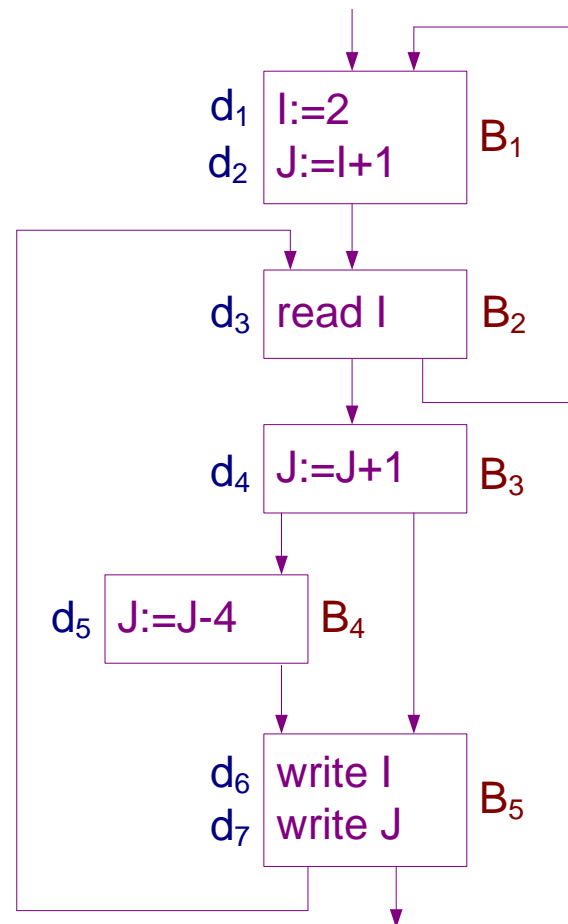


◇ 到达-定值数据流分析

— 数据流方程求解过程举例

对于右边的流图，如下是上页数据流方程的一个求解过程：

	GEN	KILL	IN	OUT
B_1	$\{d_1, d_2\}$	$\{d_3, d_4, d_5\}$	$\{d_3\}$	$\{d_1, d_2\}$
B_2	$\{d_3\}$	$\{d_1\}$	$\{d_1, d_2\}$	$\{d_2, d_3\}$
B_3	$\{d_4\}$	$\{d_2, d_5\}$	$\{d_2, d_3\}$	$\{d_3, d_4\}$
B_4	$\{d_5\}$	$\{d_4\}$	$\{\}$	$\{d_5\}$
B_5	$\{\}$	$\{\}$	$\{\}$	$\{\}$

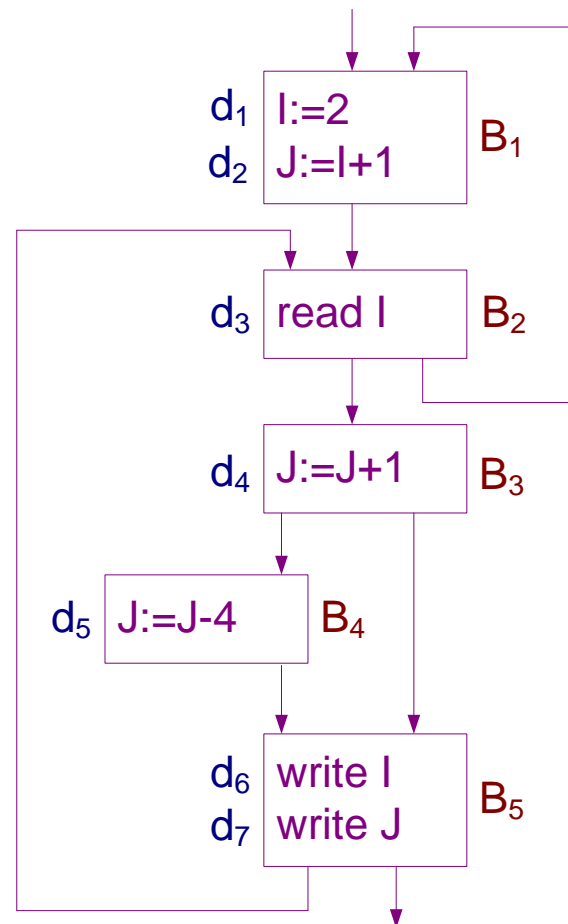


◇ 到达-定值数据流分析

— 数据流方程求解过程举例

对于右边的流图，如下是上页数据流方程的一个求解过程：

	GEN	KILL	IN	OUT
B_1	$\{d_1, d_2\}$	$\{d_3, d_4, d_5\}$	$\{d_3\}$	$\{d_1, d_2\}$
B_2	$\{d_3\}$	$\{d_1\}$	$\{d_1, d_2\}$	$\{d_2, d_3\}$
B_3	$\{d_4\}$	$\{d_2, d_5\}$	$\{d_2, d_3\}$	$\{d_3, d_4\}$
B_4	$\{d_5\}$	$\{d_4\}$	$\{d_3, d_4\}$	$\{d_3, d_5\}$
B_5	$\{\}$	$\{\}$	$\{\}$	$\{\}$

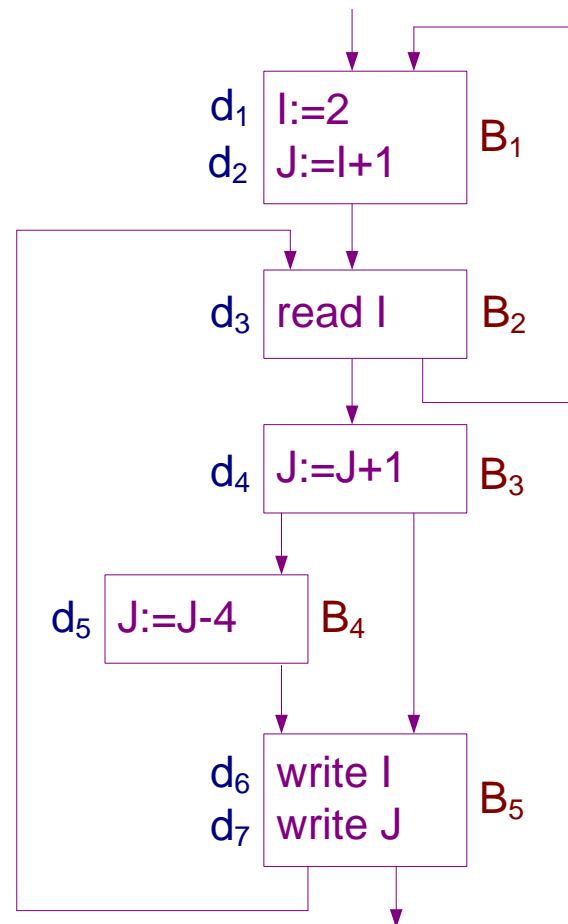


◇ 到达-定值数据流分析

— 数据流方程求解过程举例

对于右边的流图，如下是上页数据流方程的一个求解过程：

	GEN	KILL	IN	OUT
B_1	$\{d_1, d_2\}$	$\{d_3, d_4, d_5\}$	$\{d_3\}$	$\{d_1, d_2\}$
B_2	$\{d_3\}$	$\{d_1\}$	$\{d_1, d_2\}$	$\{d_2, d_3\}$
B_3	$\{d_4\}$	$\{d_2, d_5\}$	$\{d_2, d_3\}$	$\{d_3, d_4\}$
B_4	$\{d_5\}$	$\{d_4\}$	$\{d_3, d_4\}$	$\{d_3, d_5\}$
B_5	$\{\}$	$\{\}$	$\{d_3, d_4, d_5\}$	$\{d_3, d_4, d_5\}$

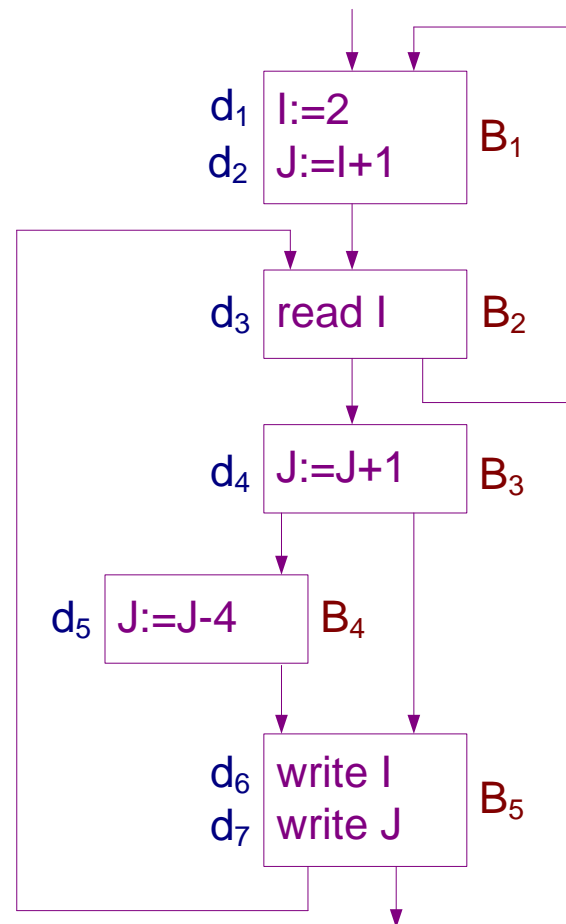


◇ 到达-定值数据流分析

— 数据流方程求解过程举例

对于右边的流图，如下是上页数据流方程的一个求解过程：

	GEN	KILL	IN	OUT
B_1	$\{d_1, d_2\}$	$\{d_3, d_4, d_5\}$	$\{d_2, d_3\}$	$\{d_1, d_2\}$
B_2	$\{d_3\}$	$\{d_1\}$	$\{d_1, d_2\}$	$\{d_2, d_3\}$
B_3	$\{d_4\}$	$\{d_2, d_5\}$	$\{d_2, d_3\}$	$\{d_3, d_4\}$
B_4	$\{d_5\}$	$\{d_4\}$	$\{d_3, d_4\}$	$\{d_3, d_5\}$
B_5	$\{\}$	$\{\}$	$\{d_3, d_4, d_5\}$	$\{d_3, d_4, d_5\}$

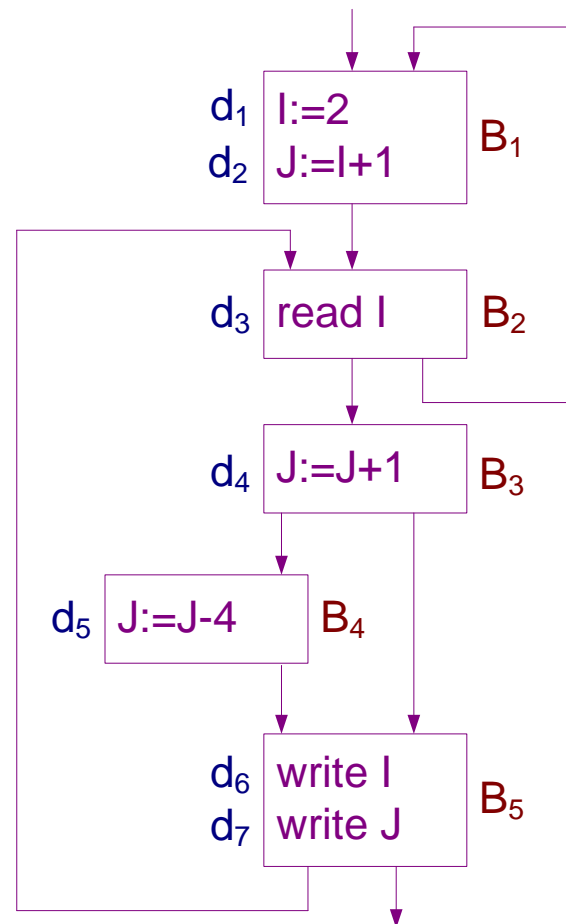


◇ 到达-定值数据流分析

— 数据流方程求解过程举例

对于右边的流图，如下是上页数据流方程的一个求解过程：

	GEN	KILL	IN	OUT
B_1	$\{d_1, d_2\}$	$\{d_3, d_4, d_5\}$	$\{d_2, d_3\}$	$\{d_1, d_2\}$
B_2	$\{d_3\}$	$\{d_1\}$	$\{d_1, d_2, d_3, d_4, d_5\}$	$\{d_2, d_3, d_4, d_5\}$
B_3	$\{d_4\}$	$\{d_2, d_5\}$	$\{d_2, d_3\}$	$\{d_3, d_4\}$
B_4	$\{d_5\}$	$\{d_4\}$	$\{d_3, d_4\}$	$\{d_3, d_5\}$
B_5	$\{\}$	$\{\}$	$\{d_3, d_4, d_5\}$	$\{d_3, d_4, d_5\}$

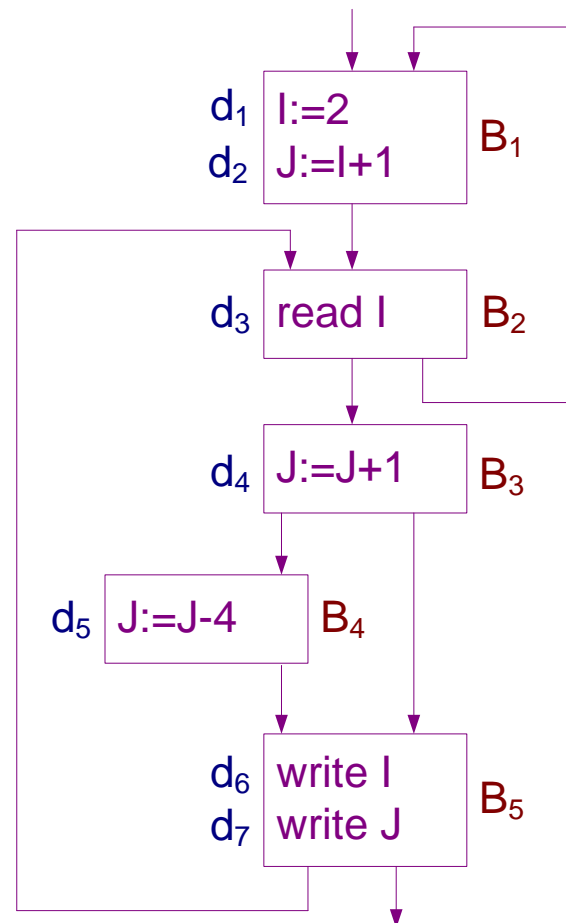


◇ 到达-定值数据流分析

— 数据流方程求解过程举例

对于右边的流图，如下是上页数据流方程的一个求解过程：

	GEN	KILL	IN	OUT
B_1	$\{d_1, d_2\}$	$\{d_3, d_4, d_5\}$	$\{d_2, d_3\}$	$\{d_1, d_2\}$
B_2	$\{d_3\}$	$\{d_1\}$	$\{d_1, d_2, d_3, d_4, d_5\}$	$\{d_2, d_3, d_4, d_5\}$
B_3	$\{d_4\}$	$\{d_2, d_5\}$	$\{d_2, d_3, d_4, d_5\}$	$\{d_3, d_4\}$
B_4	$\{d_5\}$	$\{d_4\}$	$\{d_3, d_4\}$	$\{d_3, d_5\}$
B_5	$\{\}$	$\{\}$	$\{d_3, d_4, d_5\}$	$\{d_3, d_4, d_5\}$

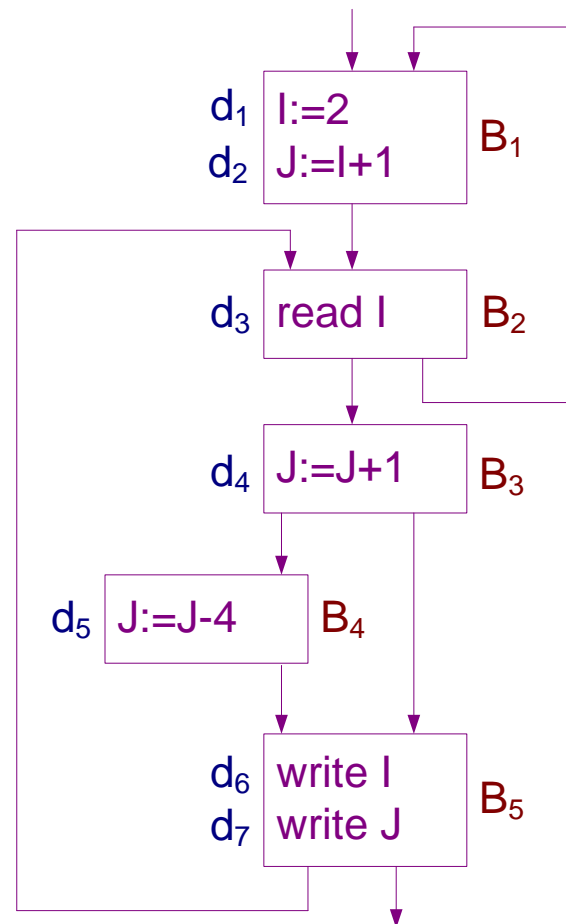


◇ 到达-定值数据流分析

— 数据流方程求解过程举例

对于右边的流图，如下是上页数据流方程的一个求解过程：

	GEN	KILL	IN	OUT
B_1	$\{d_1, d_2\}$	$\{d_3, d_4, d_5\}$	$\{d_2, d_3, d_4, d_5\}$	$\{d_1, d_2\}$
B_2	$\{d_3\}$	$\{d_1\}$	$\{d_1, d_2, d_3, d_4, d_5\}$	$\{d_2, d_3, d_4, d_5\}$
B_3	$\{d_4\}$	$\{d_2, d_5\}$	$\{d_2, d_3, d_4, d_5\}$	$\{d_3, d_4\}$
B_4	$\{d_5\}$	$\{d_4\}$	$\{d_3, d_4\}$	$\{d_3, d_5\}$
B_5	$\{\}$	$\{\}$	$\{d_3, d_4, d_5\}$	$\{d_3, d_4, d_5\}$



◇ 活跃变量数据流分析

— 活跃变量

对程序中的某变量 A 和某点 p 而言, 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

直观地, 对于全局范围的分析来说, 一个变量是活跃的, 如果存在一条路径使得该变量被重新定值之前它的当前值还要被引用

☆ 活跃变量数据流分析

— 活跃变量的数据流方程

$$\text{LiveIn}(B) = \text{LiveUse}(B) \cup (\text{LiveOut}(B) - \text{Def}(B))$$

$$\text{LiveOut}(B) = \bigcup_{s \in S[B]} \text{LiveIn}(s)$$

其中，B 为一个基本块，S[B] 为 B 的所有后继基本块；

LiveUse(B) 为 B 中被定值之前要引用变量的集合；

Def(B) 为在 B 中定值的且定值前未曾在 B 中引用过的变量集合；

LiveIn(B) 为 B 入口处为活跃的变量的集合；

LiveOut(B) 为 B 的出口处的活跃变量的集合

◇ 活跃变量数据流分析

— 数据流方程求解算法（对 n 个结点的流图）

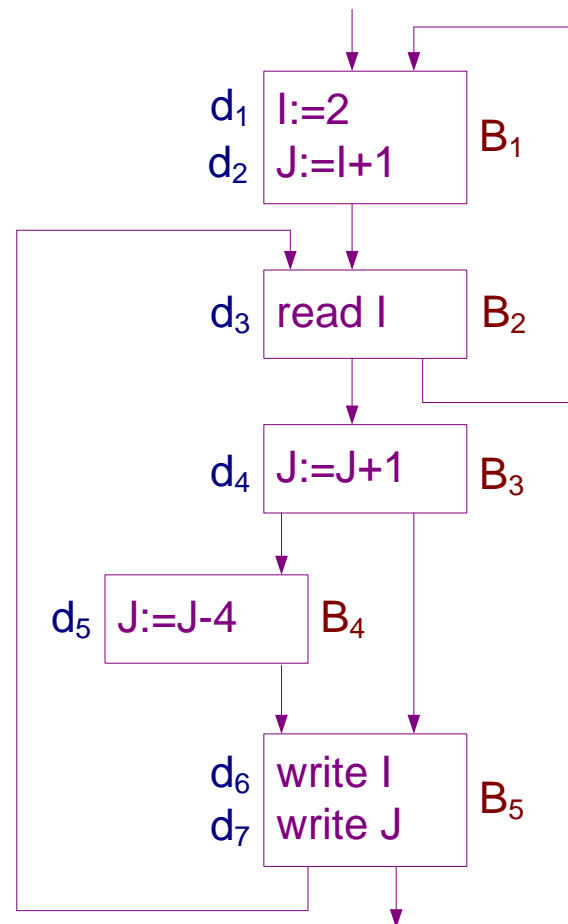
```
for  $i := 1$  to  $n$  {  LiveIn[  $B_i$  ] := LiveUse[  $B_i$  ];  LiveOut[  $B_i$  ] :=  $\emptyset$ ; }  
change := true;  
while change {  
    change := false;  
    for  $i := 1$  to  $n$  {  
        newout :=  $\cup$  LiveIn[ $p$ ];           //  $p \in S[B_i]$   
        if newout  $\neq$  LiveOut[ $B_i$ ] {  
            change := true;  LiveOut[ $B_i$ ] := newout;  
            LiveIn[ $B_i$ ] := (LiveOut[ $B_i$ ] - Def[ $B_i$ ])  $\cup$  LiveUse[ $B_i$ ]  
        }  
    }  
}
```


◇ 活跃变量数据流分析

— 活跃变量数据流方程求解举例

对于右边的流图，提取Def（在B中定值的变量集合）和 LiveUse（B中被定值前要引用变量的集合）集合：

	Def	LiveUse
B_1	$\{I, J\}$	\emptyset
B_2	$\{I\}$	\emptyset
B_3	\emptyset	$\{J\}$
B_4	\emptyset	$\{J\}$
B_5	\emptyset	$\{I, J\}$



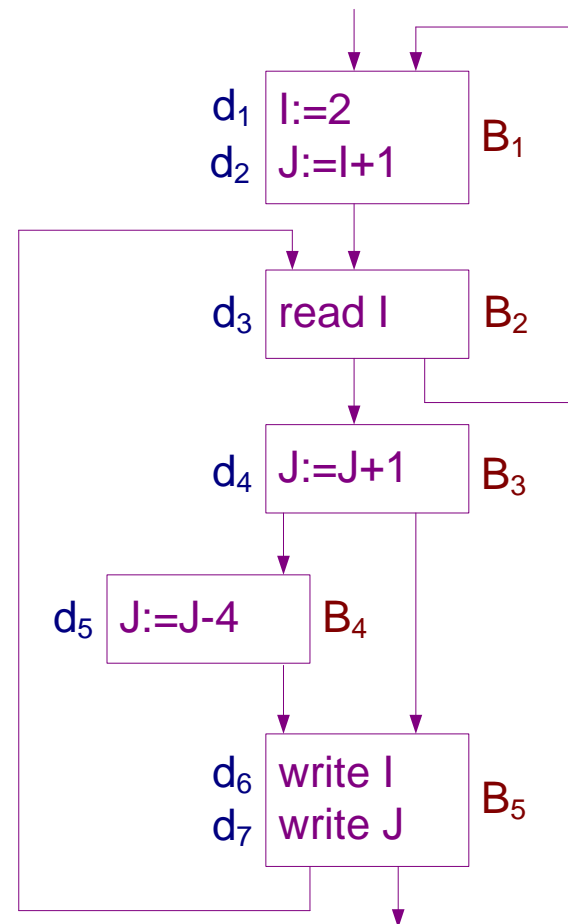
◇ 活跃变量数据流分析

— 活跃变量数据流方程求解举例

$$\text{LiveIn}(B) = \text{LiveUse}(B) \cup (\text{LiveOut}(B) - \text{Def}(B))$$

$$\text{LiveOut}(B) = \bigcup_{s \in S[B]} \text{LiveIn}(s)$$

	Def	LiveUse	LiveOut	LiveIn
B_1	$\{I, J\}$	\emptyset	\emptyset	\emptyset
B_2	$\{I\}$	\emptyset	\emptyset	\emptyset
B_3	\emptyset	$\{J\}$	\emptyset	$\{J\}$
B_4	\emptyset	$\{J\}$	\emptyset	$\{J\}$
B_5	\emptyset	$\{I, J\}$	\emptyset	$\{I, J\}$



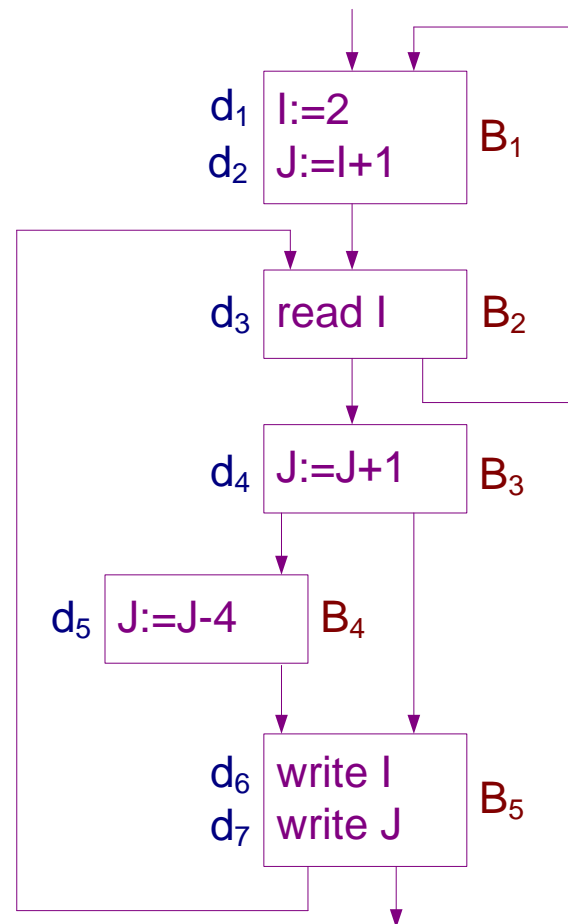
◇ 活跃变量数据流分析

— 活跃变量数据流方程求解举例

$$\text{LiveIn}(B) = \text{LiveUse}(B) \cup (\text{LiveOut}(B) - \text{Def}(B))$$

$$\text{LiveOut}(B) = \bigcup_{s \in S[B]} \text{LiveIn}(s)$$

	Def	LiveUse	LiveOut	LiveIn
B_1	$\{I, J\}$	\emptyset	\emptyset	\emptyset
B_2	$\{I\}$	\emptyset	$\{J\}$	\emptyset
B_3	\emptyset	$\{J\}$	$\{I, J\}$	$\{J\}$
B_4	\emptyset	$\{J\}$	$\{I, J\}$	$\{J\}$
B_5	\emptyset	$\{I, J\}$	\emptyset	$\{I, J\}$



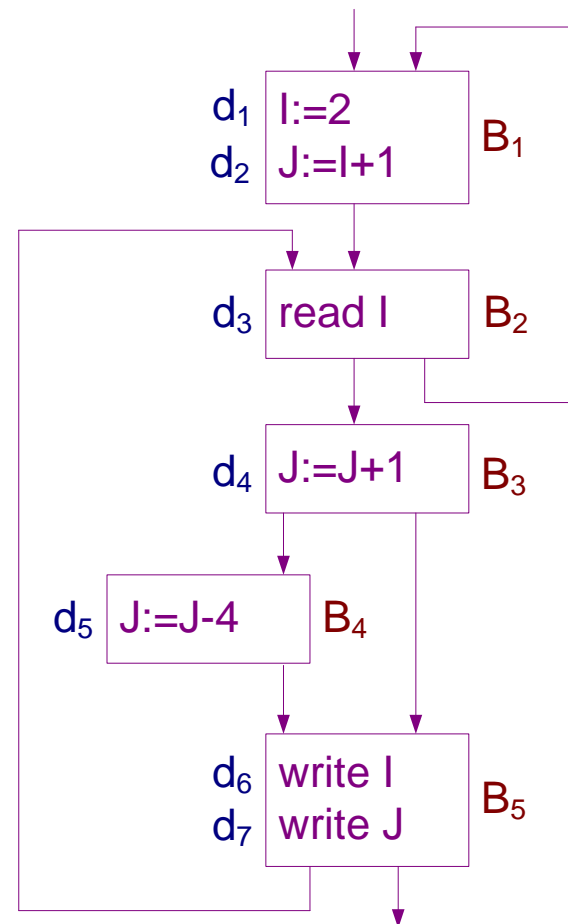
◇ 活跃变量数据流分析

— 活跃变量数据流方程求解举例

$$\text{LiveIn}(B) = \text{LiveUse}(B) \cup (\text{LiveOut}(B) - \text{Def}(B))$$

$$\text{LiveOut}(B) = \bigcup_{s \in S[B]} \text{LiveIn}(s)$$

	Def	LiveUse	LiveOut	LiveIn
B_1	$\{I, J\}$	\emptyset	\emptyset	\emptyset
B_2	$\{I\}$	\emptyset	$\{J\}$	$\{J\}$
B_3	\emptyset	$\{J\}$	$\{I, J\}$	$\{I, J\}$
B_4	\emptyset	$\{J\}$	$\{I, J\}$	$\{I, J\}$
B_5	\emptyset	$\{I, J\}$	\emptyset	$\{I, J\}$



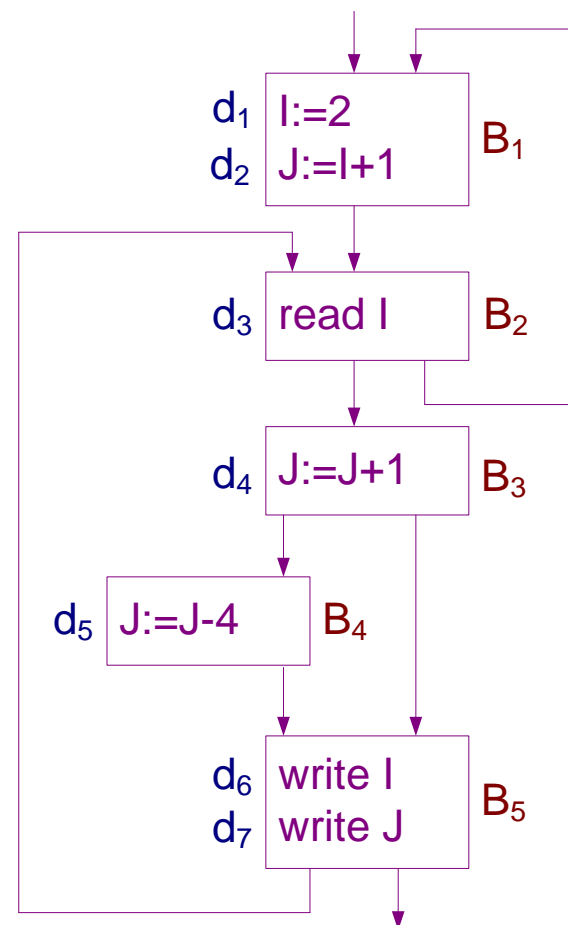
◇ 活跃变量数据流分析

— 活跃变量数据流方程求解举例

$$\text{LiveIn}(B) = \text{LiveUse}(B) \cup (\text{LiveOut}(B) - \text{Def}(B))$$

$$\text{LiveOut}(B) = \bigcup_{s \in S[B]} \text{LiveIn}(s)$$

	Def	LiveUse	LiveOut	LiveIn
B_1	$\{I, J\}$	\emptyset	$\{J\}$	\emptyset
B_2	$\{I\}$	\emptyset	$\{I, J\}$	$\{J\}$
B_3	\emptyset	$\{J\}$	$\{I, J\}$	$\{I, J\}$
B_4	\emptyset	$\{J\}$	$\{I, J\}$	$\{I, J\}$
B_5	\emptyset	$\{I, J\}$	$\{J\}$	$\{I, J\}$



◇ 向前流和向后流

— 向前流

信息流的方向与控制流是一致的（如：到达-定值数据流）

In 集合和 Out 集合的关系：

$$OUT[B] = Used[B] \cup (IN[B] - KILLED[B])$$

— 向后流

信息流的方向与控制流反向（如：活跃变量数据流）

In 集合和 Out 集合的关系：

$$IN[B] = Used[B] \cup (OUT[B] - KILLED[B])$$

◇ 其他有用的数据流信息

— UD 链

假设在程序中某点 u 引用了变量 A 的值，则把能到达 u 的 A 的所有定值点的全体，称为 A 在引用点 u 的引用一定值链 (Use-Definition Chaining)，简称 UD 链。

UD 链的计算 (借助于到达-定值数据流信息 $IN[B]$)

- 如果在基本块 B 中，变量 A 的引用点 u 之前有 A 的定值点 d ，并且 A 在点 d 的定值到达 u ，那么 A 在点 u 的 UD 链就是 $\{d\}$
- 如果在基本块中，变量 A 的引用点 u 之前没有 A 的定值点，那么， $IN[B]$ 中 A 的所有定值点均到达 u ，它们就是 A 在点 u 的 UD 链

◇ 其他有用的数据流信息

– UD 链的计算举例

对于右边的流图，

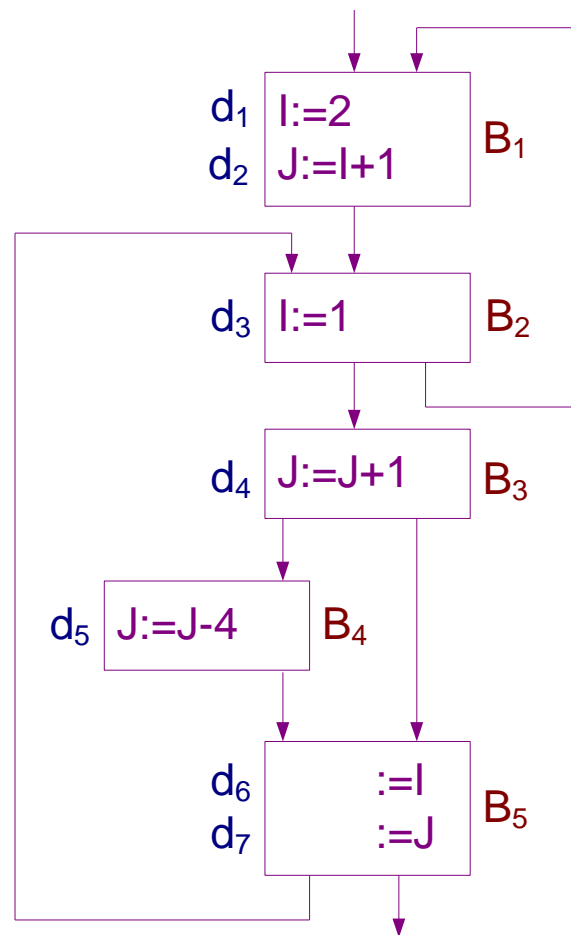
I 在引用点 d_2 的 UD 链为 $\{d_1\}$

J 在 d_4 的 UD 链为 $\{d_2, d_4, d_5\}$

J 在 d_5 的 UD 链为 $\{d_4\}$

I 在引用点 d_6 的 UD 链为 $\{d_3\}$

J 在 d_7 的 UD 链为 $\{d_4, d_5\}$



◇ 其他有用的数据流信息

— DU 链

假设在程序中某点 u 定义了变量 A 的值，从 u 存在一条到达 A 的某个引用点 s 的路径，且该路径上不存在 A 的其他定值点，则把所有此类引用点 s 的全体称为 A 在定值点 u 的**定值—引用链** (Definition-Use Chaining)，简称 DU 链

DU 链的计算 可采用类似活跃变量数据流的向后流方法 (方法细节略，可参考下页的实例)

◇ 其他有用的数据流信息

— DU 链的计算举例

对于右边的流图，

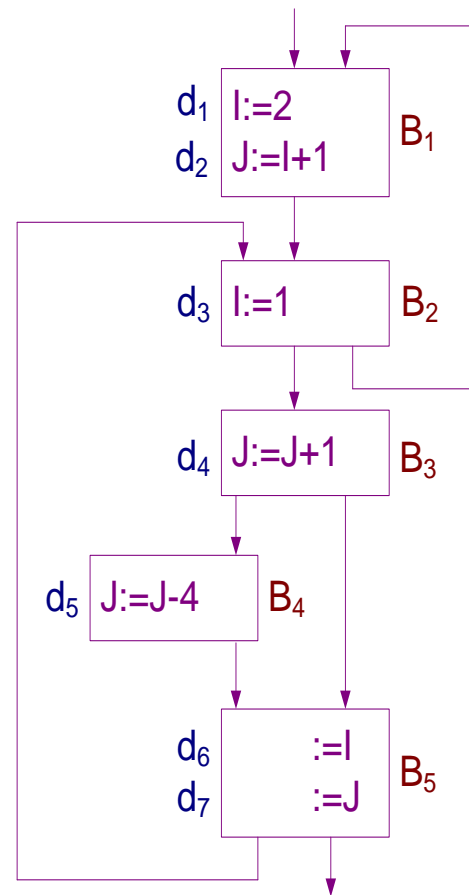
I 在定值点 d_1 的 DU 链为 $\{d_2\}$

J 在定值点 d_2 的 DU 链为 $\{d_4\}$

I 在定值点 d_3 的 DU 链为 $\{d_6\}$

J 在 d_4 的 DU 链为 $\{d_4, d_5, d_7\}$

J 在 d_5 的 DU 链为 $\{d_4, d_7\}$



◇ 其他有用的数据流信息

— 基本块内变量的待用信息和活跃信息

- 待用信息

基本块中某变量定值点的待用信息(next use)链即为该点在基本块范围内的 DU 链中最近的引用点

- 活跃信息

基本块中的活跃信息链体现了以语句为单位的活跃变量信息

☆ 其他有用的数据流信息

— 计算基本块内变量的待用信息和活跃信息

- 为每个变量建立“待用信息”栏和“活跃信息”栏

为“待用信息”栏和“活跃信息”栏置初值：即把“待用信息”栏置“非待用”，对“活跃信息”栏按在基本块出口处是否为活跃而置成“活跃”或“非活跃”（比如：假定临时变量都是非活跃的，其他变量是活跃的）

◇ 其他有用的数据流信息

— 计算基本块内变量的待用信息和活跃信息

- 从基本块出口到入口对每个TAC语句 $i: A := B \text{ op } C$ 依次执行下述步骤：
 - a) 把变量A的待用信息和活跃信息附加到 TAC 语句 i 上；
 - b) 把变量A的待用信息栏和活跃信息栏分别置为“非待用”和“非活跃”
(由于在 i 中对 A 的定值只能在 i 以后的TAC语句才能引用，因而对 i 以前的TAC语句来说 A 是不活跃也不可能是待用的)
 - c) 把变量 B 和 C 的待用信息和活跃信息附加到 TAC 语句 i 上
 - d) 把变量 B 和 C 的待用信息栏置为“i”，活跃信息栏置为“活跃”。

注：以上a), b), c), d) 的次序不能颠倒。

◇ 其他有用的数据流信息

— 计算基本块内变量的待用信息和活跃信息

- **举例** 若用 A, B, C, D 表示变量, 用 T, U, V 表示中间变量, 设有 TAC 语句构成的如下基本块:

```
T:=A-B  
U:=D-C  
V:=T+U  
D:=V+U
```

其名字表中的待用信息和活跃信息见下页的表中, 用“F”表示“非待用”以及“非活跃”, 用“L”表示活跃

(1), (2), (3), (4)表示 TAC 语句序号

(接上页)

待用信息和活跃信息

变量名	待用信息					活跃信息				
	初值	待用信息链				初值	活跃信息链			
A	F				(1)	L				L
B	F				(1)	L				L
C	F			(2)		L			L	
D	F	F		(2)		L	F		L	
T	F		(3)		F	F		L		F
U	F	(4)	(3)	F		F	L	L	F	
V	F	(4)	F			F	L	F		

表中“待用信息链”与“活跃信息链”的每列从左至右为每从后向前扫描一个 TAC 语句时相应变量的信息变化情况，空白处为没变化。

待用信息和活跃信息在 TAC 语句上的标记如下所示：

(1) $T := A - B$

$T^{(3)}L := A - B$

$T^{(3)}L := A^{FL} - B^{FL}$

(2) $U := D - C$

$U^{(3)}L := D - C$

$U^{(3)}L := D^{FF} - C^{FL}$

(3) $V := T + U$

$V^{(4)}L := T + U$

$V^{(4)}L := T^{FF} + U^{(4)}L$

(4) $D := V + U$

$D^{FL} := V + U$

$D^{FL} := V^{FF} + U^{FF}$

◇ 基本块的 DAG 表示

– DAG

DAG 指有向无圈图 (*Directed Acyclic Graph*)

– 基本块的 DAG 是在结点上带有标记的 DAG

叶结点 代表名字的初值，以唯一的标识符（变量名字或常数）标记（为避免混乱，用 x_0 表示变量名字 x 的初值）

内部结点 用运算符号标记

所有结点都可有一个附加的变量名字表

基本块的 DAG 表示

◇ 基本块 DAG 表示的构造

- 仅考虑三种形式的 TAC 语句

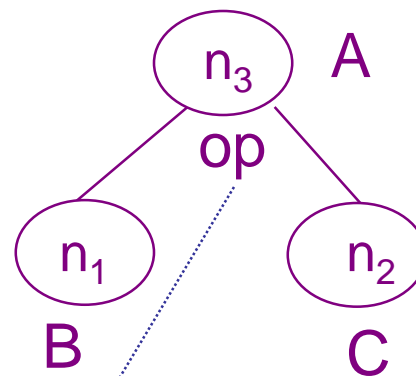
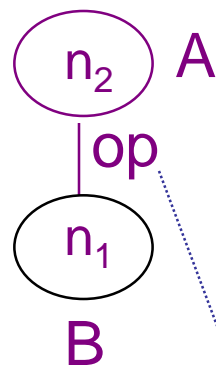
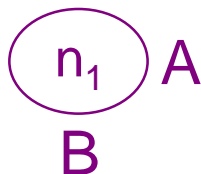
TAC 语句

$A := B$

$A := \text{op } B$

$A := B \text{ op } C$

DAG 子图



运算符标记

(其他 TAC 语句的 DAG 结点的讨论参见龙书)

◇ 基本块 DAG 表示的构造

— 构造算法

设 $x := y \text{ op } z$, $x := \text{op } y$, $x := y$ 分别为第1、2、3种 TAC 语句

设函数 $node(name)$ 返回最近创建的关联于 $name$ 的结点

首先, 置 DAG 为空.

对基本块的每一 TAC 语句, 依次进行下列步骤:

- 若 $node(y)$ 无定义, 则创建一个标记为 y 的叶结点, 并令 $node(y)$ 为这个结点; 对第1种语句, 若 $node(z)$ 无定义, 再创建标记为 z 的叶结点, 并令 $node(z)$ 为这个结点

(转下页)

◇ 基本块 DAG 表示的构造

— 构造算法

(续上页)

- 对于第 1 种语句, 若 $node(y)$ 和 $node(z)$ 都是标记为常数的叶结点, 执行 $y \text{ op } z$, 令得到的新常数为 p . 若 $node(p)$ 无定义, 则构造一个用 p 做标记的叶结点 n . 若 $node(y)$ 或 $node(z)$ 是处理当前语句时新构造出来的结点, 则删除它. 置 $node(p)=n$. (起到合并已知量的作用)

若 $node(y)$ 或 $node(z)$ 不是标记为常数的叶结点, 则检查是否存在某个标记为 op 的结点, 其左孩子是 $node(y)$, 而右孩子是 $node(z)$? 若无, 则创建这样的结点. 无论有无, 都令该结点为 n . (可能起到删除多余运算的作用)

(转下页)

◇ 基本块 DAG 表示的构造

— 构造算法

(续上页)

- 对于第 2 种语句，若 $node(y)$ 是标记为常数的叶结点，执行 $op\ y$ ，令得到的新常数为 p 。若 $node(p)$ 无定义，则构造一个用 p 做标记的叶结点 n 。若 $node(y)$ 是处理当前语句时新构造出来的结点，则删除它。置 $node(p)=n$ 。
(这一步起到合并已知量的作用)

若 $node(y)$ 不是标记为常数的叶结点，则检查是否存在某个标记为 op 的结点，其唯一的孩子是 $node(y)$ ？若无，则创建这样的结点。无论有无，都令该结点为 n 。
(这一步可能起到删除多余运算的作用)

(转下页)

◇ 基本块 DAG 表示的构造

— 构造算法

(续上页)

- 对于第 3 种语句，令 $node(y)$ 为 n
- 最后，从 $node(x)$ 的附加标识符表中将 x 删除，将其添加到结点 n 的附加附加变量名字表中，并置 $node(x)$ 为 n (这一步有删除无用赋值的作用)

基本块的 DAG 表示

◇ 基本块 DAG 表示的构造

— 举例

$T0 := 3.14$ 

$T1 := 2 * T0$

$T2 := R + r$

$A := T1 * T2$

$B := A$

$T3 := 2 * T0$

$T4 := R + r$

$T5 := T3 * T4$

$T6 := R - r$

$B := T5 * T6$

基本块的 DAG 表示

◇ 基本块 DAG 表示的构造

— 举例

$T0 := 3.14$

$T1 := 2 * T0$



$T2 := R + r$

$A := T1 * T2$

$B := A$

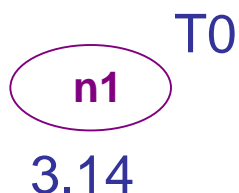
$T3 := 2 * T0$

$T4 := R + r$

$T5 := T3 * T4$

$T6 := R - r$

$B := T5 * T6$



基本块的 DAG 表示

◇ 基本块 DAG 表示的构造

— 举例

$T0 := 3.14$

$T1 := 2 * T0$

$T2 := R + r$



$A := T1 * T2$

$B := A$

$T3 := 2 * T0$

$T4 := R + r$

$T5 := T3 * T4$

$T6 := R - r$

$B := T5 * T6$



基本块的 DAG 表示

◇ 基本块 DAG 表示的构造

— 举例

$T0:=3.14$

$T1:=2*T0$

$T2:=R+r$

$A:=T1*T2$ ←

$B:=A$

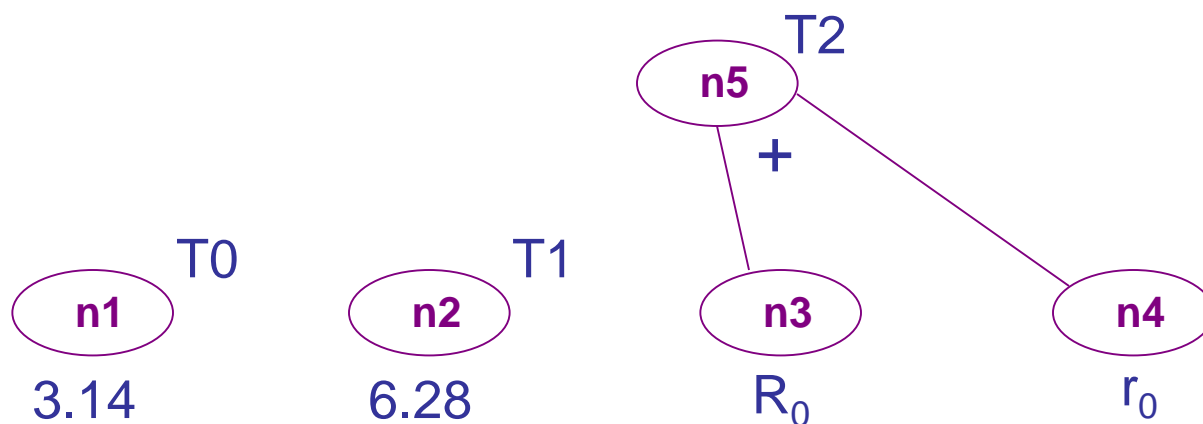
$T3:=2*T0$

$T4:=R+r$

$T5:=T3*T4$

$T6:=R-r$

$B:=T5*T6$



基本块的 DAG 表示

◇ 基本块 DAG 表示的构造

— 举例

$T0:=3.14$

$T1:=2*T0$

$T2:=R+r$

$A:=T1*T2$

$B:=A$

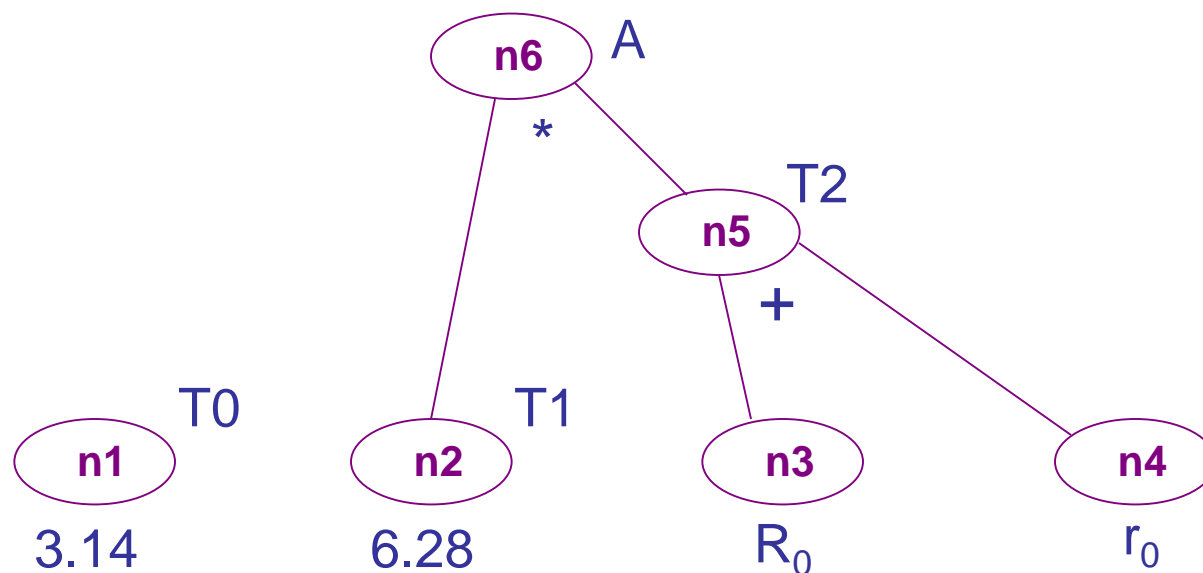
$T3:=2*T0$

$T4:=R+r$

$T5:=T3*T4$

$T6:=R-r$

$B:=T5*T6$



基本块的 DAG 表示

◇ 基本块 DAG 表示的构造

— 举例

$T0:=3.14$

$T1:=2*T0$

$T2:=R+r$

$A:=T1*T2$

$B:=A$

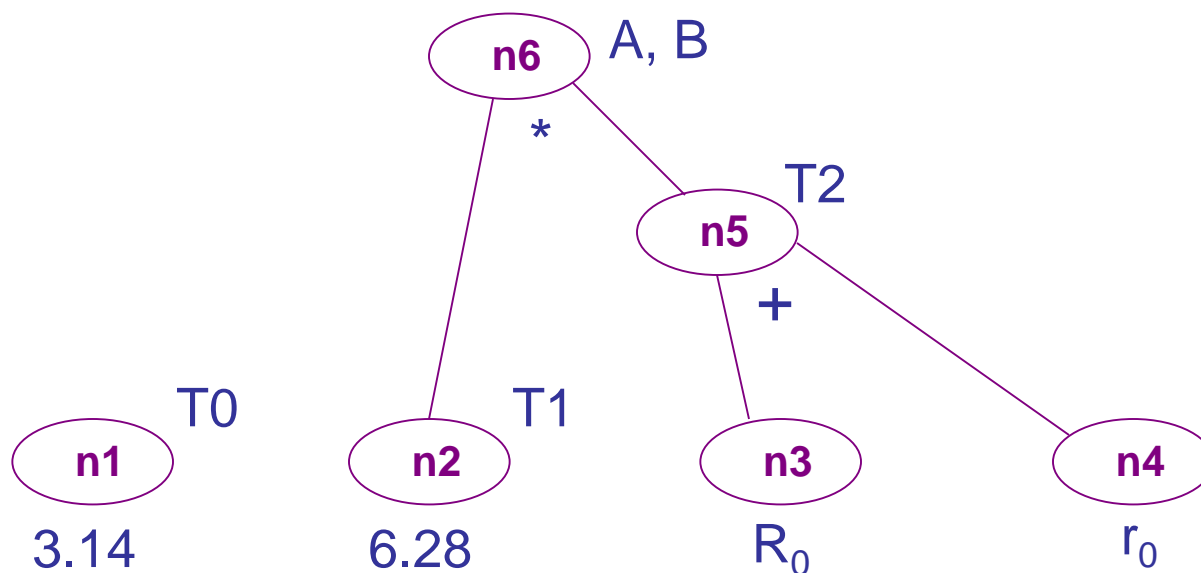
$T3:=2*T0$ ←

$T4:=R+r$

$T5:=T3*T4$

$T6:=R-r$

$B:=T5*T6$



基本块的 DAG 表示

◇ 基本块 DAG 表示的构造

— 举例

$T0:=3.14$

$T1:=2*T0$

$T2:=R+r$

$A:=T1*T2$

$B:=A$

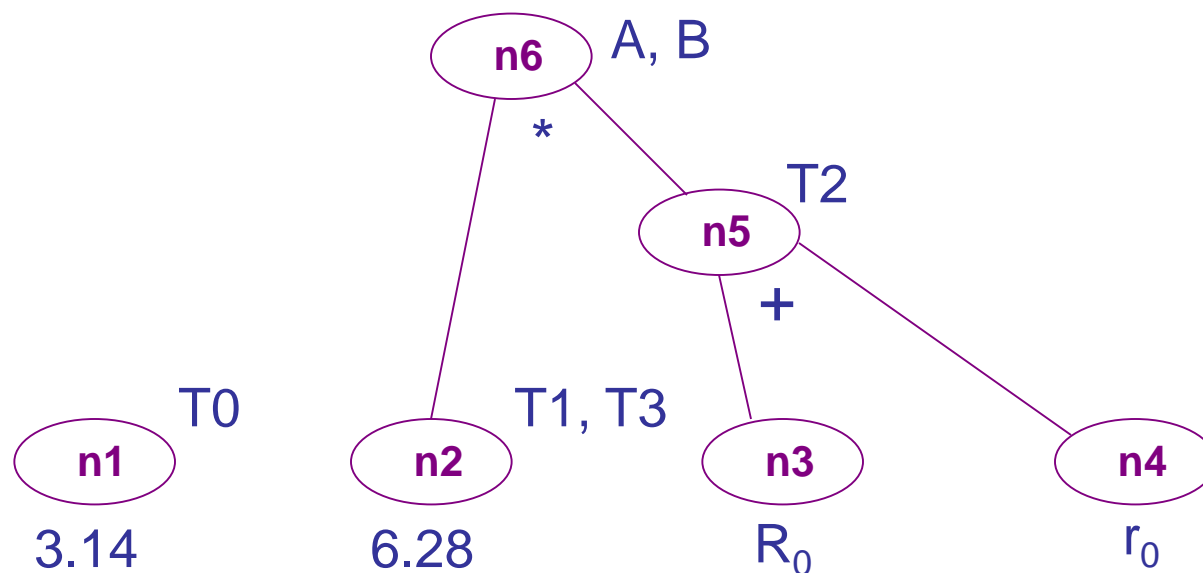
$T3:=2*T0$

$T4:=R+r$

$T5:=T3*T4$

$T6:=R-r$

$B:=T5*T6$



基本块的 DAG 表示

◇ 基本块 DAG 表示的构造

— 举例

$T0:=3.14$

$T1:=2*T0$

$T2:=R+r$

$A:=T1*T2$

$B:=A$

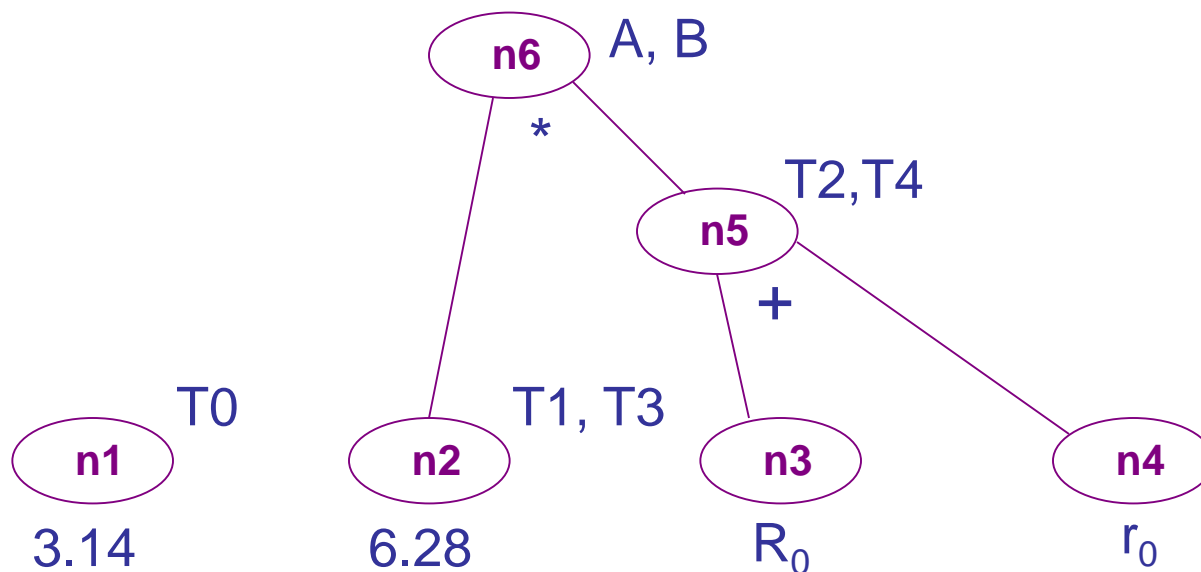
$T3:=2*T0$

$T4:=R+r$

$T5:=T3*T4$ ←

$T6:=R-r$

$B:=T5*T6$



基本块的 DAG 表示

◇ 基本块 DAG 表示的构造

— 举例

$T0:=3.14$

$T1:=2*T0$

$T2:=R+r$

$A:=T1*T2$

$B:=A$

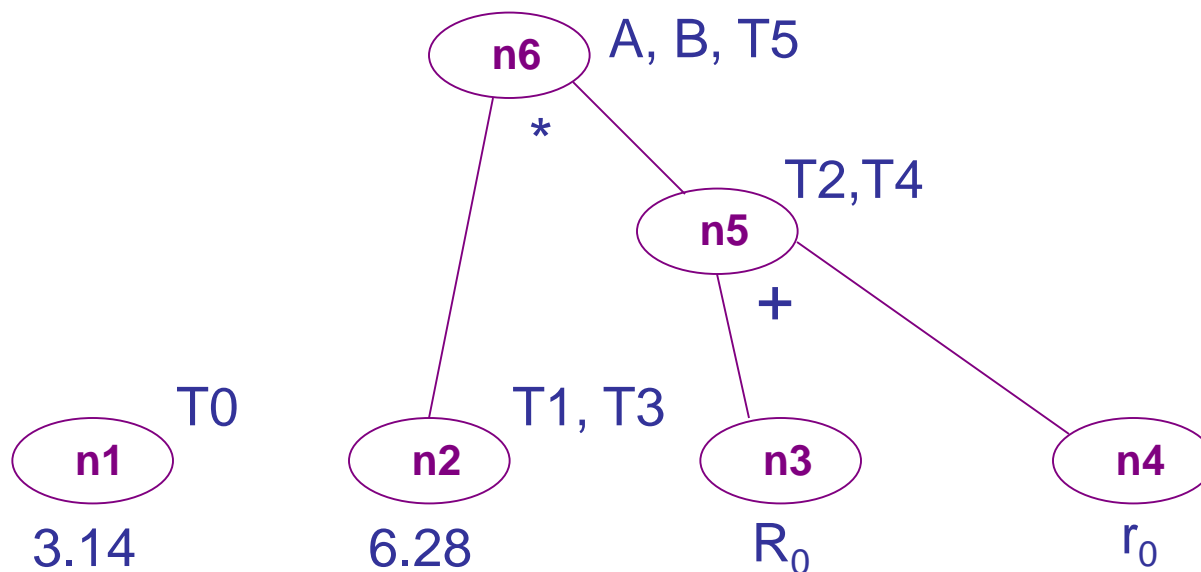
$T3:=2*T0$

$T4:=R+r$

$T5:=T3*T4$

$T6:=R-r$

$B:=T5*T6$



基本块的 DAG 表示

◇ 基本块 DAG 表示的构造

— 举例

$T0:=3.14$

$T1:=2*T0$

$T2:=R+r$

$A:=T1*T2$

$B:=A$

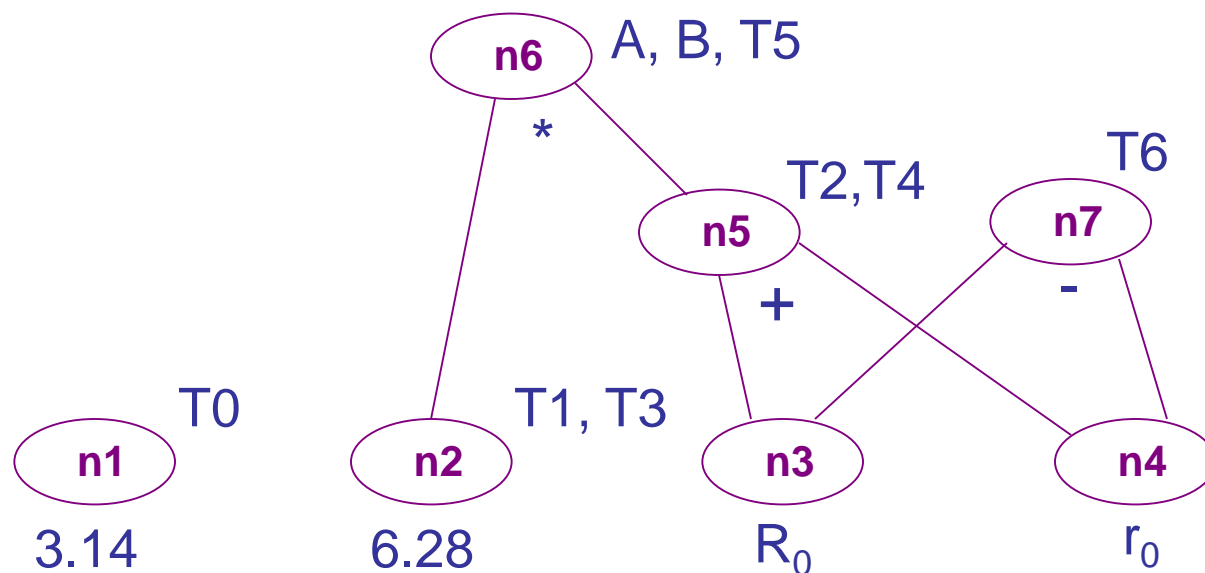
$T3:=2*T0$

$T4:=R+r$

$T5:=T3*T4$

$T6:=R-r$

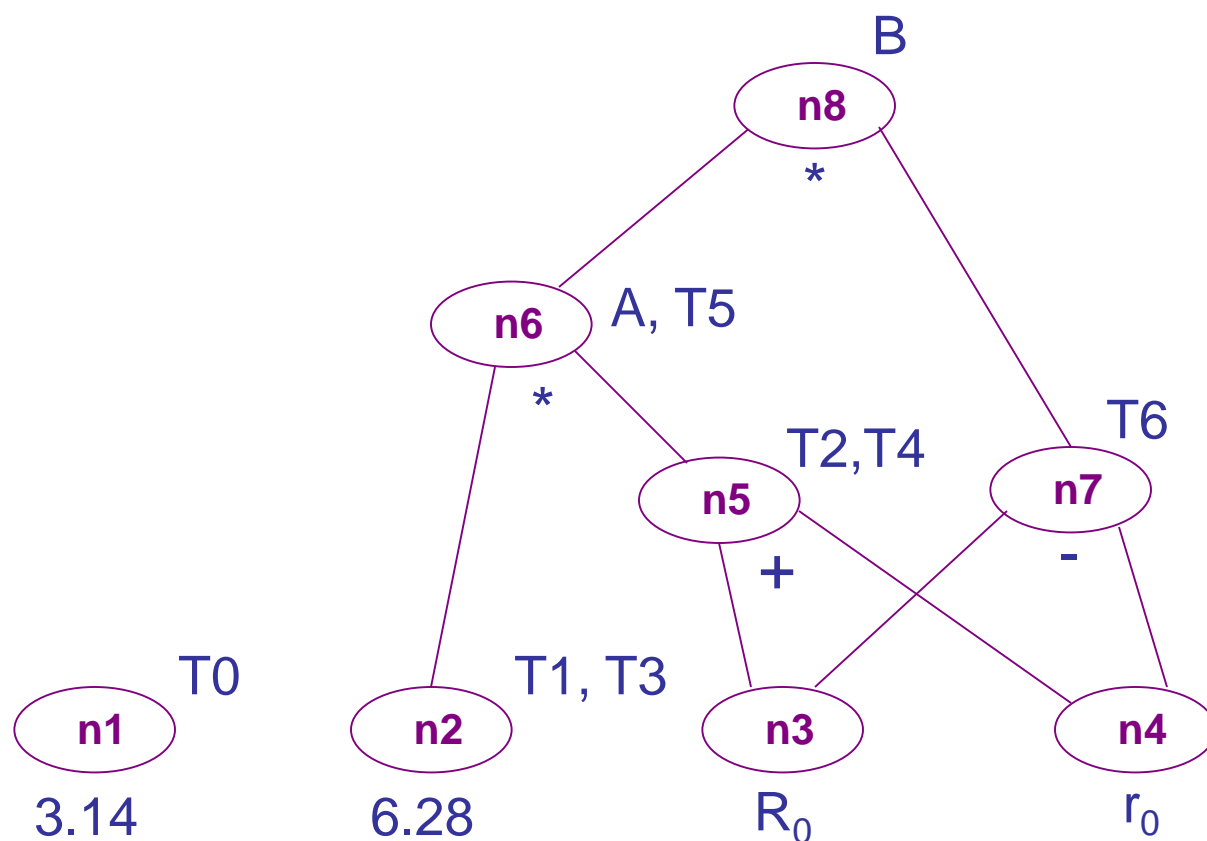
$B:=T5*T6$



基本块的 DAG 表示

◇ 基本块 DAG 表示的构造

— 举例



$T0 := 3.14$

$T1 := 2 * T0$

$T2 := R + r$

$A := T1 * T2$

$B := A$

$T3 := 2 * T0$

$T4 := R + r$

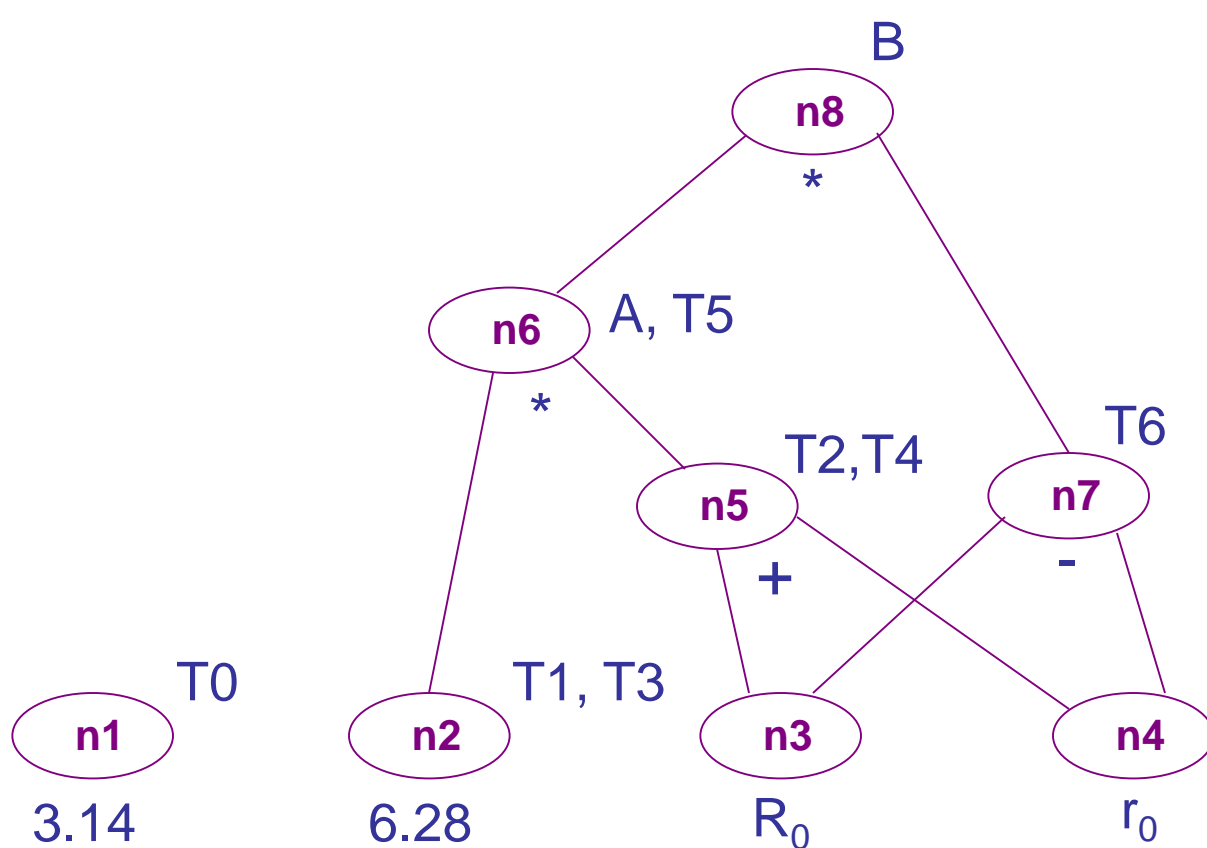
$T5 := T3 * T4$

$T6 := R - r$

$B := T5 * T6$

基本块的 DAG 表示

- ◇ 从基本块的 DAG 表示可得到等价的基本块
 - 举例：从下图的 DAG 可得到右边的新的基本块（经拓扑排序及添加适当的复写语句）



T0:=3.14
T1:=6.28
T3:=6.28
T2:=R+r
T4:=T2
A:=6.28*T2
T5:=A
T6:=R-r
B:=A*T6

基本块的 DAG 表示

- ◇ 从基本块的 DAG 表示可得到等价的基本块
 - 比较变换前后的基本块

T0:=3.14

T1:=2*T0

T2:=R+r

A:=T1*T2

B:=A

T3:=2*T0

T4:=R+r

T5:=T3*T4

T6:=R-r

B:=T5*T6



T0:=3.14

T1:=6.28

T3:=6.28

T2:=R+r

T4:=T2

A:=6.28*T2

T5:=A

T6:=R-r

B:=A*T6

所作的优化

合并已知量

删除多余运算

删除无用赋值

复写传播

- ◇ 代码生成要考虑的主要问题
- ◇ 一个简单的代码生成算法
- ◇ 高效使用寄存器
- ◇ 简单的图着色物理寄存器分配算法

◇ 代码生成要考虑的主要问题

— 指令选择 (*instruction selection*)

目标机指令集的性质和中间代码的形式决定
指令选择的难易

— 寄存器分配 (*register allocation*)

充分、高效地使用寄存器

— 指令调度 (*code scheduling*)

选择好计算的次序，充分利用目标机的特点

◇ 指令选择

— 任务

为每条中间语言语句选择恰当的目标机指令或指令序列

— 原则

- 首先要保证语义的一致性；若目标机指令系统比较完备，为中间语言语句找到语义一致的指令序列模板是很直接的（不必考虑执行效率的情形下）
- 其次要权衡所生成代码的效率（考虑时间/空间代价）
这一点较难做到，因为执行效率往往与该语句的上下文以及目标机体系结构（如流水线）有关

◇ 指令选择举例

— 为 TAC 语句选择指令模板

例 TAC 语句 $a := b + c$ 可转换为如下代码序列

MOV	b, R_0	/* b 装入寄存器 R_0 */
ADD	R_0 , c	/* c 加到 R_0 */
MOV	R_0 , a	/* 存 R_0 到 a */

(其他算术和逻辑运算的 TAC 语句与此类似, 只是选择不同的目标指令, 如减运算选择指令 “SUB”, ...)

◇ 指令选择

— 选择指令模板时可考虑指令执行的代价 (cost)

如：考虑因不同的寻址方式所附加的指令执行代价

假设每条指令在操作数准备好后执行其操作的代价均为1，而是否会有附加的代价则要视获取操作数时是否访问内存而定，每访问一次内存则增加代价1。

◇ 指令选择

— 选择指令模板时可考虑指令执行的代价 (cost)

例 TAC 语句 $a:=b+c$ 的几种转换方式

(1) MOV b, R_0

ADD R_0 , c

MOV R_0 , a cost=6

(2) MOV b, a

ADD a, c cost=6

(3) 假定 R_1 和 R_2 中已经分别
包含了 b 和 c 的值

MOV R_1 , R_0

ADD R_0 , R_2

MOV R_0 , a cost=4

(4) 假定 R_1 和 R_2 中分别包含 b
和 c 的值, 并且 b 的值在这个
赋值以后不再需要

ADD R_1 , R_2

MOV R_1 , a cost=3

◇ 简单的代码生成算法

— 基本块内 TAC 语句序列的简单代码生成

- 在基本块范围内考虑如何充分利用寄存器的问題

原则：尽可能地让变量的值或计算结果保留在寄存器中，必要时才将其Spill到内存

尽可能引用变量在寄存器中的值（对于CISC目标机）

保证语义一致的前提下，尽可能在取值相同的变量间共享寄存器

不再被引用的变量所占用的寄存器应尽早释放

.....

- 借助于在基本块范围内建立变量的待用信息链和活跃信息链

◇ 简单的代码生成算法

— 基本块内 TAC 语句

假设仅包含 $A := B \text{ op } C$ 和 $A := B$ 两类语句

假定 A、B 和 C 均为变量（易扩展至含有常量的情形）

— 目标语言 假设生成目标代码时仅用到下列指令：

- OP reg0, reg1, reg2 /* reg1 和 reg2 进行 OP 运算，结果存入 reg0
- LD reg, mem /* Load 指令，取内存量 mem 的值到寄存器 reg */
- ST reg, mem /* Store 指令，存寄存器 reg 的值到内存量 mem */
- MV reg0, reg1 /* Move 指令，将寄存器 reg1 的内容复制寄存器 reg0 */

◇ 简单的代码生成算法

— 简单代码生成算法的一个例子

与 MiniDecaf 实验框架 (C++版) 代码生成算法相类似

— 简单代码生成算法的另一个例子 (仅供课后参考)

与龙书中代码生成算法类似

◇ 简单代码生成算法的一个例子

— 寄存器描述数组及变量位置查询函数

- $VALUE[R]$ 描述寄存器 R 当前存放哪一个变量
- $lookupReg(V)$ 表示查询变量 V 存放在哪个寄存器中，若不存放在寄存器中则返回 $NULL$

— 寄存器分配函数

- $allocReg(V)$ 表示为变量 V 分配新的寄存器

◇ 简单代码生成算法的一个例子

— 基本块内 TAC 语句序列的简单代码生成 (假设只有形如 $A := B \text{ op } C$ 和 $A := B$ 的 TAC 语句序列)

step1: 对每个 TAC 语句, 依次执行下述步骤:

- 调用 $\text{lookupReg}(A)$ 确定目的寄存器, 若 A 已分配寄存器, 则使用已分配的寄存器, 命名为 A' ; 若未分配, 则调用 $\text{allocReg}(A)$ 为变量 A 分配新的寄存器, 同命名为 A' 。
- 调用 $\text{lookupReg}(B)$ 和 $\text{lookupReg}(C)$, 确定 B 和 C 现行值存放位置; 如果其现行值在寄存器中, 则把寄存器取作 B' 和 C' ; 如果 B (C) 的现行值不在寄存器中, 则调用 $\text{allocReg}(B)$ ($\text{allocReg}(C)$) 为 B (C) 分配寄存器 B' (C'), 且使用 LD 指令将 B (C) 从内存中取出, 存到寄存器 B' (C') 中。
- 按照上述寄存器分配结果更新 $\text{VALUE}[A']$ 、 $\text{VALUE}[B']$ 、 $\text{VALUE}[C']$ 。

◇ 简单代码生成算法的一个例子

— 基本块内 TAC 语句序列的简单代码生成（续前页）

- 分以下情形生成目标代码：

(a) 对于语句 $A := B \text{ op } C$ ，分情况生成如下代码序列：

若 B 和 C 不在寄存器中，则生成

```
LD B`, B
LD C`, C
OP A`, B`, C`
```

若 B 和 C 中有一个在寄存器中，则参考以上目标代码，去掉对应的 LD 指令即可

- 若 B 和 C 都在寄存器中，则生成

```
OP A`, B`, C`
```

◇ 简单代码生成算法的一个例子

— 基本块内 TAC 语句序列的简单代码生成（续前页）

- 分以下情形生成目标代码：

(b) 对于语句 $A := B$ ，分情况生成如下代码序列：

若 B 在寄存器中，则生成

$MV\ A', B'$

若 B 不在寄存器中，则生成

$LD\ B', B$

$MV\ A', B'$

◇ 简单代码生成算法的一个例子

— 基本块内 TAC 语句序列的简单代码生成（续前页）

- 如 $B(C)$ 的现行值在基本块中不再被引用，也不是基本块出口之后的活跃变量（由语句上的附加信息得知），且其现行值在某个寄存器 R_k 中，则删除 $VALUE[R_k]$ 中的 $B(C)$ ，同时使该寄存器不再被 $B(C)$ 所占用

step2: 处理完基本块中所有 TAC 语句之后，对现行值在某寄存器 R_i 中的每个变量 M_i ，若它在出口之后是活跃的，则生成 $ST\ R_i, M_i$ ，将其存入内存

◇ 简单代码生成算法的一个例子

- 函数 allocReg 以变量 V 为参数，返回一个寄存器
(设 $\{R_k\}$ 为当前所有寄存器的集合)

步骤:

- 若 $\{R_k\}$ 中存在未被使用的寄存器，则直接返回任一未被使用的 R_i ，且置 $VALUE[R_i] = V$
- 若 $\{R_k\}$ 中不存在未被使用的寄存器，即所有寄存器都被占用：

若 $\{VALUE[R_k]\}$ 中存在不再被该基本块引用的变量 $VALUE[R_i]$ ，且该变量不是该基本块出口之后的活跃变量，则返回该变量对应的寄存器 R_i ，且令 $VALUE[R_i] = V$

若 $\{VALUE[R_k]\}$ 中不存在上述“非活跃”变量，则选择从分析该基本块到当前时刻最久未被使用的寄存器 R_i (LRU 算法)，使用 ST 指令将该寄存器中存储的变量存入内存，返回 R_i 作为新的可使用的寄存器，且令 $VALUE[R_i] = V$

◇ 简单代码生成算法的一个例子

- 对于右图的基本块（假定b和d在基本块的出口是活跃的），上述算法可生成如下代码序列：

t := a - b
a := b
u := a - c
v := t + u
d := v + u

语句	生成的代码	说明
t := a - b	LD R ₁ , a LD R ₂ , b SUB R ₀ , R ₁ , R ₂	VALUE[R ₀]=t VALUE[R ₂]=b
a := b	MV R ₁ , R ₂	VALUE[R ₀]=t VALUE[R ₁]=a VALUE[R ₂]=b
u := a - c	LD R ₄ , c SUB R ₃ , R ₁ , R ₄	VALUE[R ₀]=t VALUE[R ₂]=b VALUE[R ₃]=u
v := t + u	ADD R ₁ , R ₀ , R ₃	VALUE[R ₁]=v VALUE[R ₂]=b VALUE[R ₃]=u
d := v + u	ADD R ₀ , R ₁ , R ₃ ST R ₀ , d ST R ₂ , b	VALUE[R ₀]=d VALUE[R ₂]=b 将活跃变量b和d存入内存

◇ 简单代码生成算法的另一个例子

- 对于右图的基本块（假定b和d在基本块的出口是活跃的），上述算法可生成如下代码序列：

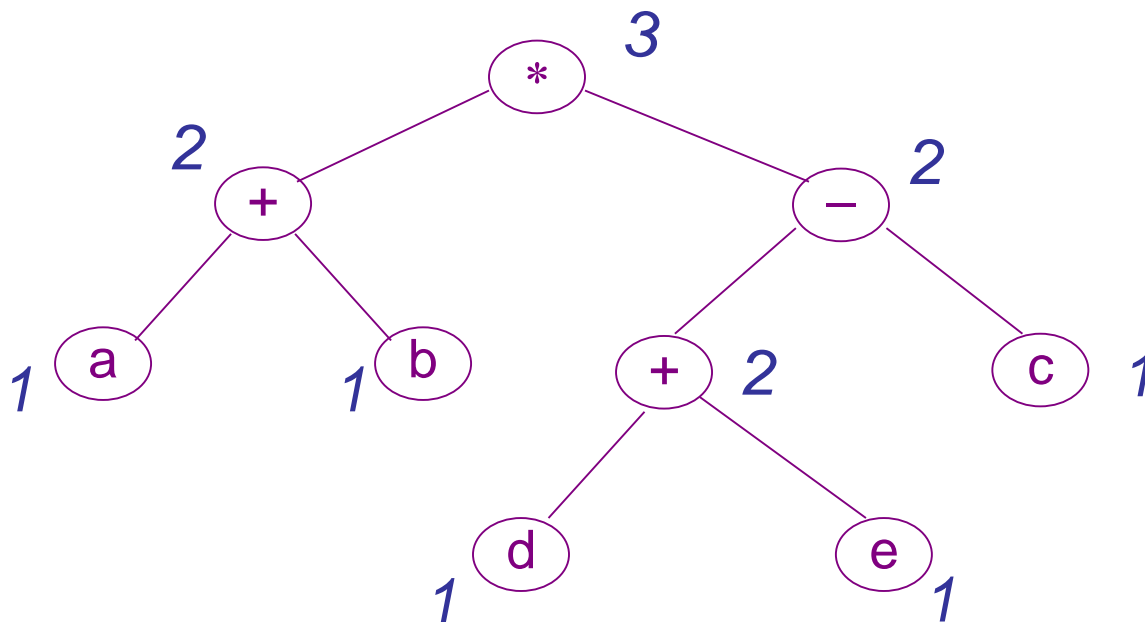
$t := a - b$
 $a := b$
 $u := a - c$
 $v := t + u$
 $d := v + u$

语句	生成的代码	寄存器描述	变量地址描述
$t := a - b$	LD R_0, a LD R_1, b SUB R_0, R_0, R_1	R_0 包含: t R_1 包含: b	a 在: a b 在: b, R_1 t 在: R_0
$a := b$		R_0 包含: t R_1 包含: b, a	a 在: R_1 b 在: b, R_1 t 在: R_0
$u := a - c$	LD R_2, c SUB R_2, R_1, R_2	R_0 包含: t R_1 包含: b, a R_2 包含: u	b 在: b, R_1 t 在: R_0 u 在: R_2
$v := t + u$	ADD R_0, R_0, R_2	R_0 包含: v R_1 包含: b, a R_2 包含: u	b 在: b, R_1 v 在: R_0 u 在: R_2
$d := v + u$	ADD R_0, R_0, R_2 ST R_0, d	R_0 包含: v R_1 包含: b, a R_2 包含: u	b 在: b, R_1 u 在: R_2

◇ 高效使用寄存器

— 表达式树的求值

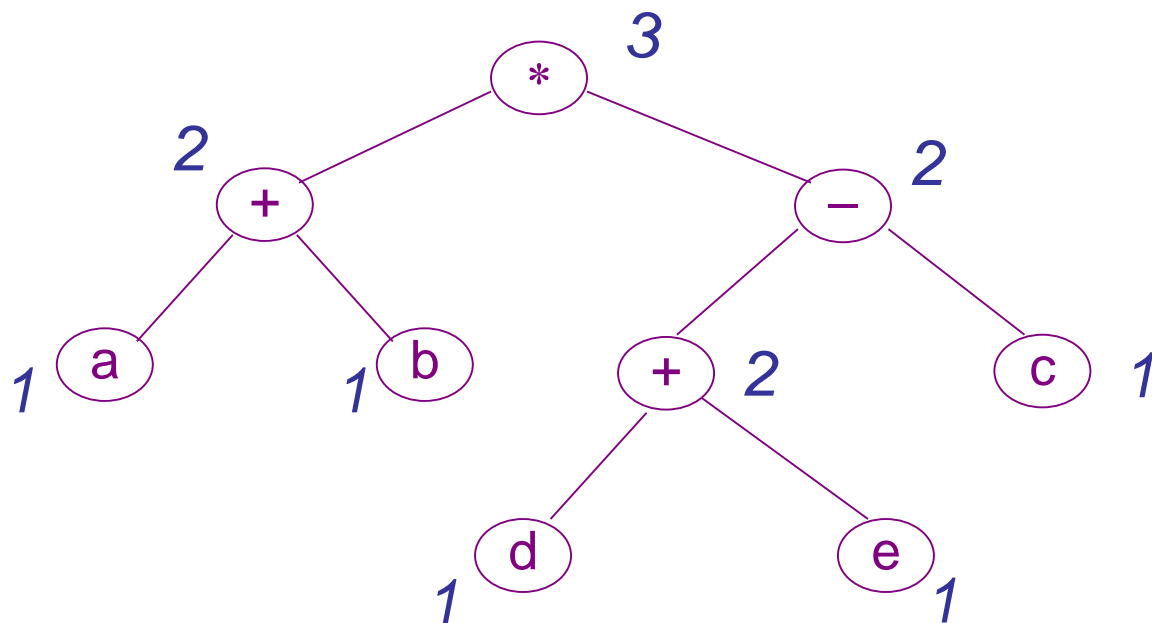
Ershov 数：表达式求值时所需寄存器数目的最小值



◇ 高效使用寄存器

— 表达式树的求值

Sethi-Ullman 算法



```
LD    R0, a
LD    R1, b
ADD   R0, R0, R1
LD    R1, d
LD    R2, e
ADD   R1, R1, R2
LD    R2, c
SUB   R1, R1, R2
MUL   R0, R0, R1
```

☆ 简单的图着色物理寄存器分配算法

— 两遍的寄存器分配和指派算法

- 第一遍先假定可用的通用寄存器是无限数量的，完成指令选择和生成
- 第二遍将物理寄存器指派到伪寄存器

物理寄存器数量不足时，会将一些伪寄存器泄露到 (*spilled into*) 内存，图着色算法的核心任务是使得泄露的伪寄存器数目最少

◇ 简单的图着色物理寄存器分配算法

— 基于寄存器相干图 (*register-interference graph*) 的图着色寄存器分配算法

- 构造寄存器相干图

结点：每一个伪寄存器为一个结点

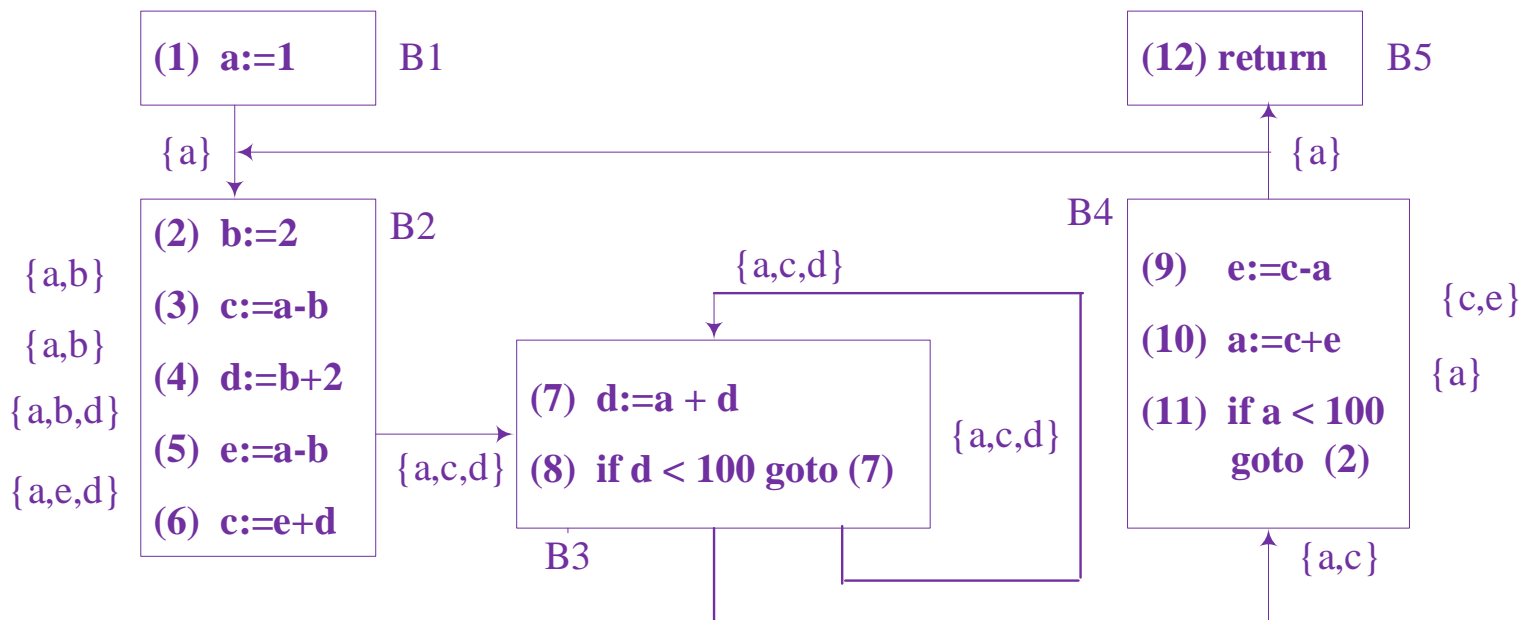
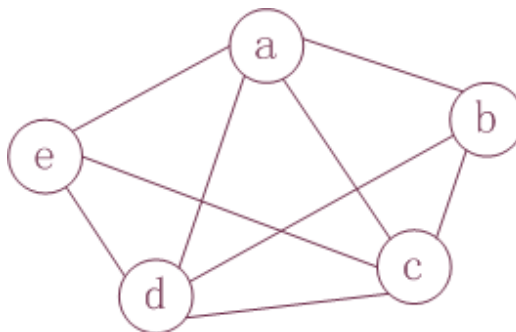
边：如果程序中存在某点，一个结点在该点被定义，而另一个结点在紧靠该定值之后的点是活跃的，则在这两个结点间连一条边

- 对相干图进行着色 (*coloring*)

使用 k （物理寄存器数量）种颜色对相干图进行着色，使任何相邻的结点具有不同的颜色（即两个相干的伪寄存器不会分配到同一个物理寄存器）

✧ 简单的图着色物理寄存器分配算法

— 基于寄存器相干图



◇ 简单的图着色物理寄存器分配算法

— 一种启发式图着色算法

“一个图是否能用 k 种颜色着色”是 NP -完全问题

以下是一个简单的启发式 k -着色算法：

- 假设图 G 中某个结点 n 的度数小于 k ，从 G 中删除 n 及其邻边得到图 G' ，对 G 的 k -着色问题可转化为先对 G' k -着色，然后给结点 n 分配一个其相邻结点在 G' 的 k -着色中没有使用过的颜色

重复这个过程从图中删除度数小于 k 的结点，如果可以到达一个空图，说明对原图可以成功实现 k -着色；否则，原图不能成功实现 k -着色，可从 G 中选择某个结点作为泄露候选，将其删除，算法可继续

代码优化技术

◇ 简单的归类

— 依优化范围划分

- 窥孔优化 (*peephole optimization*)

局部的几条指令范围内的优化

- 局部优化

基本块范围内的优化

- 全局优化

流图范围内的优化

- 过程间优化

整个程序范围内的优化

◇ 简单的归类

— 依优化对象划分

- 目标代码优化

面向目标代码

- 中间代码优化

面向程序的中间表示

- 源级优化

面向源程序

◇ 简单的归类

— 依优化侧面划分

- 指令调度
- 寄存器分配
- 存储层次优化
- 存储布局优化
- 循环优化
- 控制流优化
- 过程优化
-

✧ 窥孔优化 (*peephole optimization*)

- **工作方式** 在目标指令序列上滑动一个包含几条指令的窗口（称为窥孔），发现其中不够优化的指令序列，用一段更短或更有效的指令序列来替代它，使整个代码得到改进

✧ 窥孔优化 (*peephole optimization*)

— 举例

- 删除冗余的“取”和“存” (*redundant loads and stores*)

指令序列

- (1) LD R0, a
- (2) LD R1, b
- (3) MV R1, R0

可优化为

- (1) LD R0, a
- (2) MV R1, R0

✧ 窥孔优化 (*peephole optimization*)

— 举例

- 合并已知量 (*constant folding*)

代码序列

(1) $r2 := 3 * 2$

可优化为

(1) $r2 := 6$

☆ 窥孔优化 (*peephole optimization*)

— 举例

- 常量传播 (*constants propagating*)

代码序列

```
(1)  r2:=4  
(2)  r3:=r1+r2
```

可优化为

```
(1)  r2:=4  
(2)  r3:=r1+ 4
```

注：虽然条数未少，但若是知道r2不再活跃时，可删除 (1)

◇ 窥孔优化 (*peephole optimization*)

— 举例

- 代数化简 (*algebraic simplification*)

代码序列

```
(1)  x:=x+0  
(2)  .....  
(n)  y:=y*1
```

中的 (1) , (n) 可在窥孔优化时删除

◇ 窥孔优化 (*peephole optimization*)

— 举例

- 控制流优化 (*flow-of-control optimization*)

代码序列

```
goto L1
.....
L1: goto L2
```

可替换为

```
goto L2
.....
L1: goto L2
```

◇ 窥孔优化 (*peephole optimization*)

— 举例

- 死代码删除 (*dead-code elimination*)

代码序列

```
debug := false  
if (debug) print ...  
.....
```

可替换为

```
debug := false  
.....
```

◇ 窥孔优化 (*peephole optimization*)

— 举例

- 强度削弱 (*reduction in strength*)

$x := 2.0 * f$ 可替换为 $x := f + f$

$x := f / 2.0$ 可替换为 $x := f * 0.5$

◇ 窥孔优化 (*peephole optimization*)

— 举例

- 使用目标机惯用指令 (*use of idioms*)

某个操作数与1相加，通常用“加1”指令，而不是用“加”指令

某个定点数乘以2，可以采用“左移”指令；而除以2，则可以采用“右移”指令

...

◇ 基本块内的优化

— 举例

- DAG 的构造过程中已经进行过一些基本块内的优化
 - 合并已知量
 - 删除多余运算（公共表达式删除）
 - 删除无用赋值

◇ 全局优化 (*global optimization*)

— 借助于针对流图的数据流分析进行的优化

- 例

全局公共表达式删除

全局死代码删除 (删除从流图入口不能到达的代码)

.....

◇ 循环优化 (*loop optimization*)

— 举例

- 代码外提 (*code motion*)

```
while (i < limit/2) {...}
```

等价于 $t := \text{limit}/2;$

```
while (i < t) {...}
```

- 循环不变量 (*loop-invariant*) 代码可以外提

借助于 UD 链可以查找循环不变量，如对于循环内部的语句 $x := y + z$ ，若 y 和 z 的定值点都在循环外，则 $x := y + z$ 为循环不变量

◇ 循环优化 (loop optimization)

— 举例

- 循环不变量代码外提

循环不变量代码 $x := y + z$ 可以外提的一个充分条件:

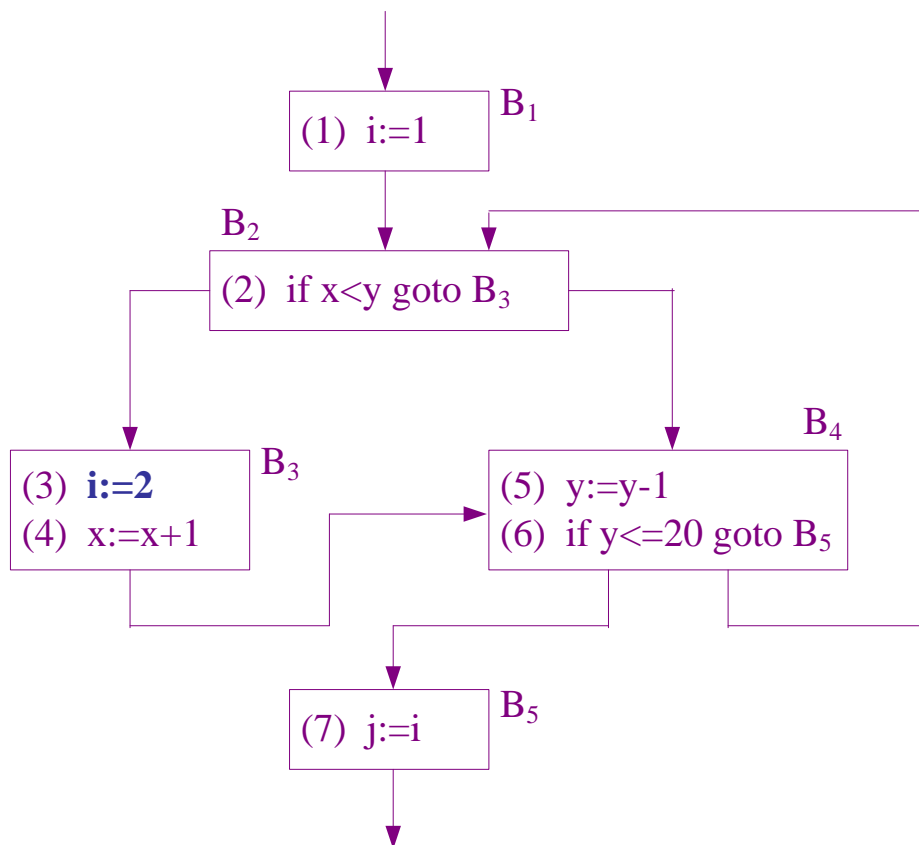
- 1) 所在结点是循环的所有出口结点的支配结点
- 2) 循环中其它地方不再有 x 的定值点
- 3) 循环中 x 的所有引用点都是且仅是这个定值所能达到的
- 4) 若 y 或 z 是在循环中定值的, 则只有当这些定值点的语句 (一定也是循环不变量) 已经被执行过代码外提

或者, 在满足上述第 2、3 和 4 条的前提下, 将第 1 条替换为:

- 5) 要求 x 在离开循环之后不再是活跃的

◇ 循环优化 (loop optimization)

— 举例 下图中循环不变量不符合外提条件



◇ 循环优化 (loop optimization)

— 举例

- 归纳变量 (induction variable) 相关的优化

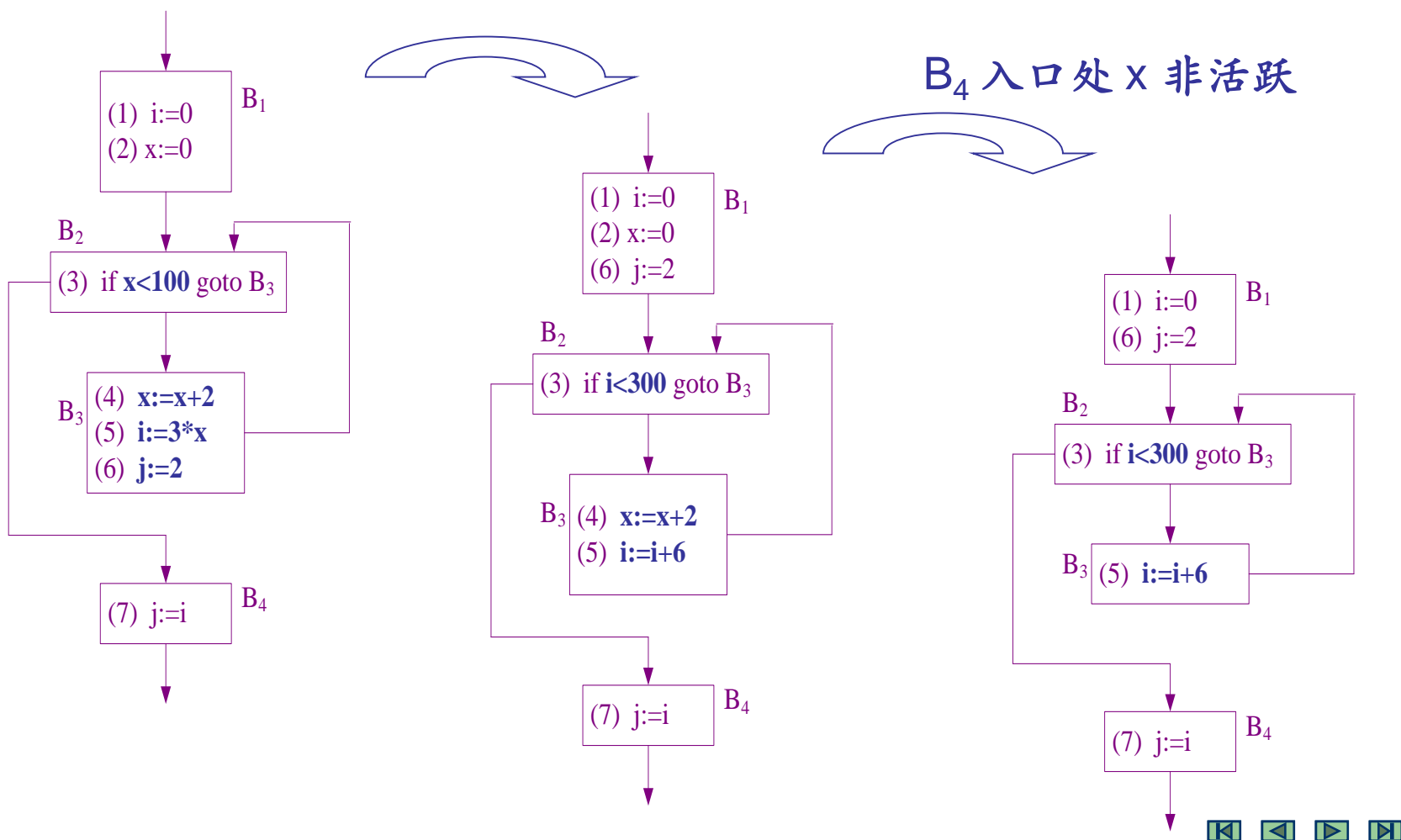
归纳变量是在循环的顺序迭代中取得一系列值的变量

常见的归纳变量如循环下标及循环体内显式增量和减量的变量

通常可以针对归纳变量可以进行如下优化:

- 1) 削弱归纳变量的计算强度
- 2) 因常常可以有冗余的归纳变量, 可以只在寄存器中保存个别归纳变量, 而不是全部. 特别是经强度削弱后, 往往可以删除某些归纳变量

◇ 循环优化举例



课后作业

参见网络学堂公告：“第五次书面作业”

Wish You a Great Success,

Thank You