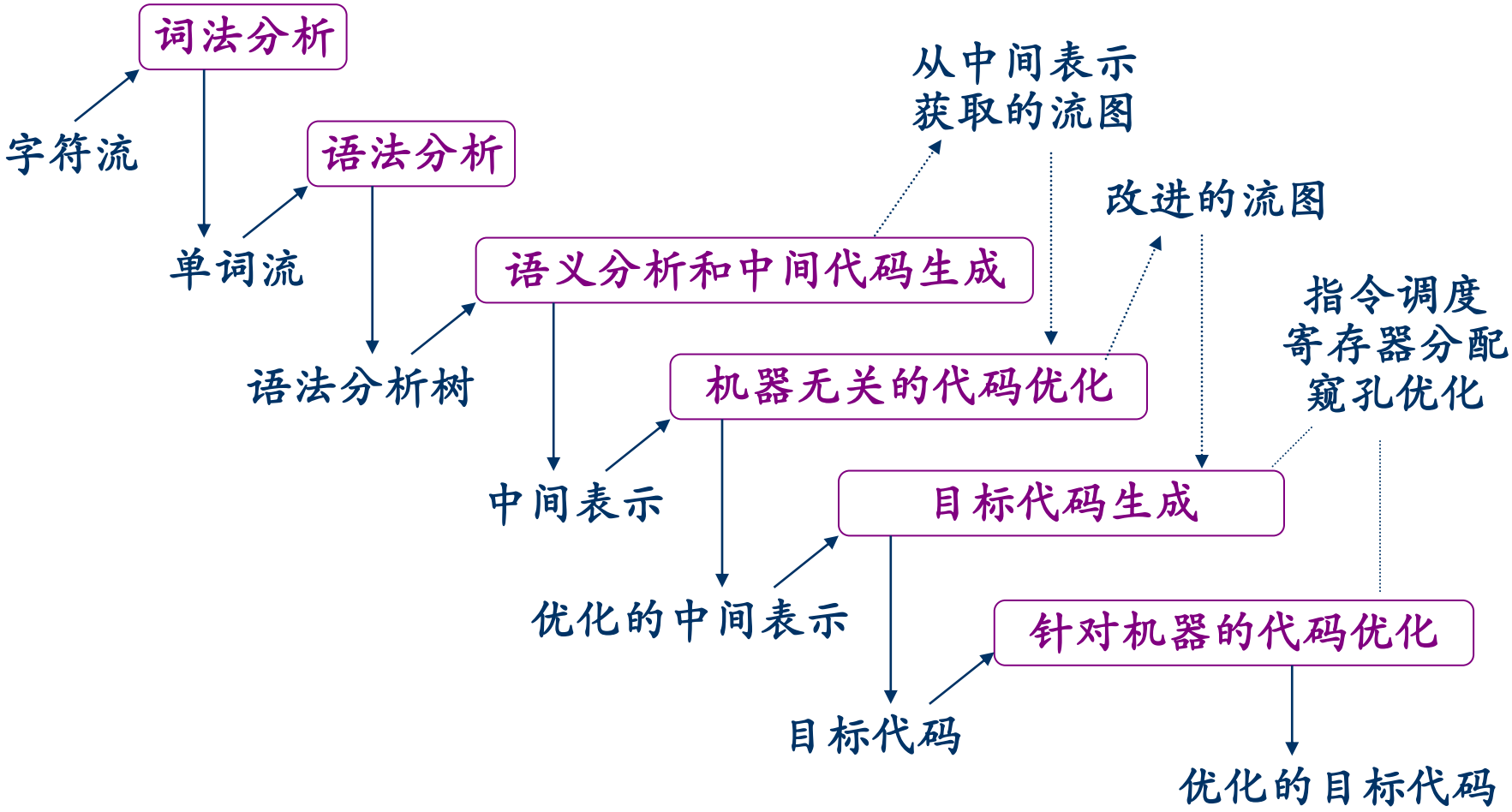
- 目标代码生成及代码优化基础

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

✧ 二者在编译程序中的逻辑位置



基本块、流图和循环

✧ 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

基本块、流图和循环

✧ 划分基本块的算法

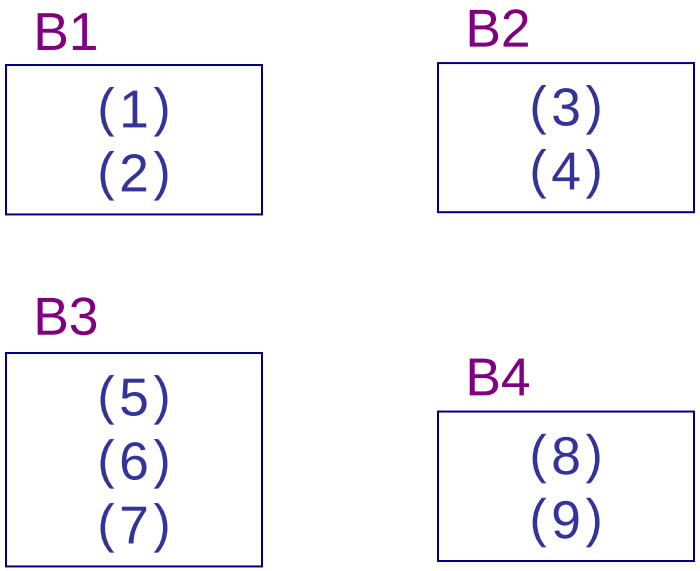
- 针对三地址码（TAC）
- 步骤
  - 求出 TAC 程序之中各个基本块的入口语句
  - 对每一入口语句，构造其所属的基本块。它是由该语句到下一入口语句（不包括下一入口语句），或到一转移语句（包括该转移语句），或到一停语句（包括该停语句）之间的语句序列组成的
  - 凡未被纳入某一基本块的语句，都是程序中控制流程无法到达的语句，因而也是不会被执行到的语句，可以把它们删除

基本块、流图和循环

✧ 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

```
*(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
```



基本块、流图和循环

✧ 流图（*flow graph*）

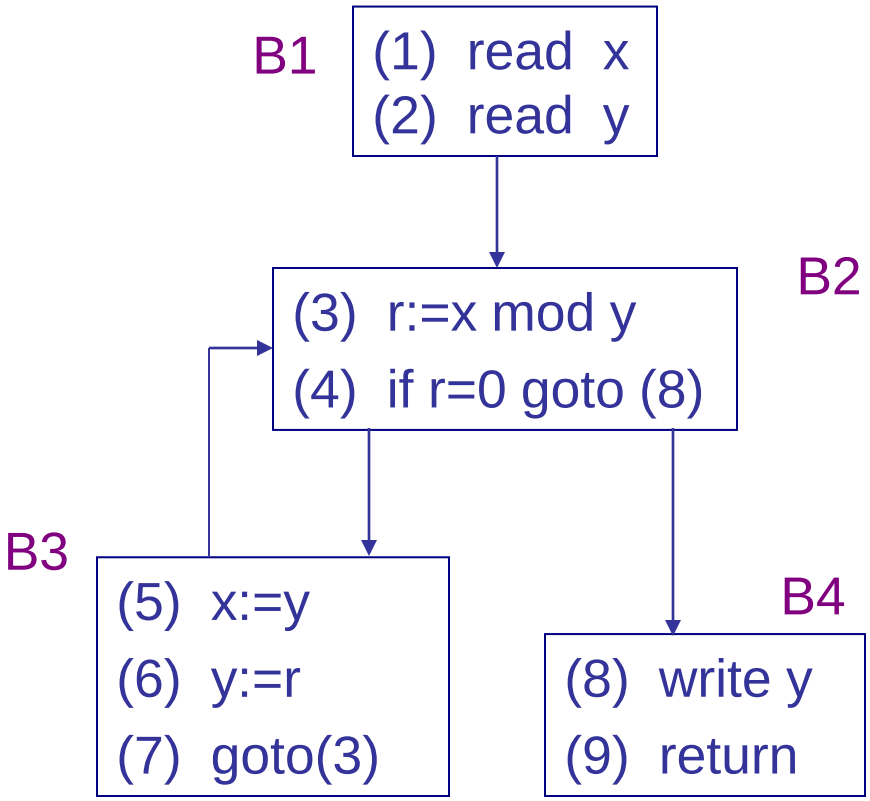
– **概念** 可以为构成程序的基本块增加控制流信息，方法是构造一个有向图，称之为**流图**或**控制流图**（*CFG*，*Control-Flow Graph*）

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 *goto(S)* 或 *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 DOM n*
(∀*a*. *a 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}

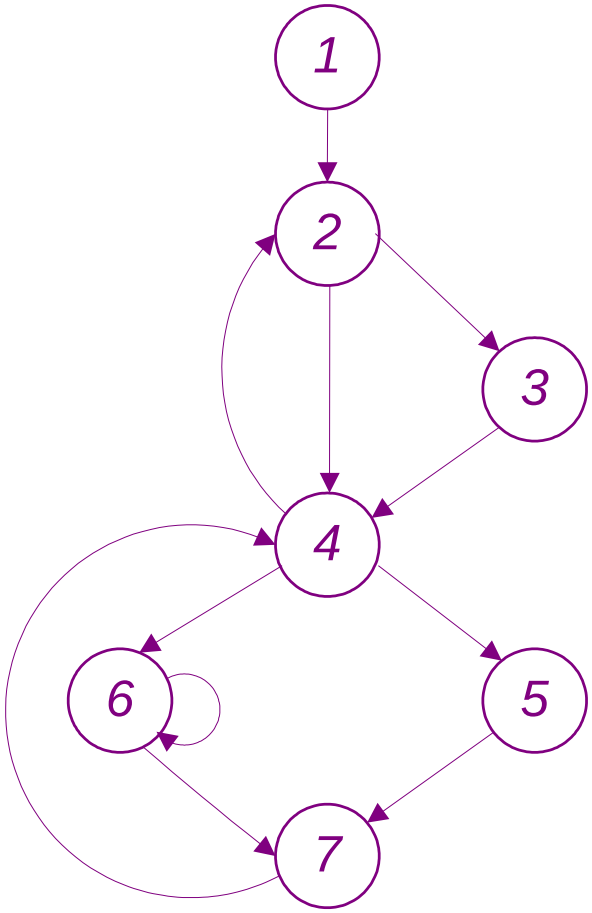
D(3)={1, 2, 3}

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 $n \to d$ 是流图中的一条有向边，如果 d DOM n 则称 $n \to d$ 是流图中的一条**回边**（*back edge*）

有向边 $n \to d$ 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→6：

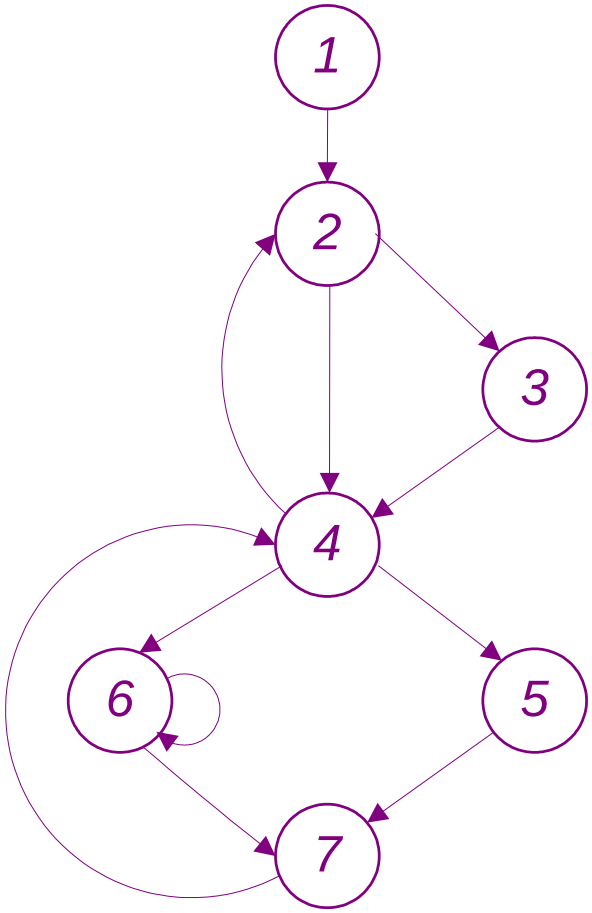
{6}

对应回边 7→4：

{4, 5, 6, 7}

对应回边 4→2：

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



数据流分析基础

✧ 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*）

数据流分析基础

✧ 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

数据流分析基础

✧ 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

数据流分析基础

✧ 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

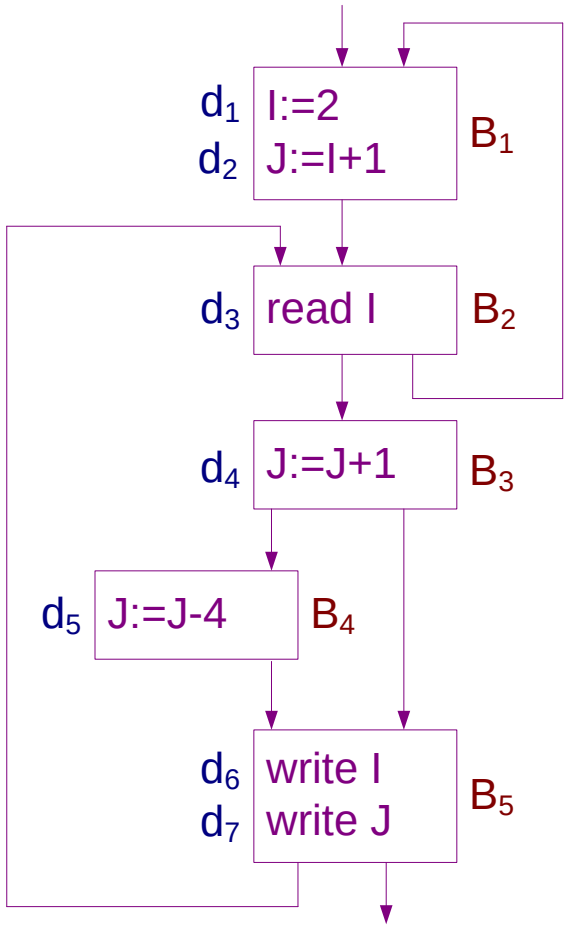
数据流分析基础

✧ 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）

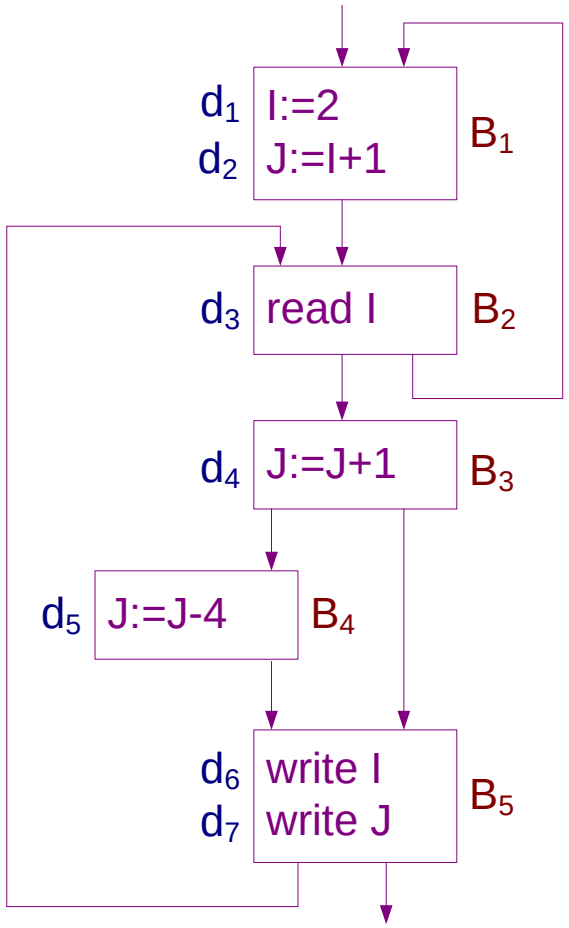
```
for  i:= 1 to n {   IN[ B_i] :=∅;  OUT[B_i ] := GEN[B_i];  }
change := true;
while change  {
      change := false;
      for i := 1 to n  {
            newin := ∪ OUT[p];                //p∈P[B_i]
                if newin ≠ IN[B_i]  {
                     change := ture;     IN[B_i] := newin;
                     OUT[B_i] : = (IN[B_i] – KILL[B_i]) ∪ 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	{ }	{ }	{ }	{ }

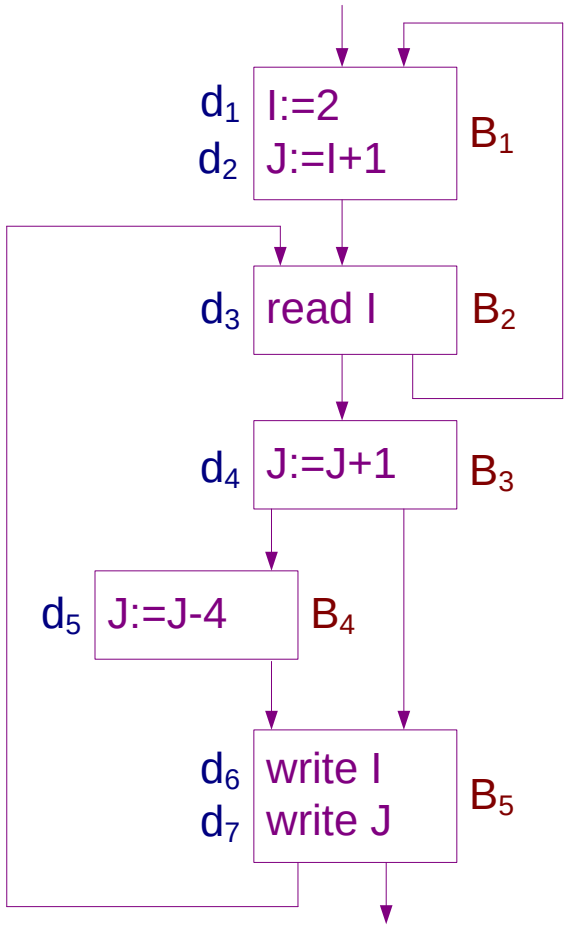


✧ 到达-定值数据流分析

– 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	$\{\ \}$	$\{\ \}$	$\{\ \}$	$\{\ \}$

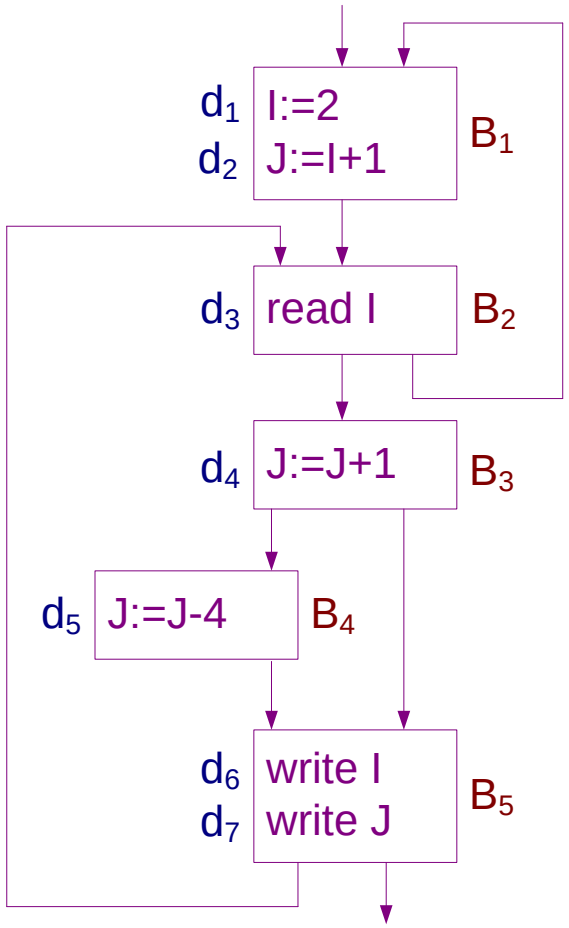


✧ 到达-定值数据流分析

– 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	$\{\ \}$	$\{\ \}$	$\{\ \}$	$\{\ \}$

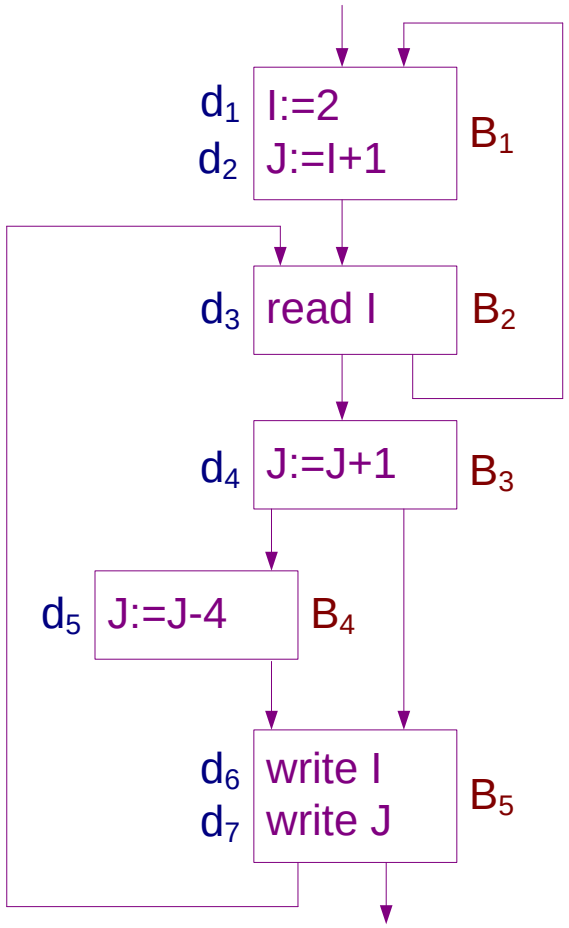


✧ 到达-定值数据流分析

– 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	$\{\ \}$	$\{\ \}$	$\{\ \}$	$\{\ \}$

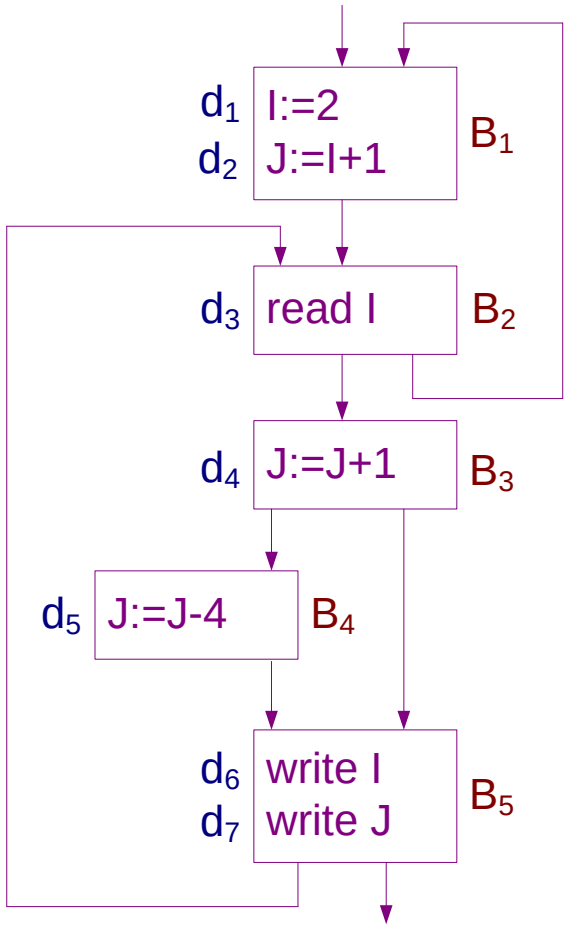


✧ 到达-定值数据流分析

– 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	{ }	{ }	{ }	{ }



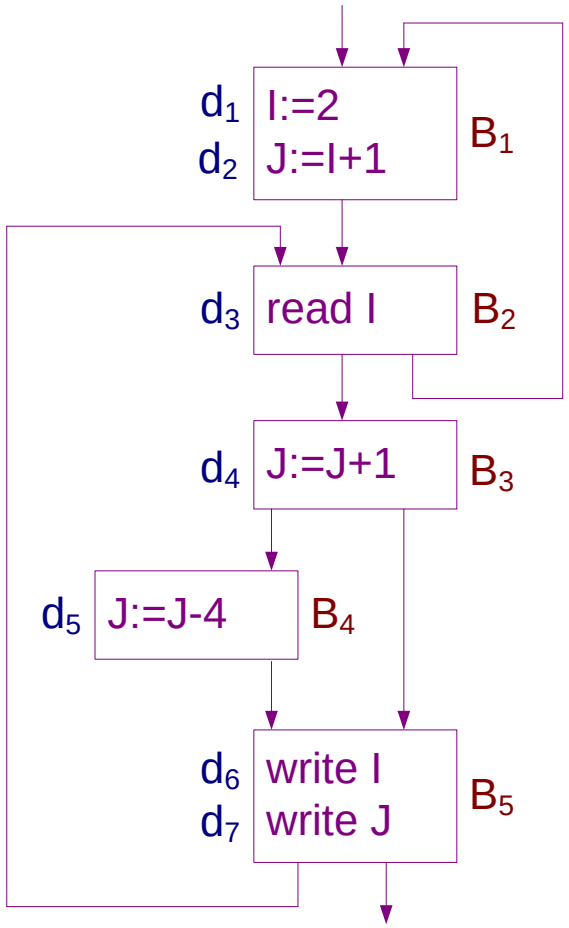
数据流分析基础

✧ 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	$\{\ \}$	$\{\ \}$	$\{d_3,d_4,d_5\}$	$\{d_3,d_4,d_5\}$



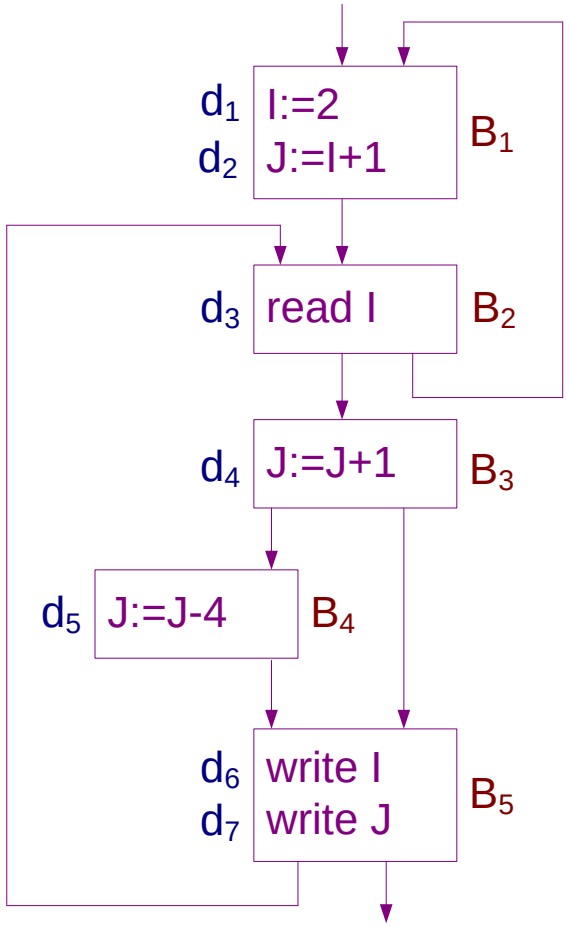
数据流分析基础

✧ 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	$\{\ \}$	$\{\ \}$	$\{d_3,d_4,d_5\}$	$\{d_3,d_4,d_5\}$



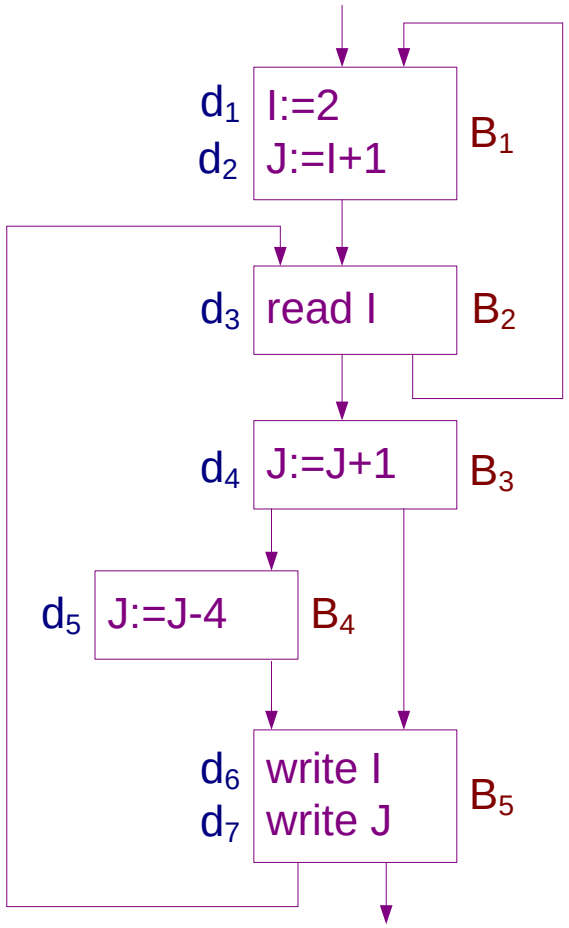
数据流分析基础

✧ 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	$\{\ \}$	$\{\ \}$	$\{d_3,d_4,d_5\}$	$\{d_3,d_4,d_5\}$



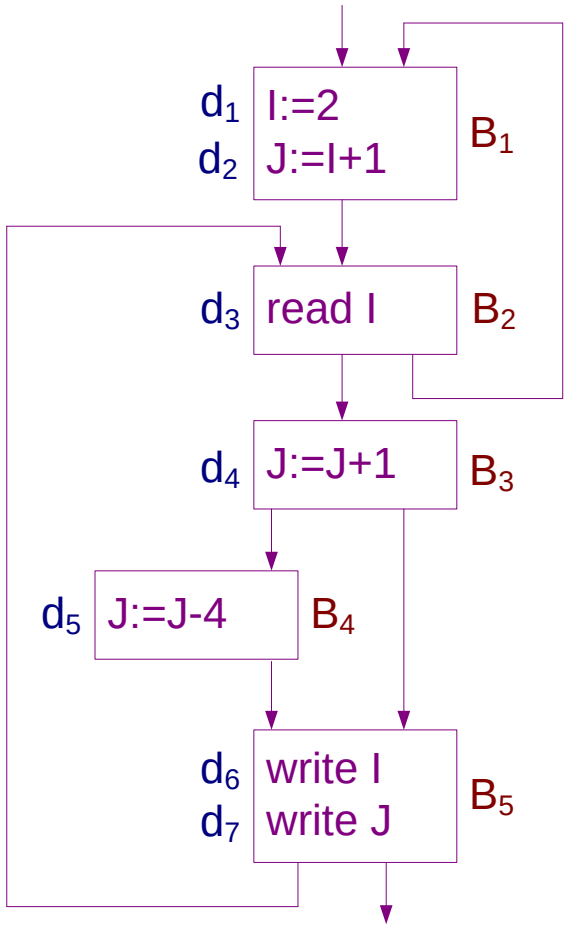
数据流分析基础

✧ 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	$\{\ \}$	$\{\ \}$	$\{d_3,d_4,d_5\}$	$\{d_3,d_4,d_5\}$



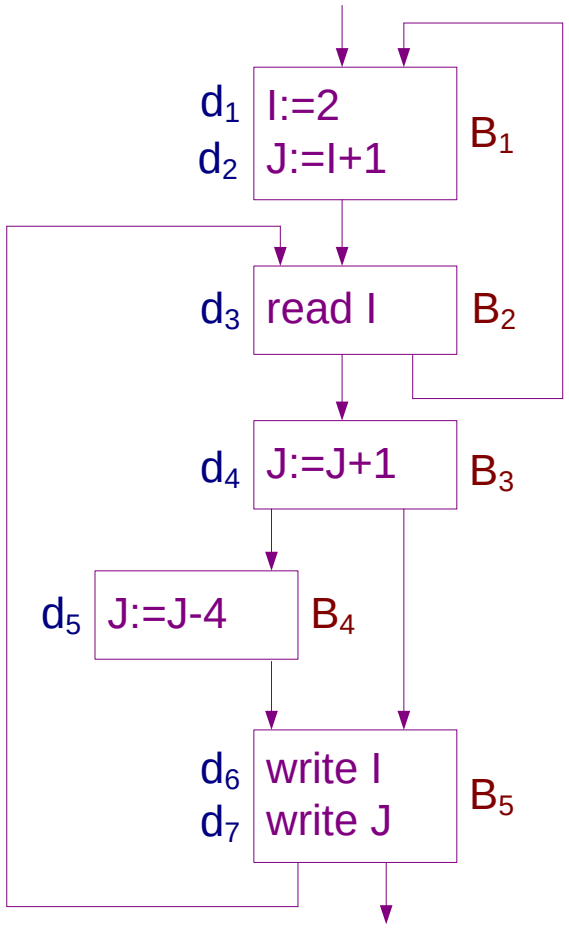
数据流分析基础

✧ 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

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

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

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 ] := ∅;  }
change := true;
while change  {
      change := false;
      for i := 1 to n  {
            newout := ∪ LiveIn[p];                    //p∈S[B_i]
                if newout ≠ LiveOut[B_i]  {
                      change := ture;     LiveOut[B_i] := newout;
                      LiveIn[B_i] : = (LiveOut[B_i] – Def[B_i]) ∪ LiveUse[B_i]
                }
      }
}
```

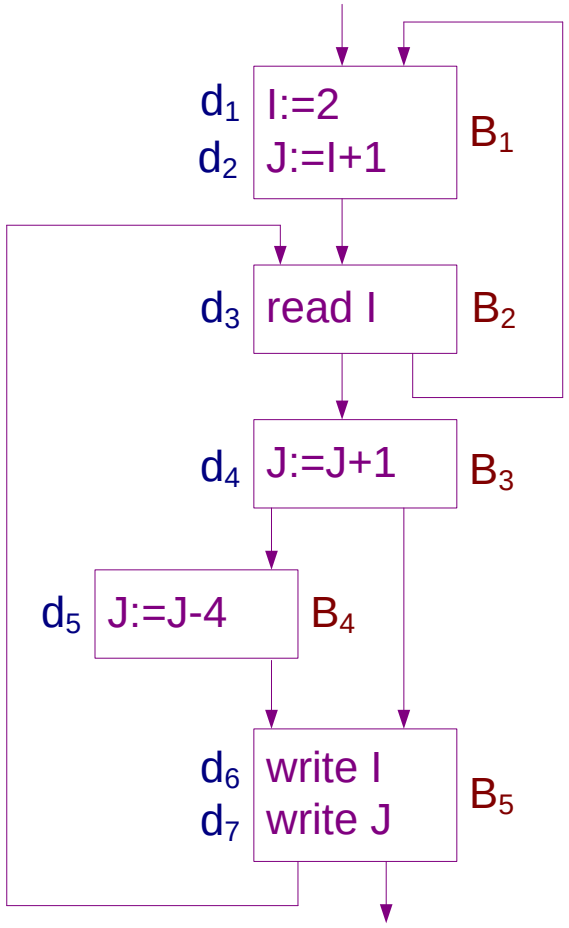
数据流分析基础

✧ 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}	Ø
B_2	{I}	Ø
B_3	Ø	{J}
B_4	Ø	{J}
B_5	Ø	{I,J}



数据流分析基础

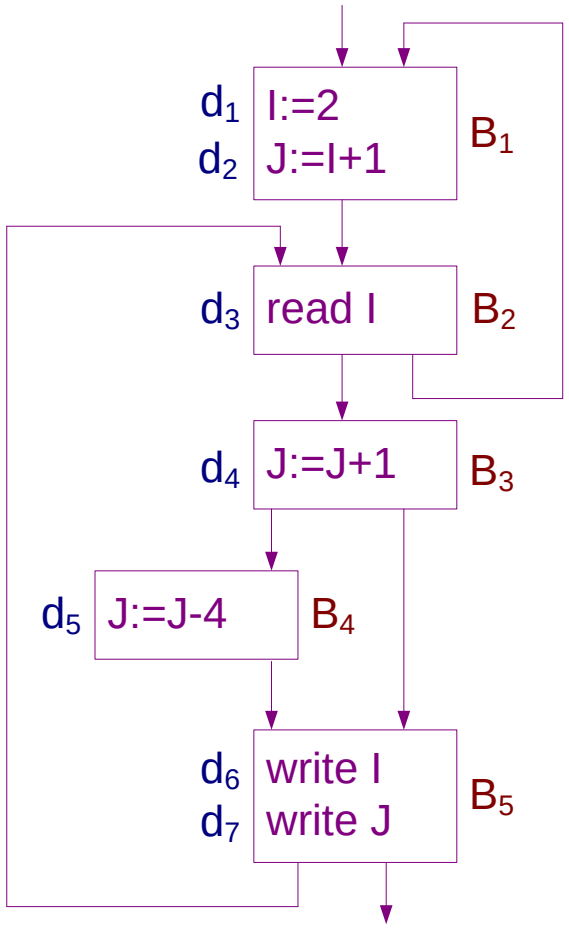
✧ 活跃变量数据流分析

– 活跃变量数据流方程求解举例

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

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

	Def	LiveUse	LiveOut	LiveIn
B_1	{I,J}	Ø	Ø	Ø
B_2	{I}	Ø	Ø	Ø
B_3	Ø	{J}	Ø	{J}
B_4	Ø	{J}	Ø	{J}
B_5	Ø	{I,J}	Ø	{I,J}



数据流分析基础

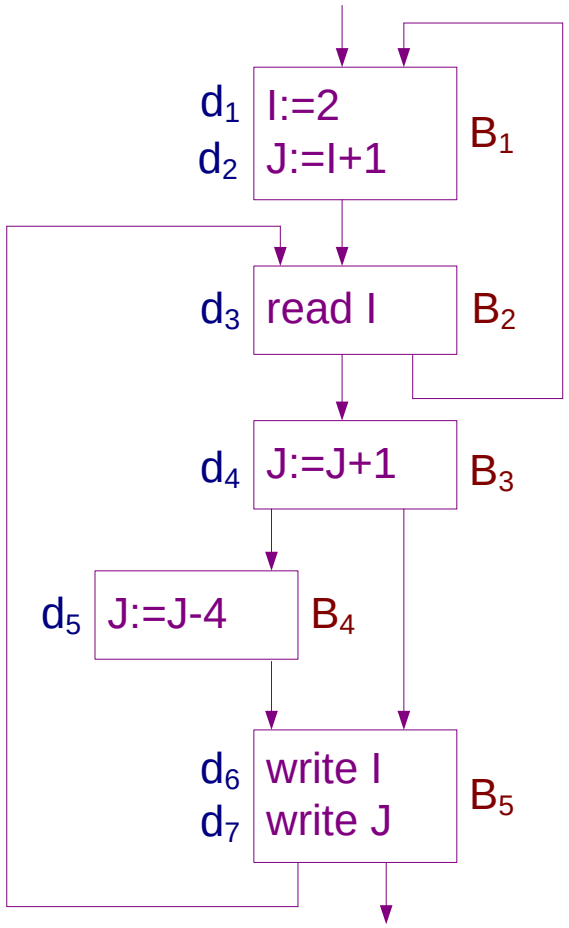
✧ 活跃变量数据流分析

– 活跃变量数据流方程求解举例

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

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

	Def	LiveUse	LiveOut	LiveIn
B_1	{I,J}	Ø	Ø	Ø
B_2	{I}	Ø	{J}	Ø
B_3	Ø	{J}	{I,J}	{J}
B_4	Ø	{J}	{I,J}	{J}
B_5	Ø	{I,J}	Ø	{I,J}



数据流分析基础

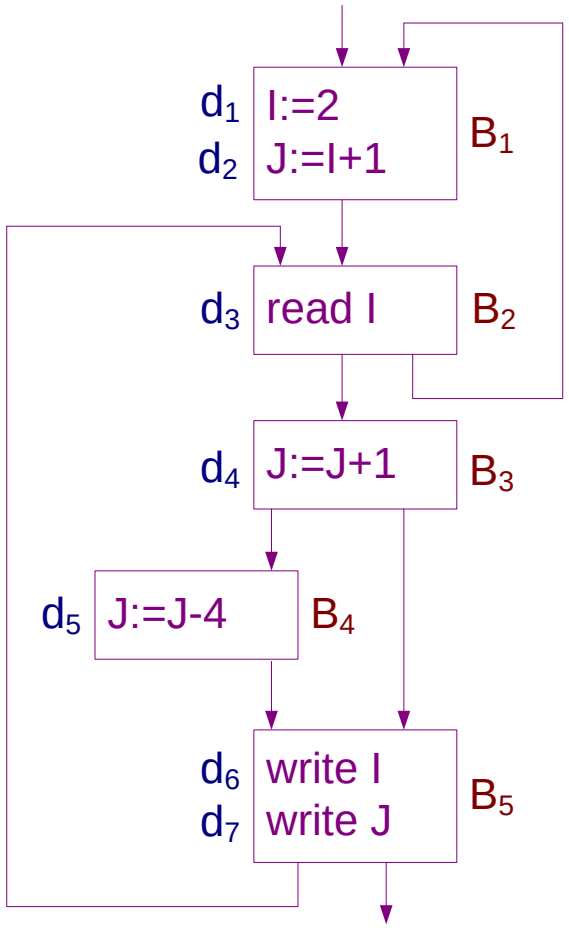
✧ 活跃变量数据流分析

– 活跃变量数据流方程求解举例

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

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

	Def	LiveUse	LiveOut	LiveIn
B_1	{I,J}	Ø	Ø	Ø
B_2	{I}	Ø	{J}	{J}
B_3	Ø	{J}	{I,J}	{I,J}
B_4	Ø	{J}	{I,J}	{I,J}
B_5	Ø	{I,J}	Ø	{I,J}



数据流分析基础

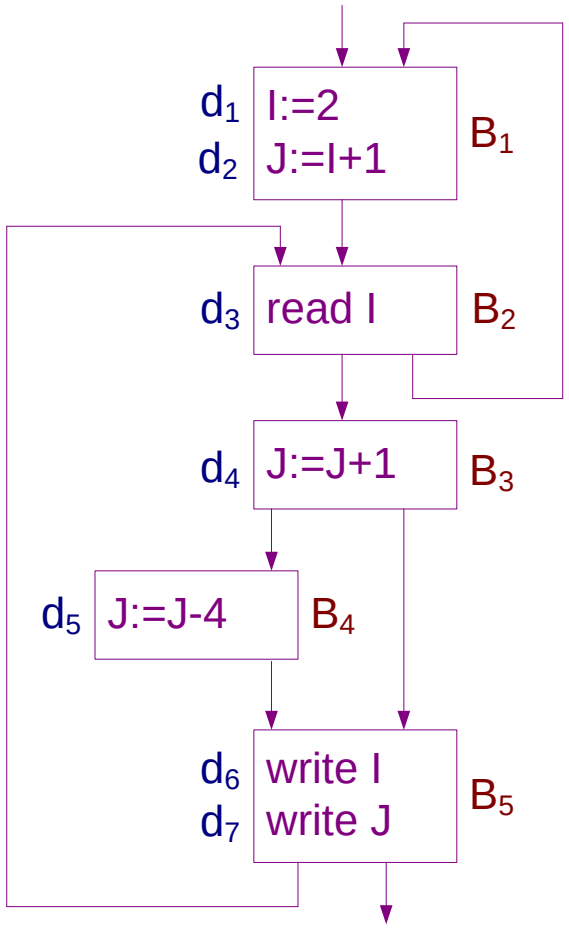
✧ 活跃变量数据流分析

– 活跃变量数据流方程求解举例

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

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

	Def	LiveUse	LiveOut	LiveIn
B_1	{I,J}	Ø	{J}	Ø
B_2	{I}	Ø	{I,J}	{J}
B_3	Ø	{J}	{I,J}	{I,J}
B_4	Ø	{J}	{I,J}	{I,J}
B_5	Ø	{I,J}	{J}	{I,J}



数据流分析基础

✧ 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—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，

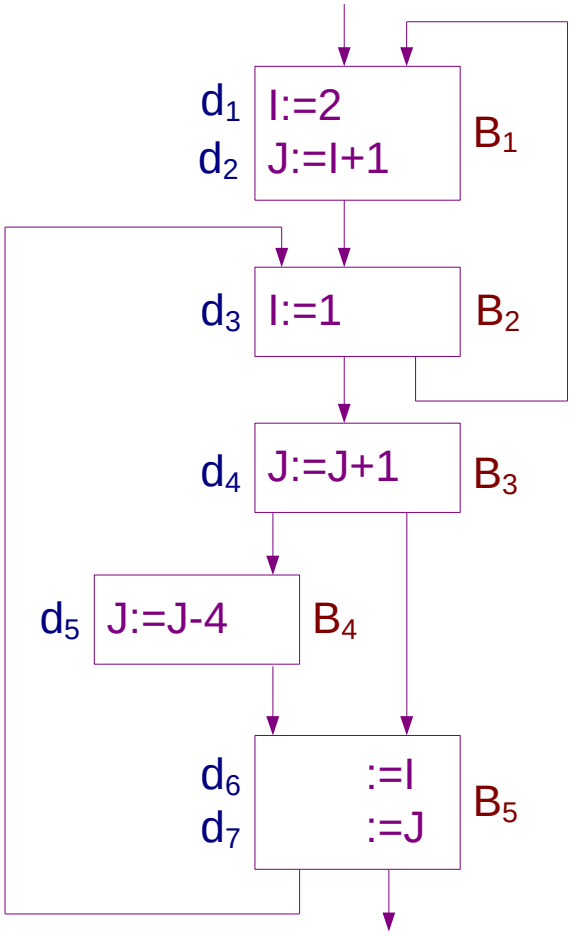
I 在引用点 d_2 的 UD 链为 $\{d_1\}$

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 $\{d_4, d_5\}$



数据流分析基础

✧ 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）

数据流分析基础

✧ 其他有用的数据流信息

– DU 链的计算举例

对于右边的流图，

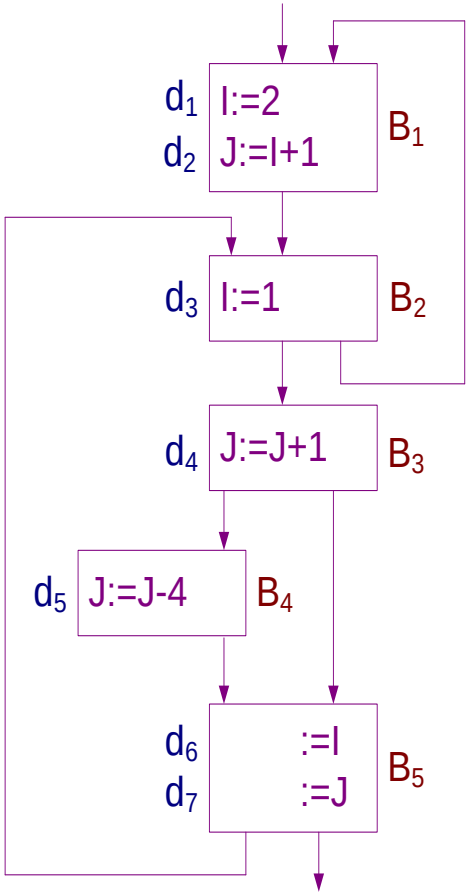
I 在定值点 d_1 的 DU 链为 $\{d_2\}$

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 $\{d_4, d_7\}$



数据流分析基础

✧ 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 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

数据流分析基础

（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

✧ 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

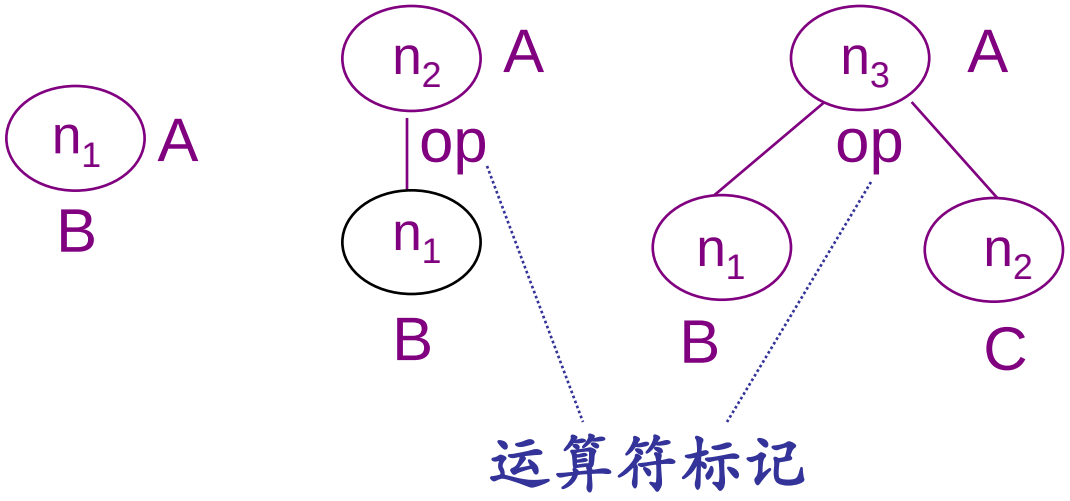
基本块的 DAG 表示

✧ 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:=op B　　A:= B op C

DAG 子图



（其他 TAC 语句的 DAG 结点的讨论参见龙书）

基本块的 DAG 表示

✧ 基本块 DAG 表示的构造

– 构造算法

设 x:=y op z, x:=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）

基本块的 DAG 表示

✧ 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 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）

基本块的 DAG 表示

✧ 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）

基本块的 DAG 表示

✧ 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

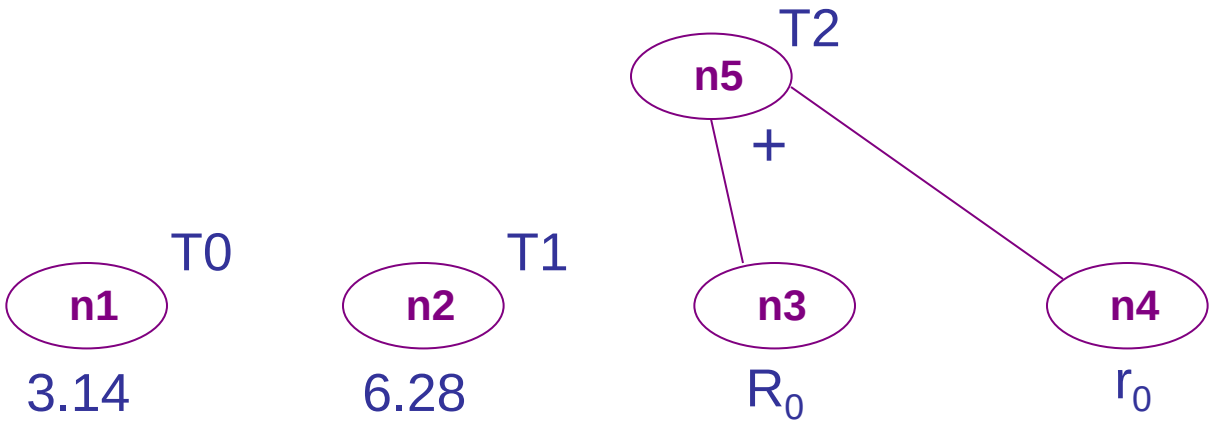
T0:=3.14
T1:=2*T0
T2:=R+r
A:=T1*T2
B:=A
T3:=2*T0
T4:=R+r
T5:=T3*T4
T6:=R-r
B:=T5*T6

基本块的 DAG 表示

✧ 基本块 DAG 表示的构造

– 举例

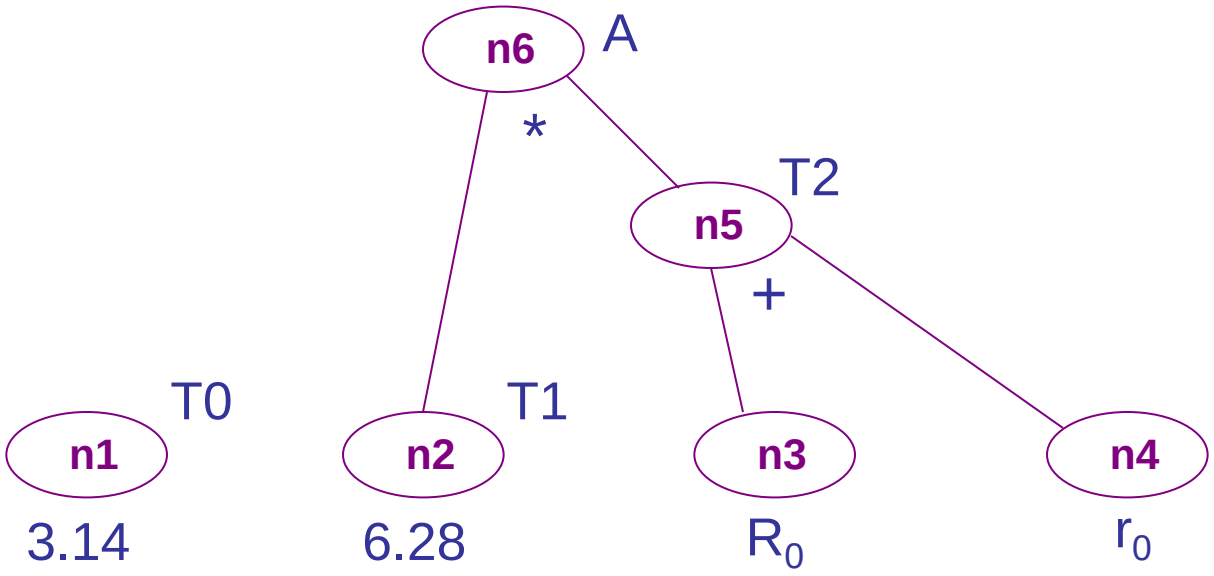
T0:=3.14
T1:=2*T0
T2:=R+r
A:=T1*T2 ⇦
B:=A
T3:=2*T0
T4:=R+r
T5:=T3*T4
T6:=R-r
B:=T5*T6



基本块的 DAG 表示

✧ 基本块 DAG 表示的构造

– 举例

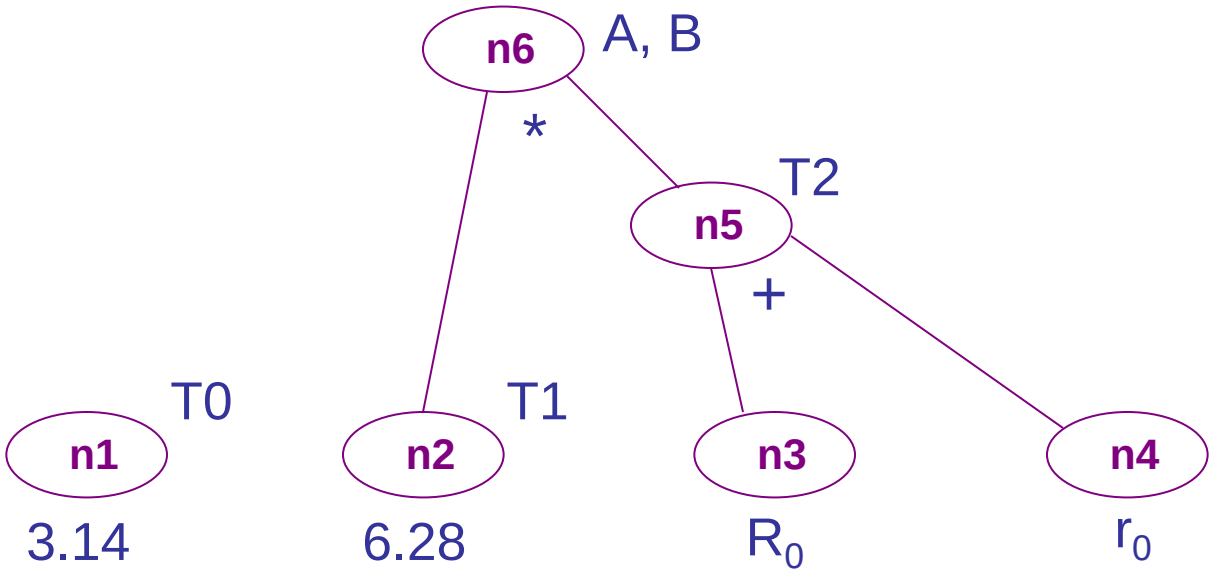


T0:=3.14
T1:=2*T0
T2:=R+r
A:=T1*T2
B:=A
T3:=2*T0
T4:=R+r
T5:=T3*T4
T6:=R-r
B:=T5*T6

基本块的 DAG 表示

✧ 基本块 DAG 表示的构造

– 举例

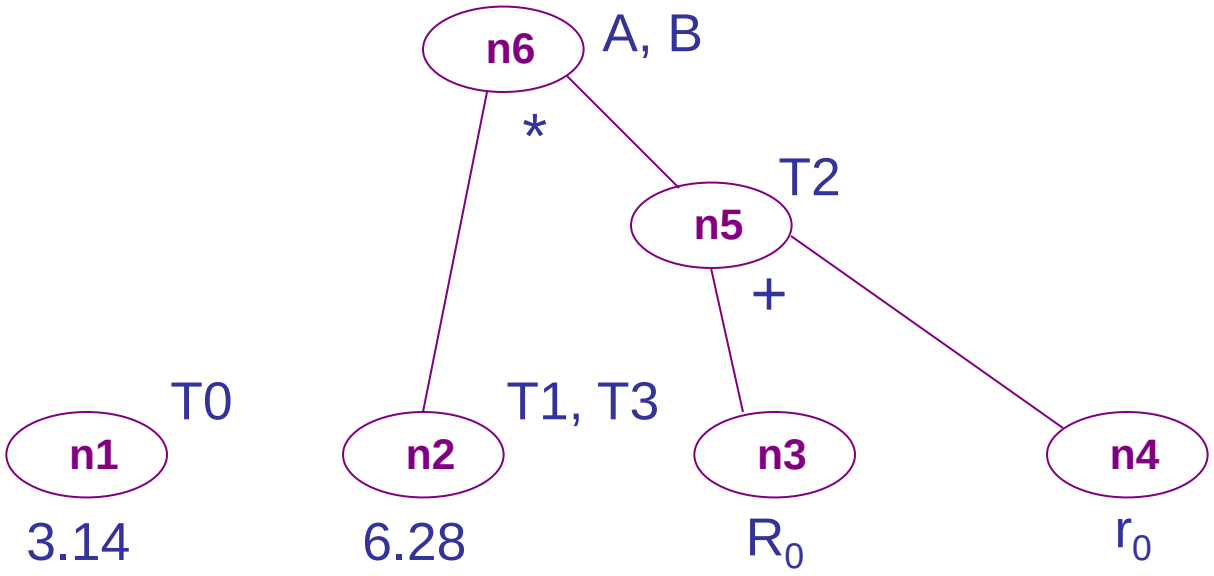


T0:=3.14
T1:=2*T0
T2:=R+r
A:=T1*T2
B:=A
T3:=2*T0 ⇦
T4:=R+r
T5:=T3*T4
T6:=R-r
B:=T5*T6

基本块的 DAG 表示

✧ 基本块 DAG 表示的构造

– 举例

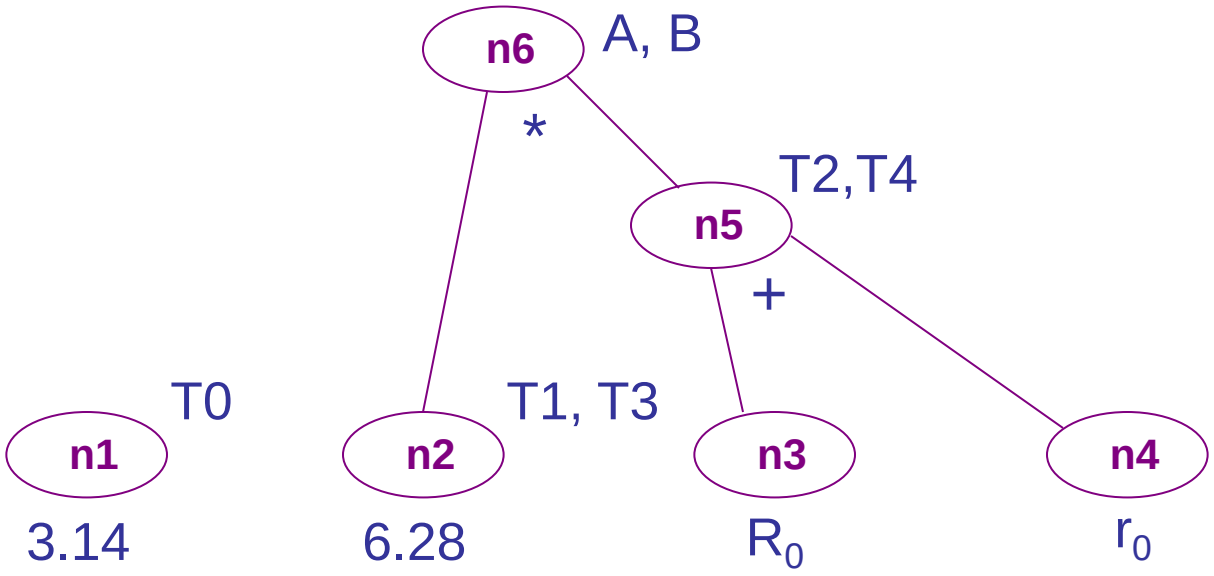


T0:=3.14
T1:=2*T0
T2:=R+r
A:=T1*T2
B:=A
T3:=2*T0
T4:=R+r
T5:=T3*T4
T6:=R-r
B:=T5*T6

基本块的 DAG 表示

✧ 基本块 DAG 表示的构造

– 举例

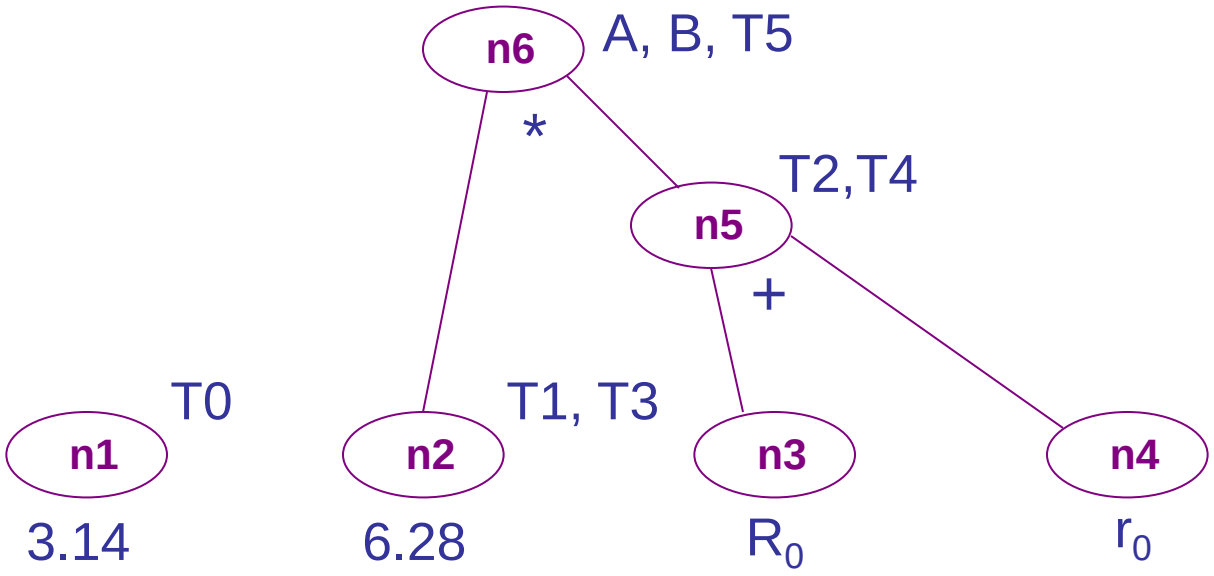


T0:=3.14
T1:=2*T0
T2:=R+r
A:=T1*T2
B:=A
T3:=2*T0
T4:=R+r
T5:=T3*T4 ⇦
T6:=R-r
B:=T5*T6

基本块的 DAG 表示

✧ 基本块 DAG 表示的构造

– 举例

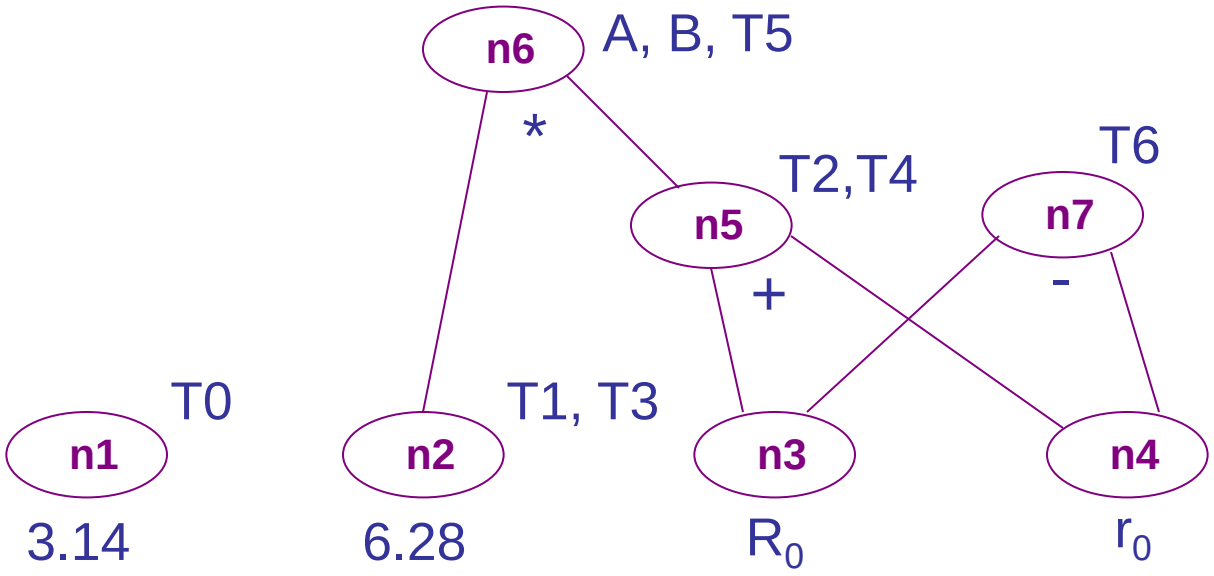


T0:=3.14
T1:=2*T0
T2:=R+r
A:=T1*T2
B:=A
T3:=2*T0
T4:=R+r
T5:=T3*T4
T6:=R-r
B:=T5*T6

基本块的 DAG 表示

✧ 基本块 DAG 表示的构造

– 举例

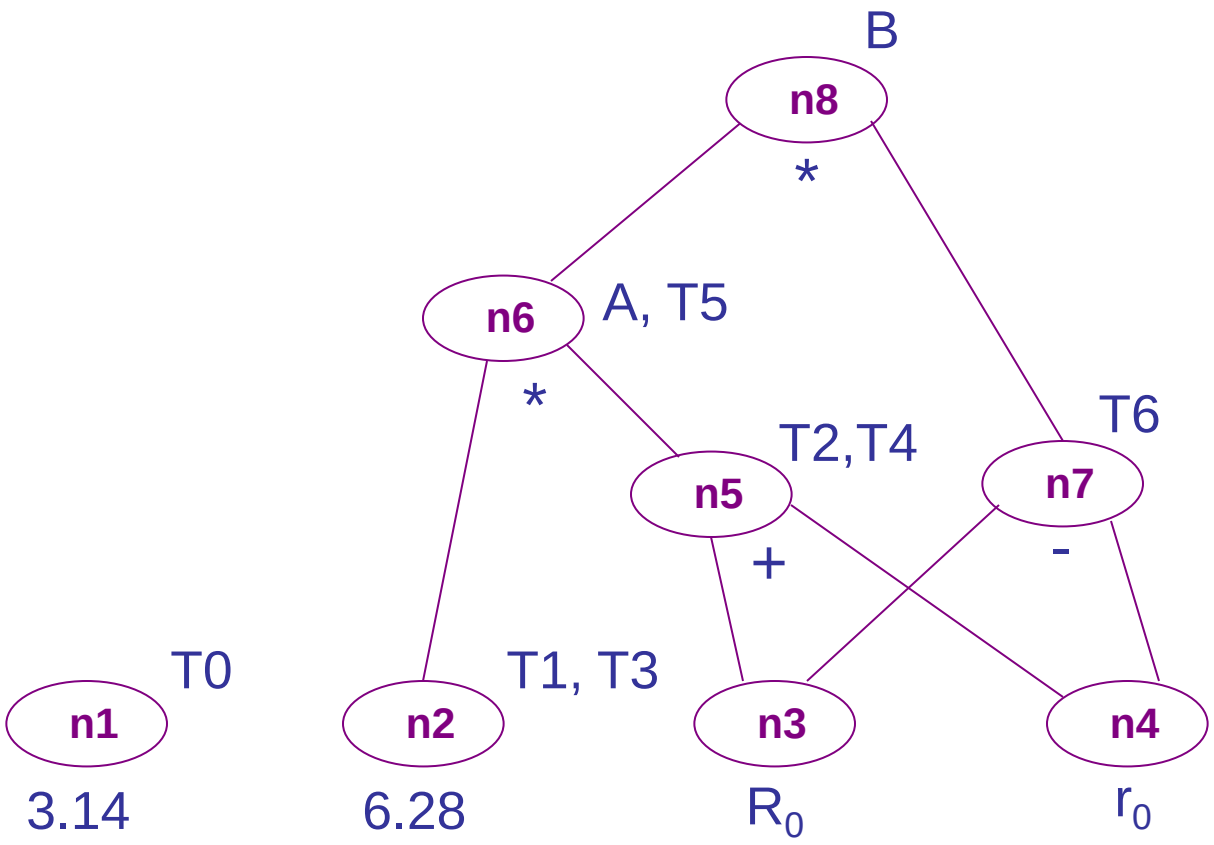


T0:=3.14
T1:=2*T0
T2:=R+r
A:=T1*T2
B:=A
T3:=2*T0
T4:=R+r
T5:=T3*T4
T6:=R-r
B:=T5*T6

基本块的 DAG 表示

✧ 基本块 DAG 表示的构造

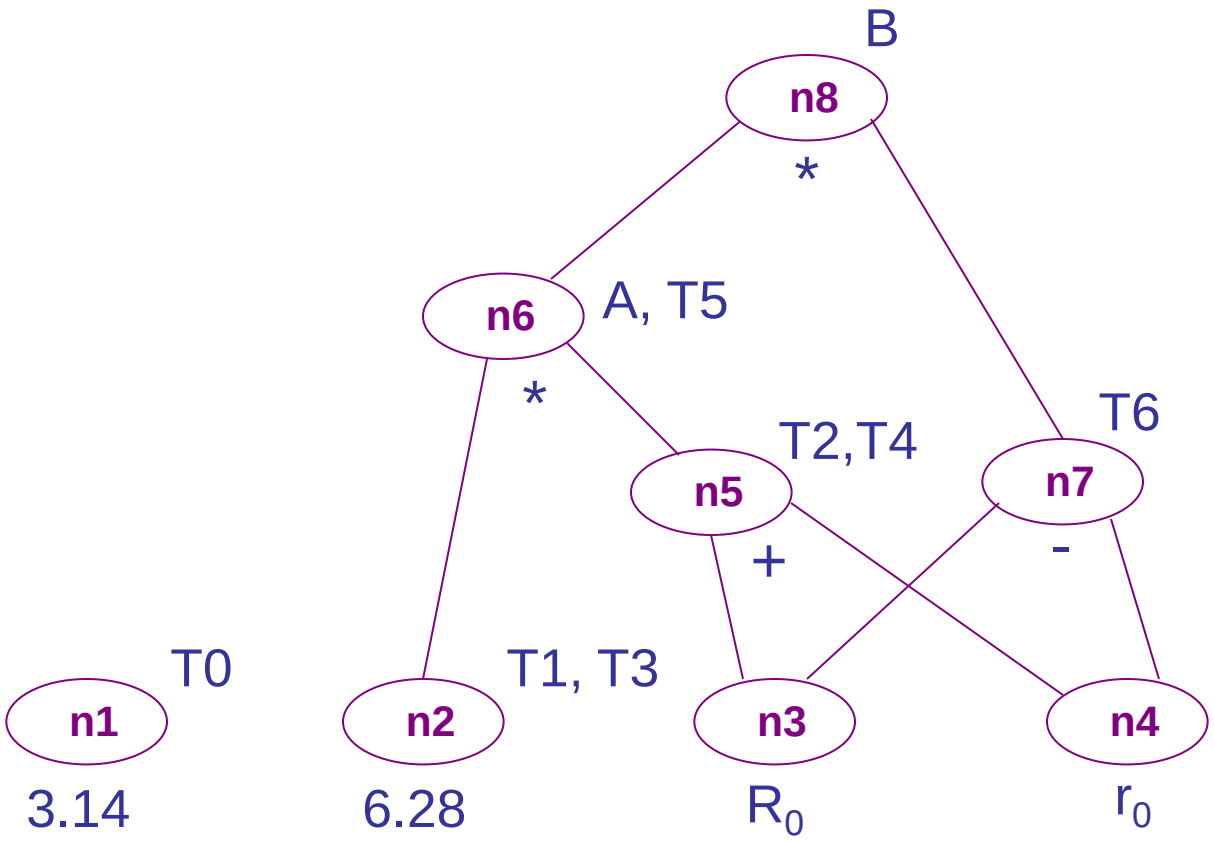
– 举例



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

目标代码生成技术

- ✧ 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

目标代码生成技术

✧ 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

目标代码生成技术

✧ 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

目标代码生成技术

✧ 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, R0      /* b 装入寄存器 R0 */
ADD   R0, c      /* c 加到 R0 */
MOV   R0, a      /* 存 R0 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。

目标代码生成技术

✧ 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

目标代码生成技术

✧ 一个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题

  原则：尽可能地让变量的值保留在寄存器中

  尽可能引用变量在寄存器中的值

  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

✧ 一个简单的代码生成算法

– 寄存器描述数组和变量地址描述数组

- RVALUE[R] 描述寄存器 R 当前存放哪些变量
- AVALUE[A] 表示变量 A 的值存放在哪个寄存器中（或不在任何寄存器中）

目标代码生成技术

✧ 一个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

（假设只有形如 A:=B op C 和A:=B的*TAC* 语句序列）

step1：对每个*TAC* 语句i，依次执行下述步骤：

- 以 i 为参数，调用 getreg(i); 从 getreg 返回时，得到一寄存器 R（这里先假定 R 为伪寄存器），作为存放 A 现行值的寄存器；
- 利用 AVALUE[B] 和 AVALUE[C]，确定出 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`；如果其现行值不在寄存器中，则在相应指令中仍用B和 C表示。
- 分两种情形生成目标代码：

a) 对于 i: A:=B op C。如果B现行值不在寄存器或者B`≠R，则生成

MOV B, R 或 MOV B, R 或 MOV B`, R 或 MOV B`, R
OP R, C OP R, C` OP R, C OP R, C`
否则，则生成 OP R, C 或 OP R, C`

✧ 一个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（续前页）

如 B` 或 C` 为R，则删除 AVALUE[B] 或 AVALUE[C] 中的 R

对每个 D≠B，D∈RVALUE[R]，并且在语句 i 之后 D 仍然是活跃变量，则在生成以上代码之前先插入一条指令 MOV R, D

令 AVALUE[A]={R}，并令 RVALUE[R]={A}，以表示变量 A 的现行值只在 R 中并且 R 中的值只代表 A 的现行值

b) 对于 i: A:=B。如果B现行值不在寄存器中，则生成 MOV B, R
令 AVALUE[B] = {R}，并令 RVALUE[R]={A, B}

如果B现行值在寄存器(R)中，则将 A 加入集合 RVALUE[R]

无论何种情况，都令 AVALUE[A] = {R}

✧ 一个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（续前页）

- 如 **B** 或 **C** 的现行值在基本块中不再被引用，它们也不是基本块出口之后的活跃变量（由语句 i 上的附加信息知道），并且其现行值在某个寄存器 R_k 中，则删除 RVALUE[R_k] 中的 B 或C 以及 AVALUE[B] 或 AVALUE[C] 中的 R_k，使该寄存器不再为 B 或 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 中的每个变量 M，若它在出口之后是活跃的，则生成 MOVE R, M，将其存入主存

目标代码生成技术

✧ 一个简单的代码生成算法

– 函数 getreg

（以 i: A:=B op C 或i: A:=B为参数， 返回一个伪寄存器）

步骤：

• 对于 i: A:=B op C

若 B∈RVALUE[R]，且在语句 i 之后 B 在基本块中不再被引用，同时也不是基本块出口之后的活跃变量（由 i 上的附加信息可知道），则返回 R;

否则，返回一个新的伪寄存器R'

• 对于 i: A:=B

若 B∈RVALUE[R] ，则返回 R;

否则，返回一个新的伪寄存器R'

目标代码生成技术

✧ 一个简单的代码生成算法举例

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

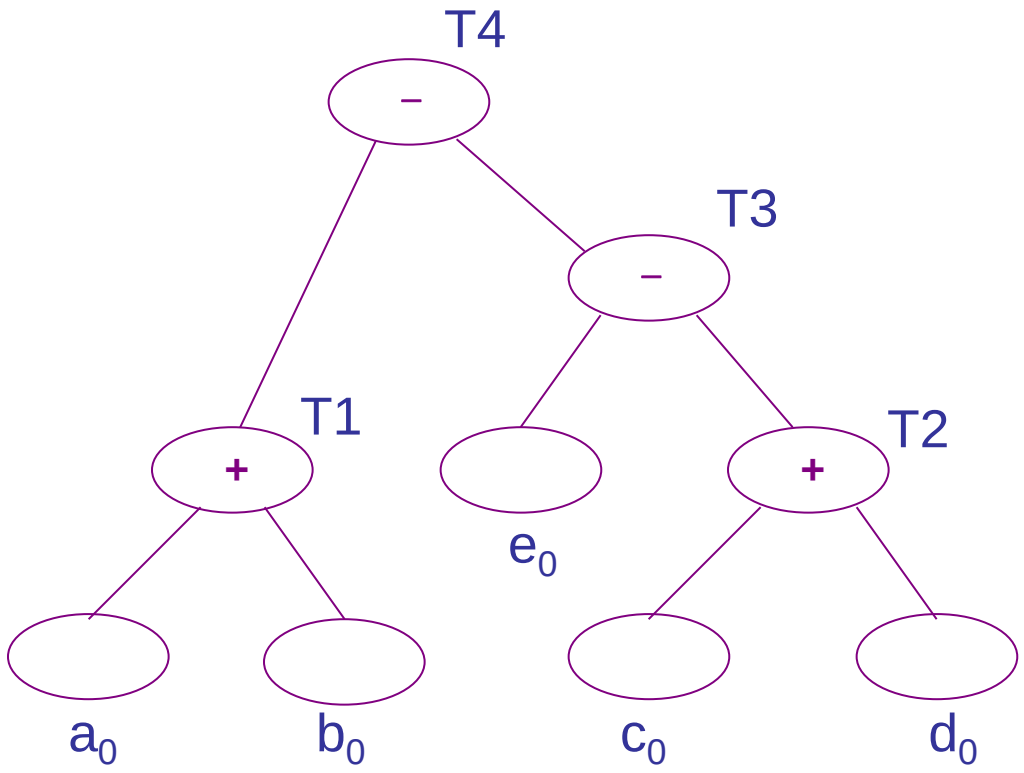
```
t := a - b
a := b
u := a - c
v := t + u
d := v + u
```

语句	生成的代码	寄存器描述	变量地址描述
t: = a－b	MOV a,R_0 SUB R_0, b	空寄存器 R_0 包含 t	 t 在 R_0 中
a := b	MOV b, R_1	R_0 包含 t R_1 包含 a,b	t 在 R_0 中 a,b 在 R_1 中
u: = a－c	MOV R_1, b SUB R_1, c	R_0 包含 t R_1 包含 u	t 在 R_0 中 u 在 R_1 中
v: = t＋u	ADD R_0, R_1	R_0 包含 v R_1 包含 u	u 在 R_1 中 v 在 R_0 中
d: = v＋u	ADD R_0, R_1 MOV R_0,d	R_0 包含 d	d 在 R_0 中 d 在 R_0 中和内存中

目标代码生成技术

✧ 高效使用寄存器

– 从某个基本块的 *DAG* 表示得到的两段 *TAC* 代码




```
T1:=a+b
T2:=c+d
T3:=e-T2
T4:=T1-T3
```

```
T2:=c+d
T3:=e-T2
T1:=a+b
T4:=T1-T3
```

✧ 高效使用寄存器

– 将上述简单的代码生成算法应用于如下两个基本块比较其结果（这里假设基本块出口处只有T4是活跃的）

```
T1:=a+b
T2:=c+d
T3:=e-T2
T4:=T1-T3
```

⇨

```
MOV a, R0
ADD R0, b
MOV c, R1
ADD R1, d
MOV e, R2
SUB R2, R1
SUB R0, R2
MOV R0,T4
```

```
T2:=c+d
T3:=e-T2
T1:=a+b
T4:=T1-T3
```

⇨

```
MOV c, R0
ADD R0, d
MOV e, R1
SUB R1, R0
MOV a, R0
ADD R0, b
SUB R0, R1
MOV R0,T4
```

第二段代码较优（少用了寄存器）

目标代码生成技术

✧ 高效使用寄存器

– 从上例可知：对于上述简单的代码生成算法，从基本块 DAG 表示产生语句的次序影响到目标代码生成的质量

一个从 DAG 产生语句序列的启发式排序算法

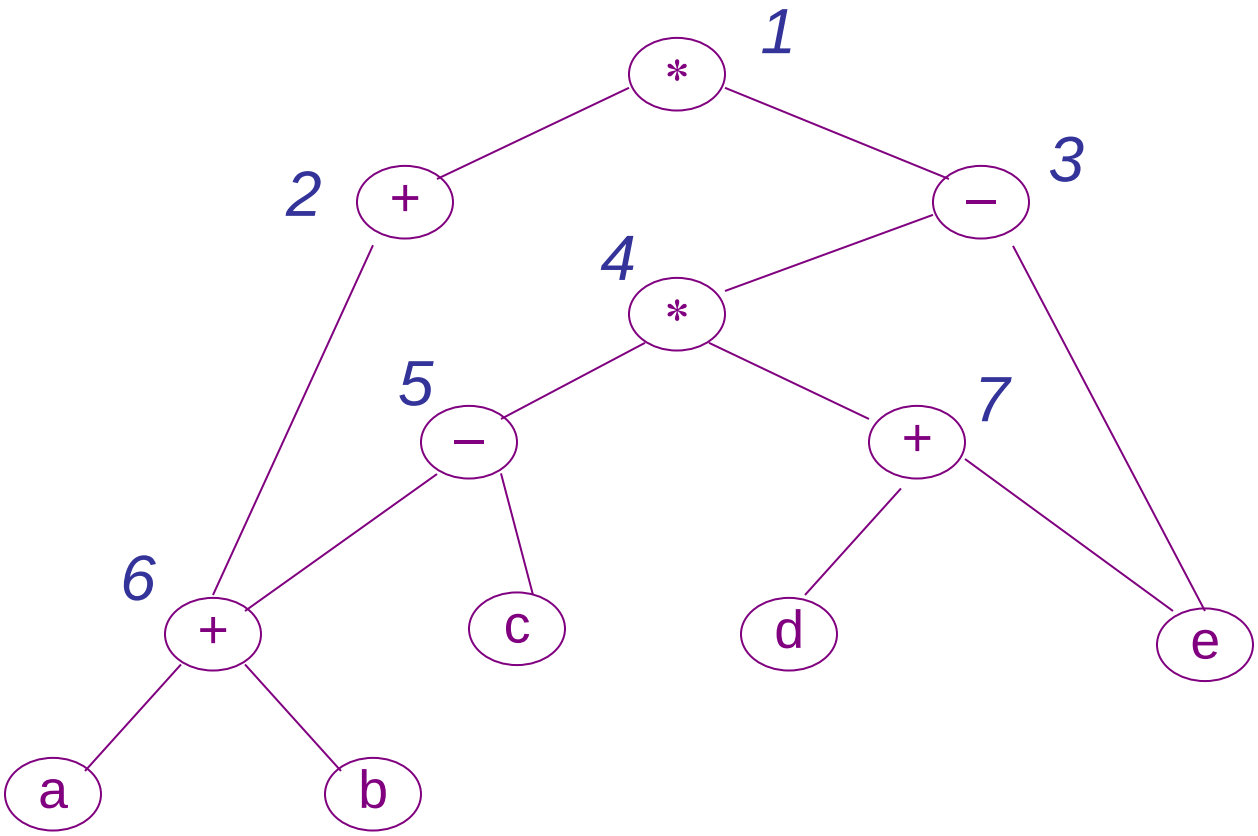
```
while 存在未列入表的内部结点{
    选取一个未列入表的但其全部父结点均已列入表的结点 n;
    将 n 列入表中;
    while n 的最左孩子 m 不是叶结点且其所有父结点均已在表中{
        将 m 列入表中;
        n := m
    }
}
```

结果：产生语句的次序应与内部结点列入表中的次序相反

目标代码生成技术

✧ 高效使用寄存器

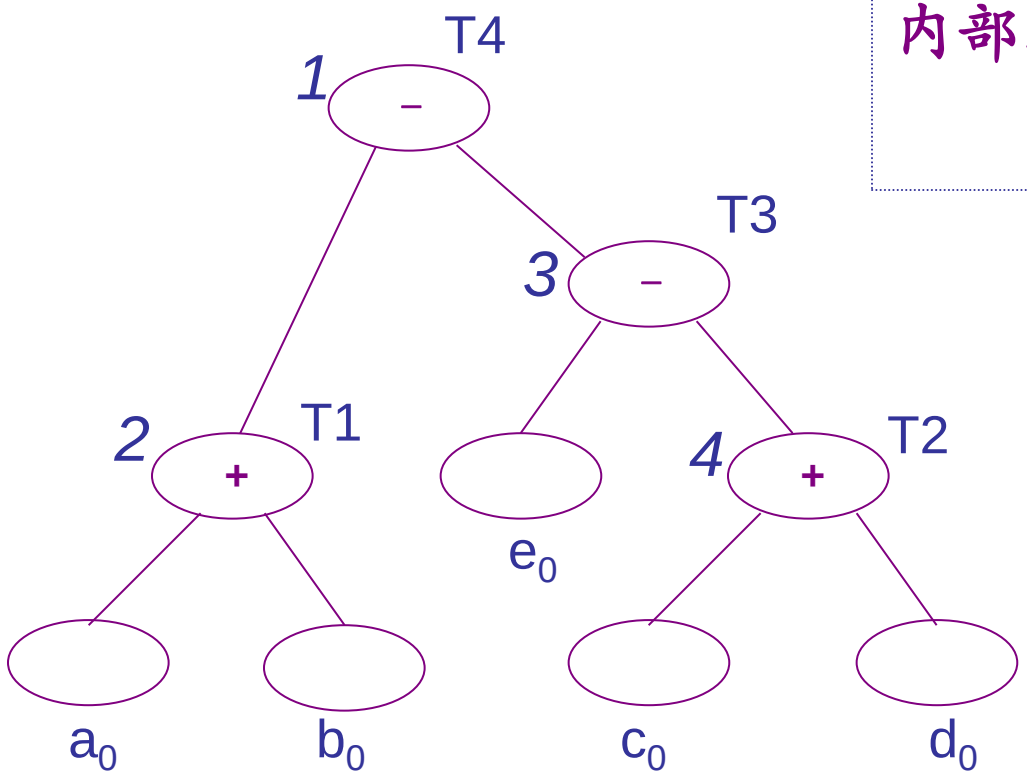
– 从 DAG 产生语句序列的启发式排序算法举例



目标代码生成技术

✧ 高效使用寄存器

– 从 DAG 产生语句序列的启发式排序算法举例



内部结点列入表中的次序

T4 T1 T3 T2

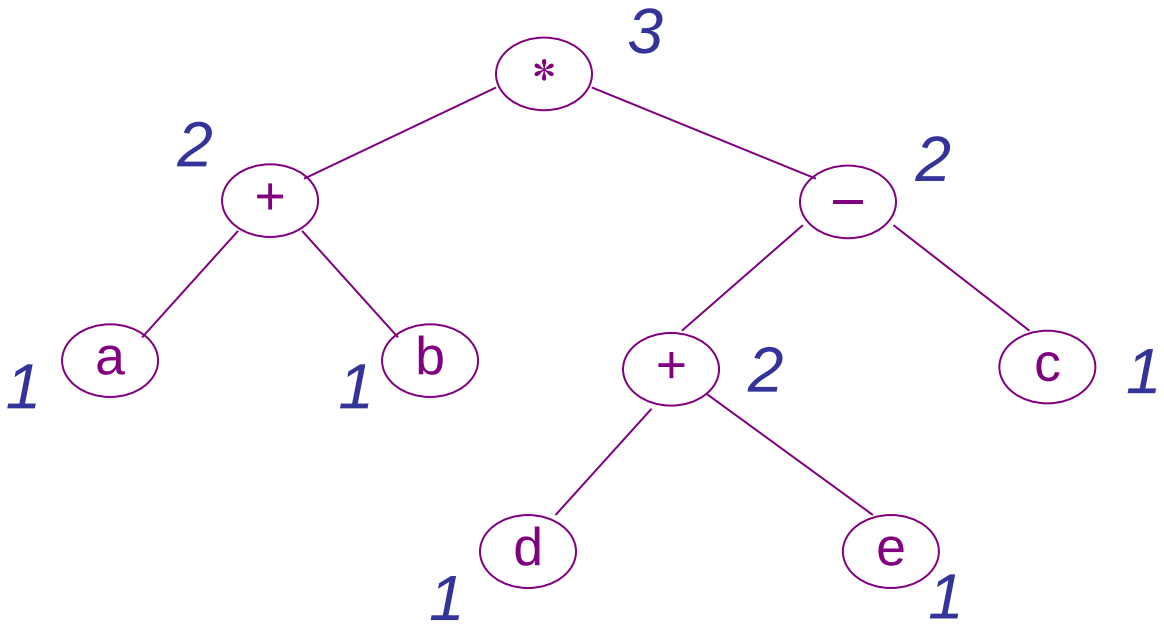
结果

T2:=c+d
T3:=e-T2
T1:=a+b
T4:=T1-T3

✧ 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

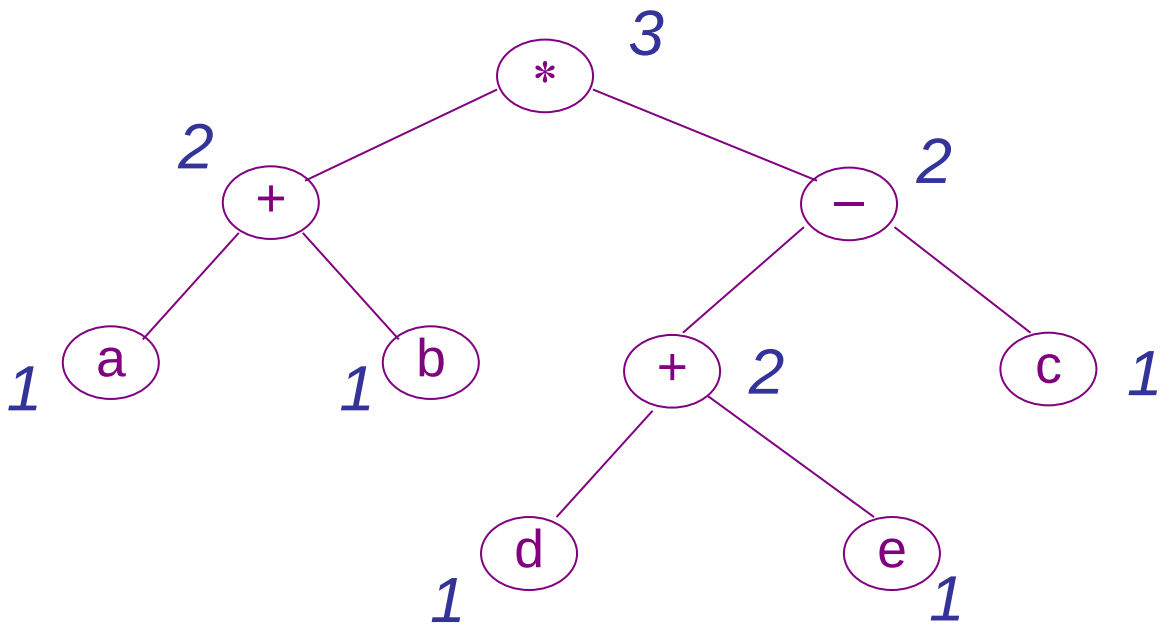


目标代码生成技术

✧ 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

  例如：前面介绍的简单代码生成算法中的 getreg 函数返回一个伪寄存器（不管物理寄存器的个数）

- 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

目标代码生成技术

✧ 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

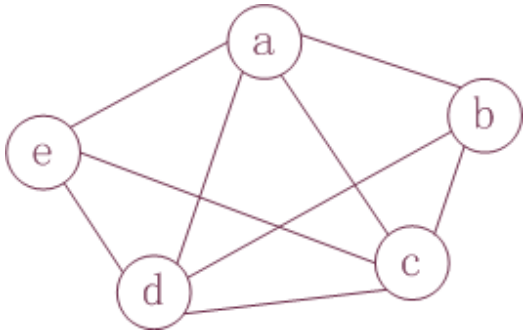
- 对相干图进行着色（*coloring*）

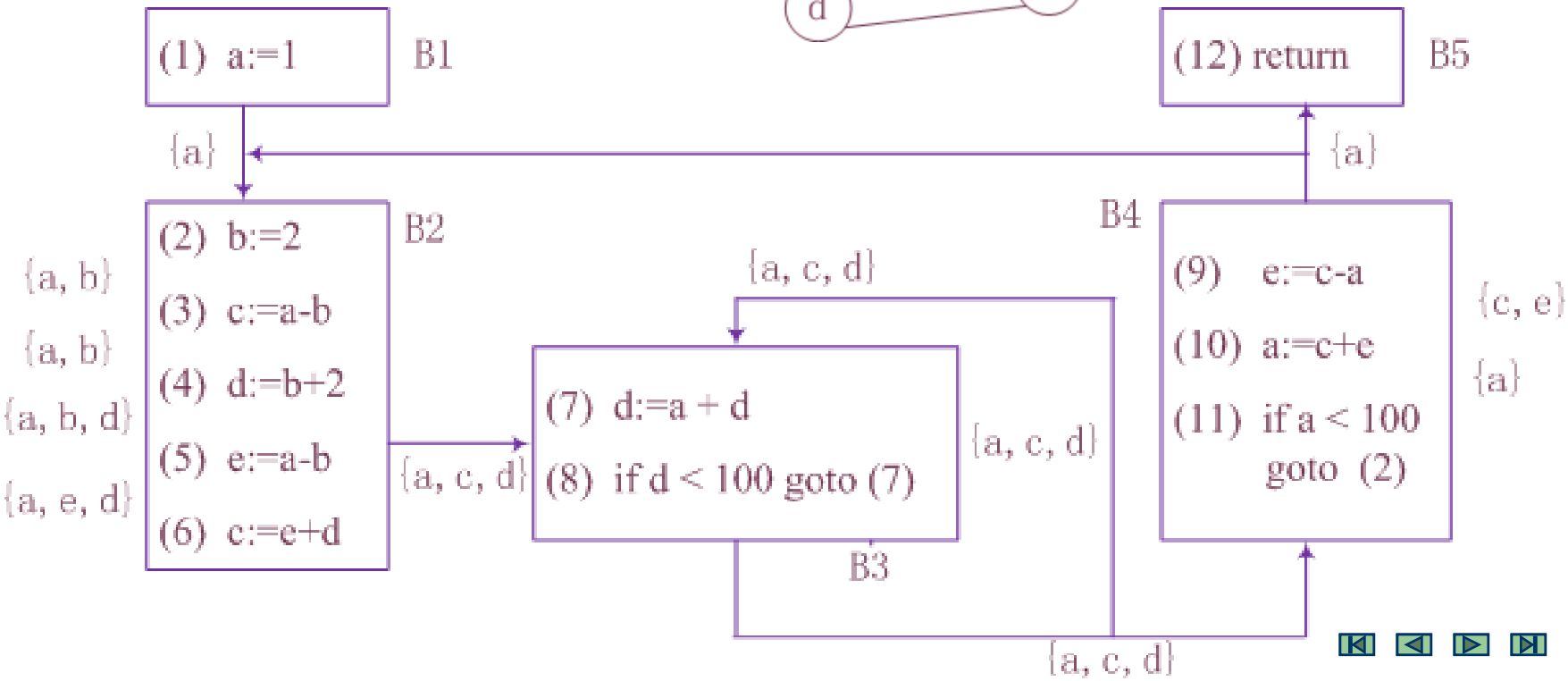
  使用k（物理寄存器数量）种颜色对相干图进行着色，使任何相邻的结点具有不同的颜色（即两个相干的伪寄存器不会分配到同一个物理寄存器）

目标代码生成技术

✧ 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

代码优化技术

✧ 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

代码优化技术

- 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
    - ……

代码优化技术

✧ 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

代码优化技术

✧ 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） MOV R0，a
（2） MOV a，R0

可优化为

（1） MOV R0，a

代码优化技术

✧ 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

代码优化技术

✧ 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）

代码优化技术

✧ 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
```

代码优化技术

✧ 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
……
```

代码优化技术

✧ 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

代码优化技术

✧ 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

  …

代码优化技术

✧ 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

代码优化技术

✧ 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）

  ……

代码优化技术

✧ 循环优化（*loop optimization*）

– 举例（选讲）

- 代码外提（*code motion*）

  while （i< limit/2）{…}

  等价于 t:=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

代码优化技术

✧ 循环优化（*loop optimization*）

– 举例（选讲）

- 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：

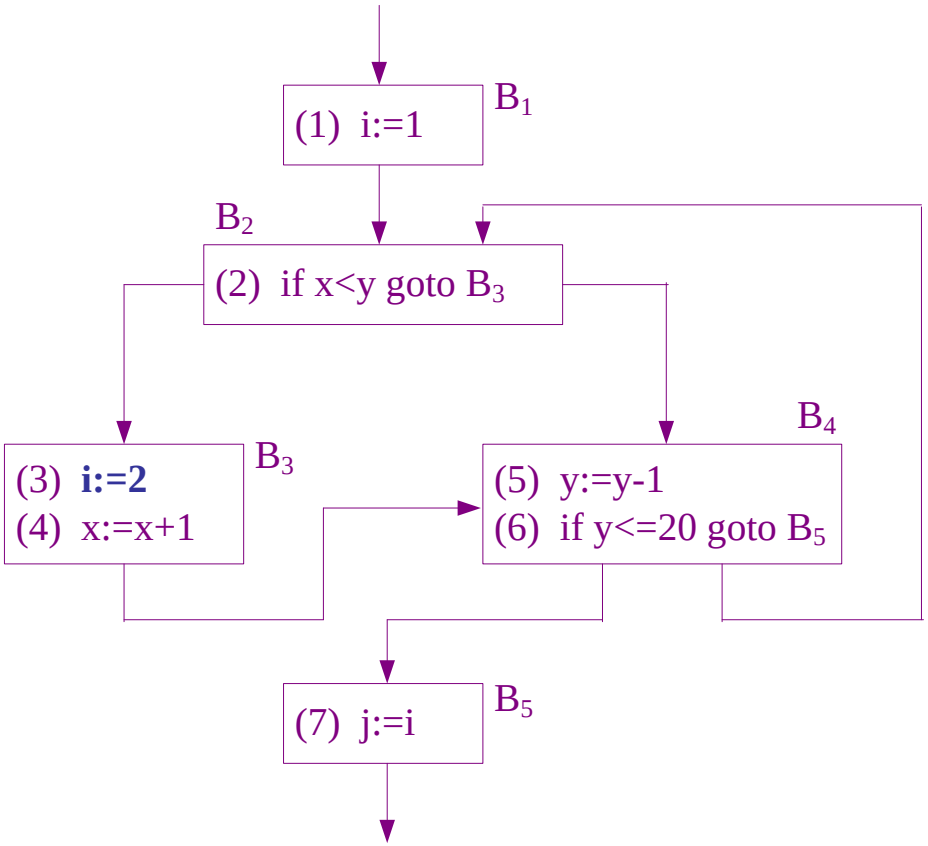
1) 所在结点是循环的所有出口结点的支配结点
2) 循环中其它地方不再有 x 的定值点
3) 循环中 x 的所有引用点都是且仅是这个定值所能达到的
4) 若y或z是在循环中定值的，则只有当这些定值点的语句（一定也是循环不变量）已经被执行过代码外提

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 （选讲）



代码优化技术

✧ 循环优化（*loop optimization*）

– 举例（选讲）

- 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

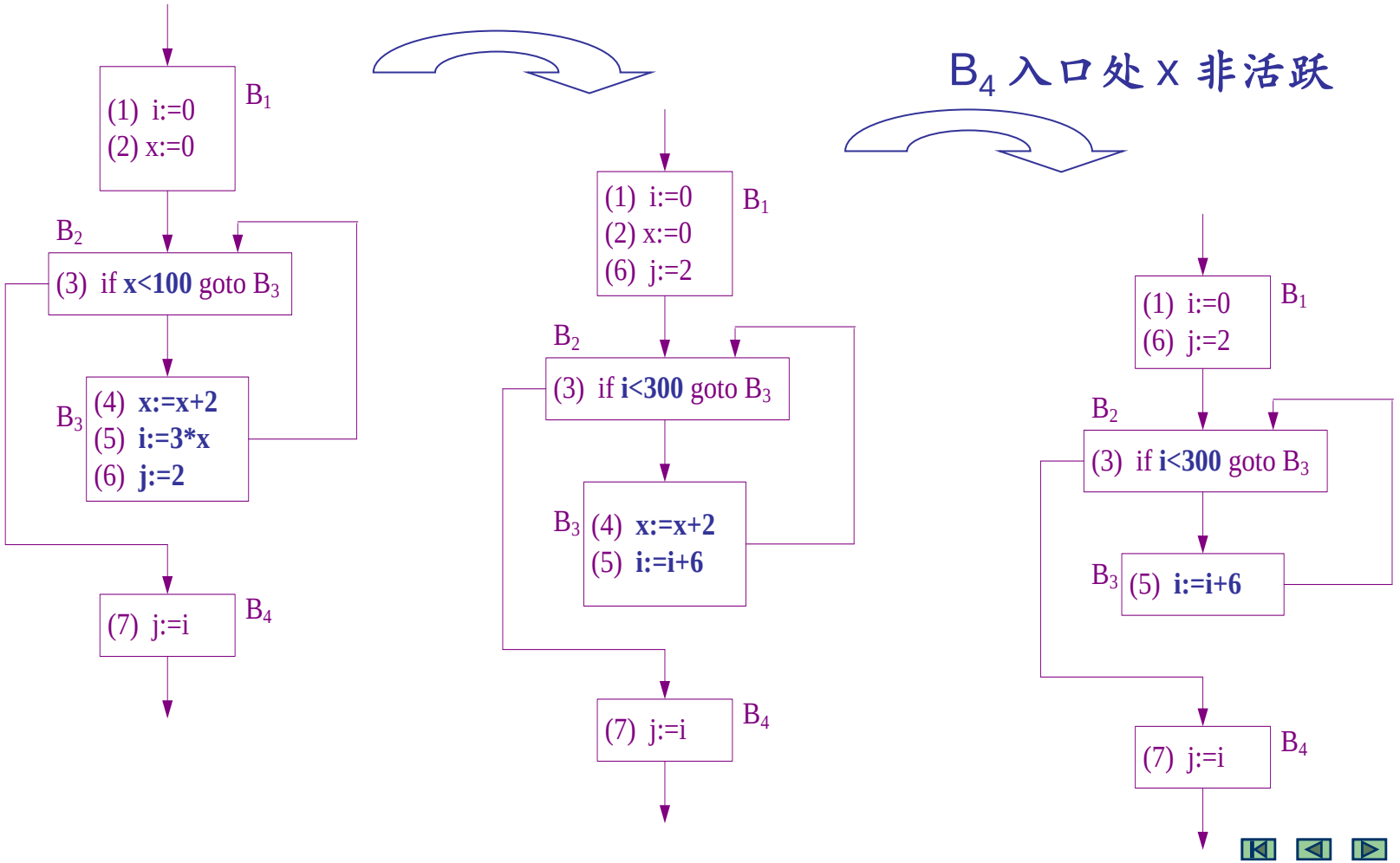
  通常可以针对归纳变量可以进行如下优化：

  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

代码优化技术

✧ 循环优化举例（选讲）



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