



第 8 章 代码优化 (optimization)

8.1 代码优化综述

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



第 8 章 代码优化 (optimization)

8.1.1 代码优化概述

8.1.2 优化技术分类

8.1.3 具优化功能编译器的组织



第 8 章 代码优化 (optimization)

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

■ 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。

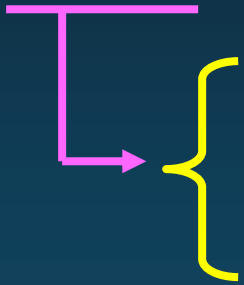
空间效率和时间效率

■ 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；

■ 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；

■ 改进、提高程序途径

(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。

■ 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:



■ 优化原则

等价: 是指不改变程序的运行结果;

有效: 主要指优化后的目标代码运行时间较短,
以及占用的存储空间较小。

合算: 应尽可能以较低的代价取得较好的优化效果。

■ 优化的时机及分类

优化可在编译的各个阶段进行。主要时机是在语法、语义分析生成中间代码之后，另一类优化则是在生成目标程序时进行的。

前者优化不依赖于具体的计算机而取决于语言的结构，后者依赖于具体目标机。

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 — 源程序大范围内优化;

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. Constant folding and propagation

Before optimization

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

After optimization

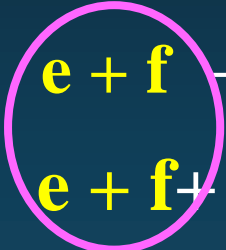
```
X = 2;  
Y = 12;  
Z = 24;
```

常量合并、传播

2. Common subexpression elimination

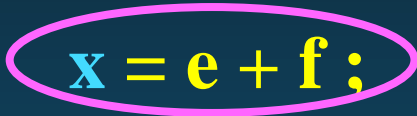
Before optimization

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



After optimization

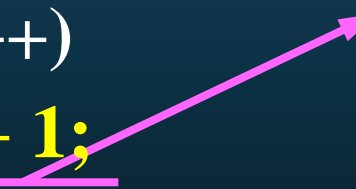
```
x = e + f;  
d = x + g;  
y = x + z;
```



3. Loop invariant code motion

Before optimization

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



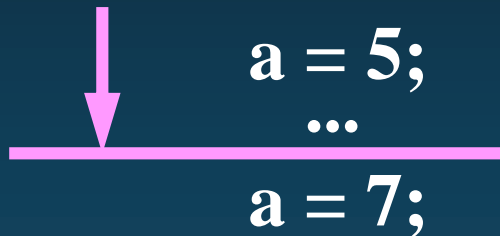
After optimization

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



4. Dead storage/assignment elimination

Before optimization



```
a = 5;  
...  
a = 7;
```

After optimization

```
a = 7;
```

5. Jump-to-jump elimination

Before optimization

```
if(x)  
...  
else goto J1;  
J1: goto J2;
```

After optimization

```
if(x)  
...  
else goto J2;
```



6. Dead code elimination

Before optimization

```
char c;
```

```
if (c > 300) a = 1;
```

```
else
```

```
    a = 2 ;
```

After optimization

```
a = 2;
```

永假式




7. Function embed

Before optimization

```
int Check(int x);  
{  
    return (x>10);  
}
```

```
void main()  
{  
    if check(y)  
        a=5;  
}
```



After optimization

```
void main()  
{  
    if (y>10) a=5;  
}
```

8. Loop transformation(强度削弱) - simple loop

C source code

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

before optimization

```
i = 0;
```

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

Loop

after optimization

```
i = 0;
```

```
t1 = i * 4;
```

```
L1: table[t1] = 0 ;
```

```
t1 = t1 + 4;
```

```
i++;
```

```
if(i<100) goto L1
```


9. Loop transformation - dynamic loop

C source code

```
step = step_table[1];  
for(i=0; i<MAX; i+=step)  
    table[i] = 0;
```

before optimization

```
step = step_table[1];  
i = 0;
```

```
L1: t1 = i * 4;  
    table[t1] = 0;  
i = i + step;  
    if(i<MAX) goto L1
```

$$\begin{aligned} & (i + \text{step}) * 4 \\ = & \underbrace{i * 4}_{t1} + \underbrace{\text{step} * 4}_{t2} \end{aligned}$$

after optimization

```
i = 0;  
step = step_table[1];  
t1 = i * 4;  
t2 = step * 4;  
L1: table[t1] = 0;  
t1 = t1 + t2;  
i = i + step;  
    if(i<MAX) goto L1
```

10. Loop transformation - composed variables

C source code

```
int table[100];  
for(i=0, j=0; j<10; i++, j++)  
    table[ 10 * i + j ] = i ;
```

```
table[ 0 ] = 0  
table[ 11 ] = 1  
table[ 22 ] = 2  
.....  
table[ 99 ] = 9
```

before optimization

```
i = 0; j = 0;
```

```
t1 = i * 10;
```

```
L1: t2 = t1 + j;
```

```
t3 = t2 * 4;
```

```
table[t3] = i;
```

```
i = i + 1;
```

```
t1 = t1 + 10;
```

```
j = j + 1;
```

```
if(j < 10) goto L1
```

```
/* address */
```

after optimization

```
i = 0; j = 0;
```

```
t1 = i * 10;
```

```
t2 = t1 + j;
```

```
t3 = t2 * 4; /*t3=0*/
```

```
Repeat 10 times:
```

```
table[t3] = i;
```

```
i = i + 1;
```

```
t3 = t3 + 44;
```

■ 考虑目标机指令系统特点

Before optimization

```
int x, y, z;  
  x=1;  
  y=x;  
  z=1;
```

After optimization

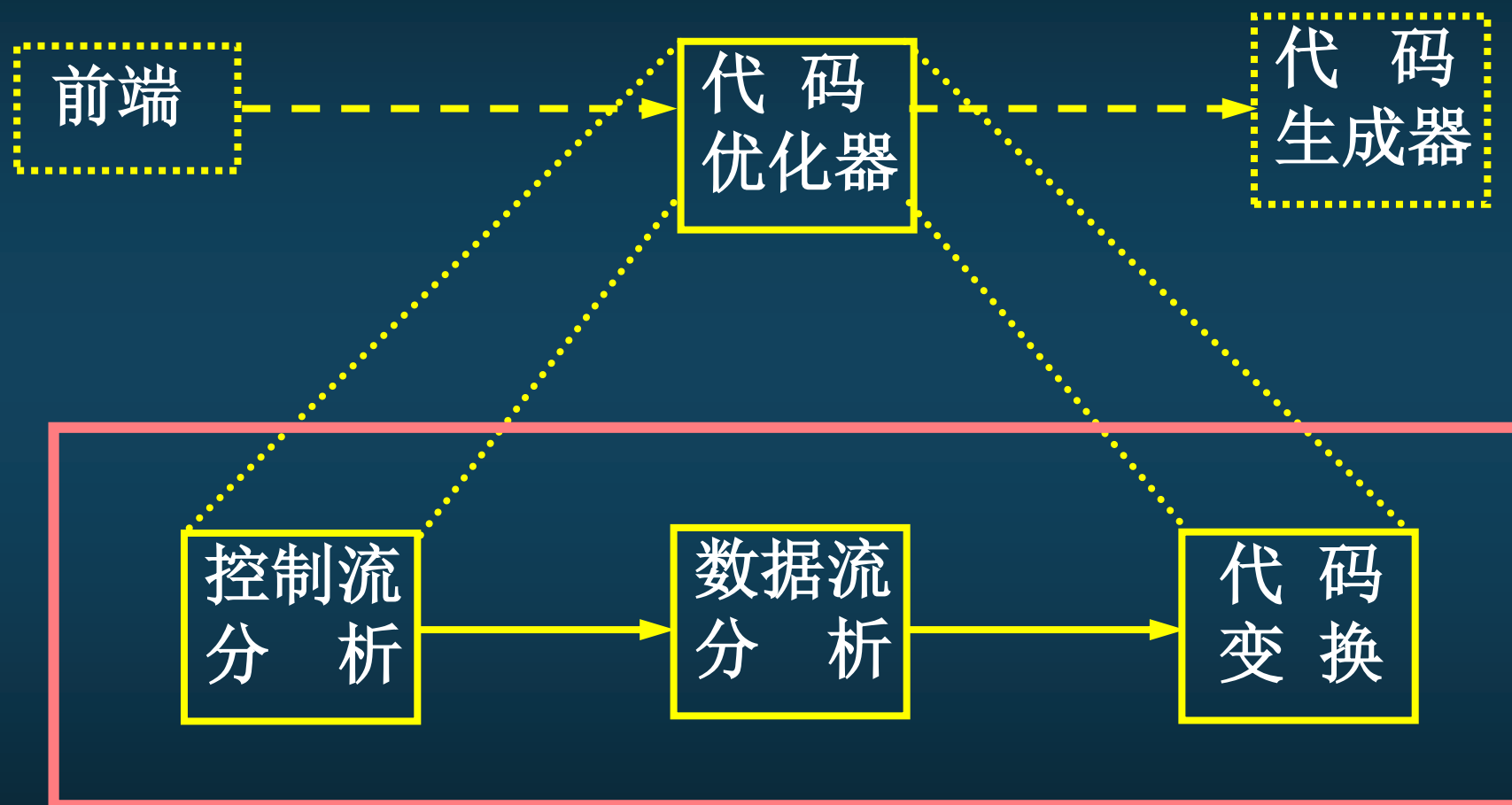
```
int x, y, z;  
  x=1;  
  z=1;  
  y=x;
```

After optimization

C source code

```
int foo(int n)
{ int x;
  for (int i=0;i<n;i++)
    x++;
  return x;
}
```

```
int foo(int n)
{ int x;
  for ( int i=0; i<n/4;i++ )
    { x++;
      x++;
      x++;
      x++;
    }
  for ( int i=0; i<n%4;i++ )
    x++;
  return x;
}
```



■ 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**执行的语句序列，只有**惟一入口**和**惟一出口**，且分别对应该序列的第一个语句和最后一个语句。

■ 基本块特点

基本块内的语句是顺序执行的，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。

根据基本块的结构特点，它的入口语句是下述三种类型的语句之一：

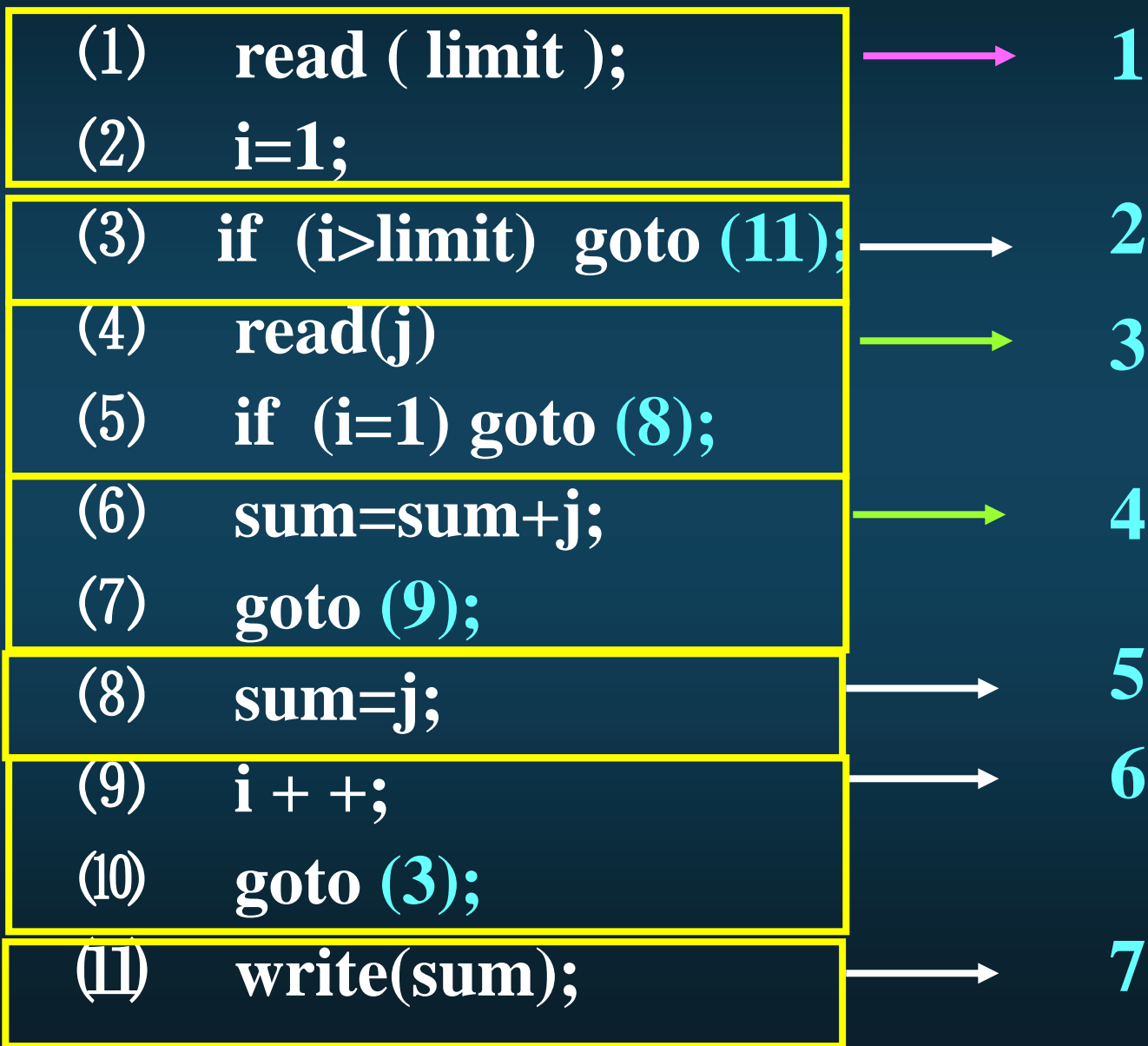
- (1) 程序的第一个语句；
- (2) 由条件转移语句或无条件转移语句转移到的语句；
- (3) 紧跟在条件转移或无条件转移后面的语句。

第2步：根据确定的基本块的入口语句，构造其所属的基本块。即：

- (1) 由该入口语句直到下一个入口语句(不包含下一个入口语句)之间的所有语句构成一个基本块；
- (2) 由该入口语句到一转移语句(含该转移语句)之间的所有语句构成一个基本块；或到程序中的停止或暂停语句(包含该停止或暂停语句)之间的语句序列组成的。

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。

例8.1 对如下程序段实施基本块的划分。



■ 基本块的确定

Step1: 求四元式序列各基本块的入口语句;

Step2: 对求出每一个的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;

■ 程序的控制流图

具有唯一首结点的有向图。流图G为

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

其中：

N：是流图的所有的结点组成的集合。流图中的结点为基本块。

n_0 ：是流图的首结点。

E：是流图的所有的有向边组成的集合。

■ 流图中的有向边 E_i 的形成:

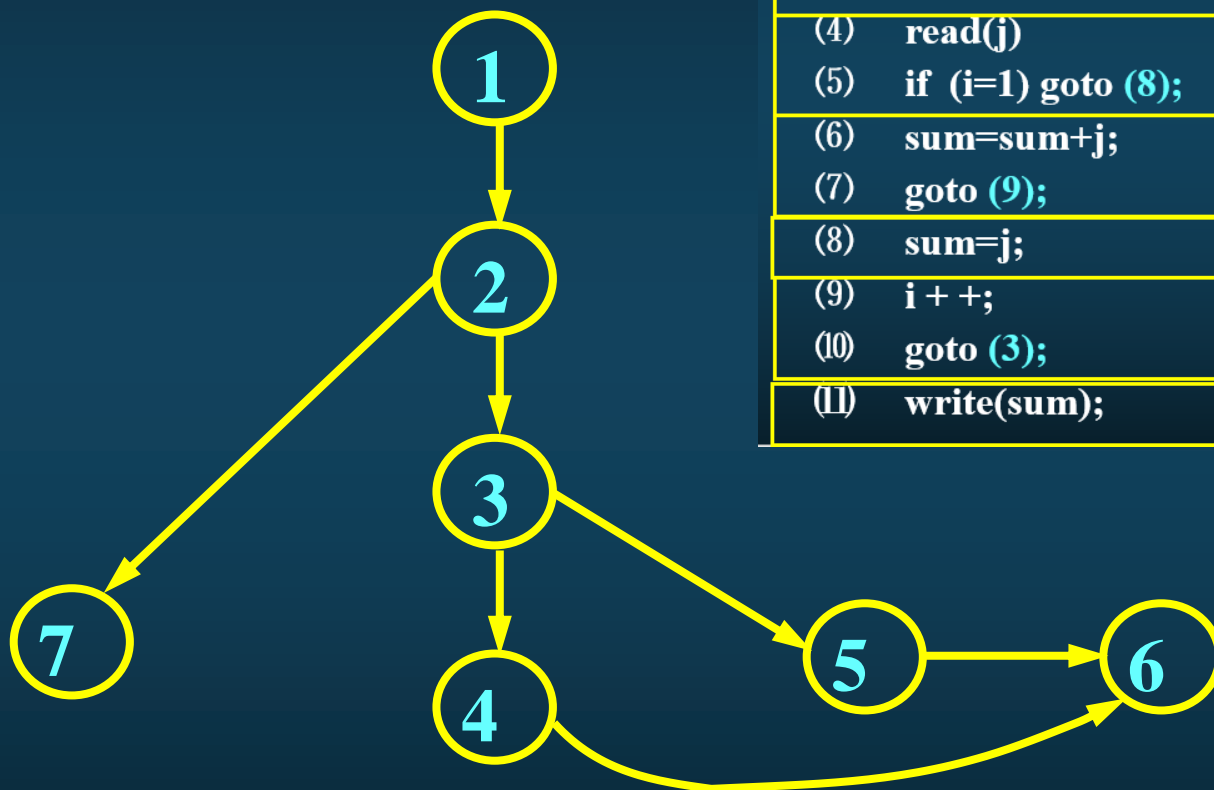
设有结点 i 到结点 k (或说从结点 i 到结点 k 由有向边 E_i 相连) 可表示为



其条件是

- ① 基本块 k 在流图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;
- ② 基本块 i 的出口语句是goto (s)或if...goto (s) 且(s)是基本块 k 的入口语句。

例8.1 程序的流图。



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

小结

■ 局部优化

指在程序的一个基本块内进行的优化。

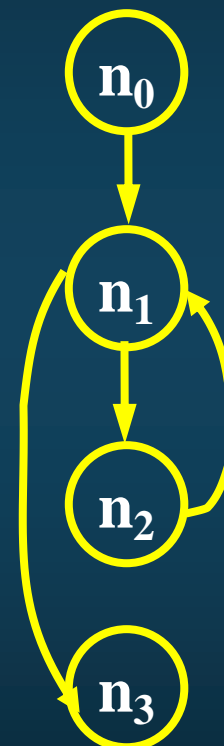
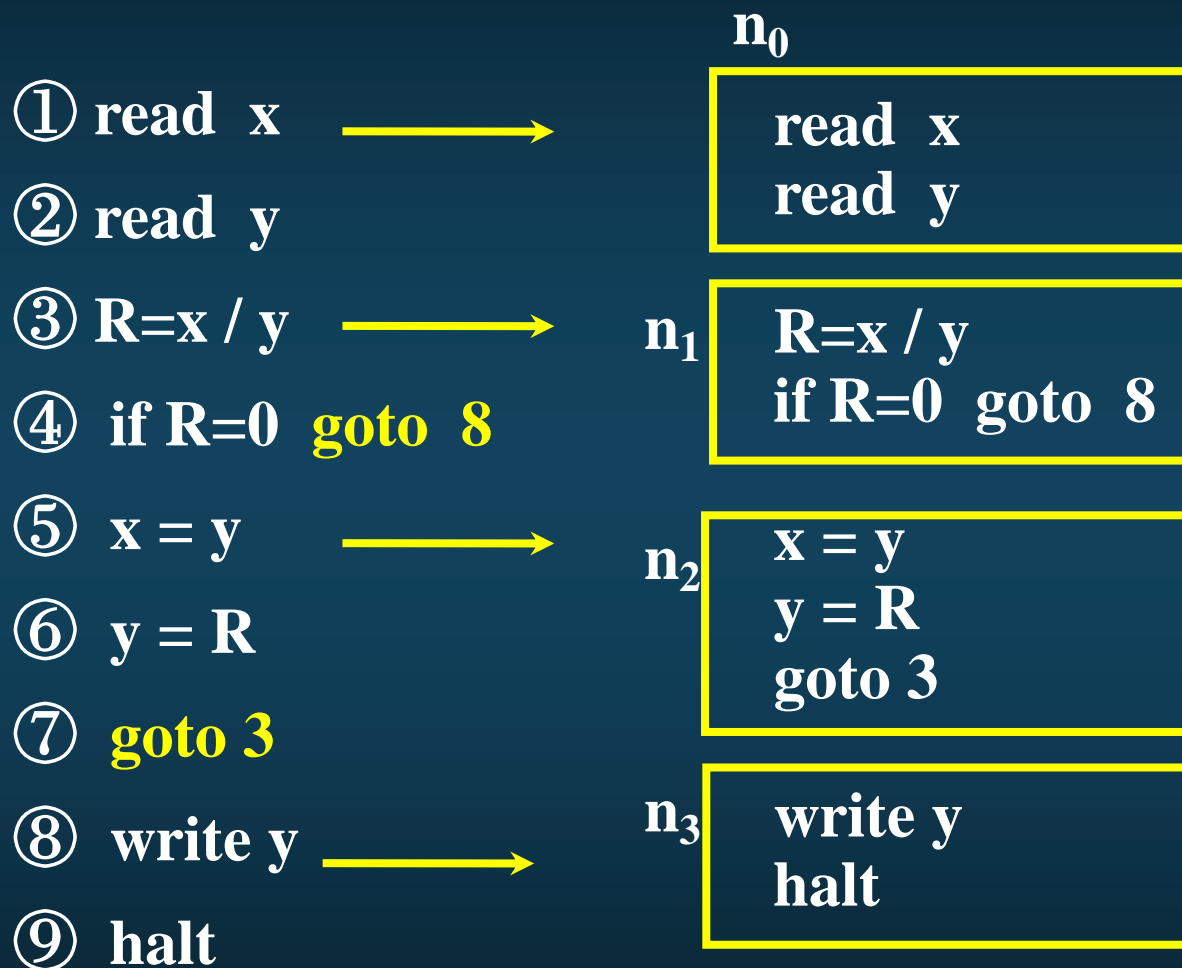
■ 基本块特点

基本块内的语句是顺序执行的，没有转进转出，分叉汇合。

■ **基本块**：单入口单出口的程序段。

■ **程序流图**：以基本块为结点的有向图，有向边表示程序执行的流程。

例8.2 对如下程序段划分基本块，给出流图。



■ DAG (Directed Acyclic Graph)

无环路的有向图。

■ 定义8.1

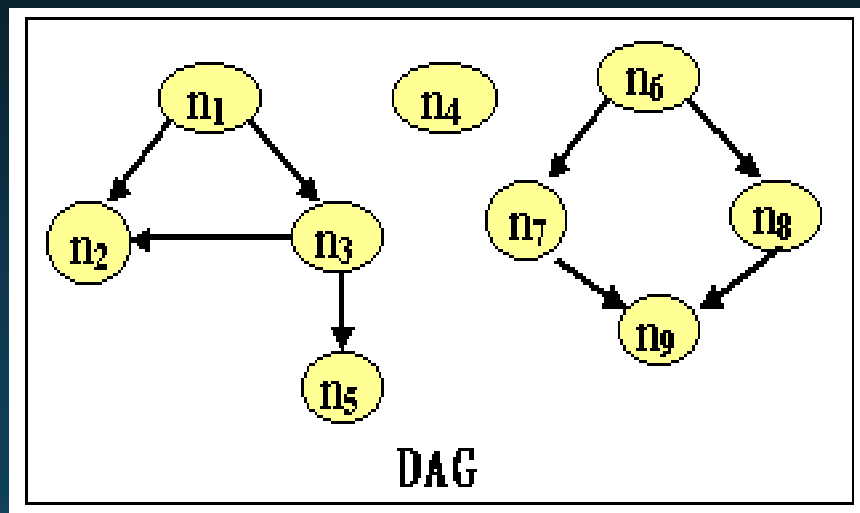
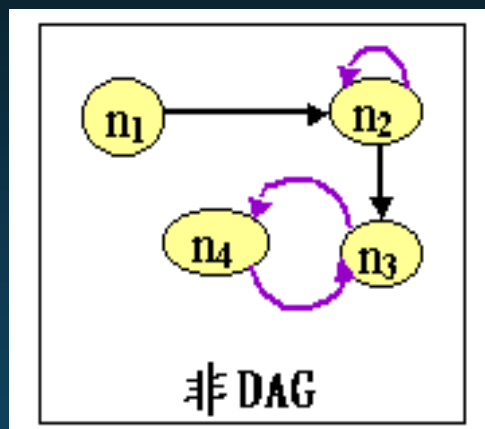
设G是由若干结点构成的有向图，从结点 n_i 到结点 n_j 的有向边用 $n_i \rightarrow n_j$ 表示。

- ① 若存在有向边序列 $n_{i1} \rightarrow n_{i2} \rightarrow \dots \rightarrow n_{im}$ ，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。路径上有向边的数目为**路径的长度**；
- ② 如果存在一条路径，其长度 ≥ 2 ，且该路径起始和结束于同一个结点，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

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惟一的标记，当叶结点是标识符时，代表名字的初值可加下标0；
- ② **内部结点**用运算符标记，同时也表示计算的值；
- 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。



DAG图中结点的特点:

1. 叶结点

标记: 标识符名(变量名)或常数, 写在结点下面;

代表: 该结点代表该变量或常数的值。

通常将其标识符加上下标0,表示该变量的初值。

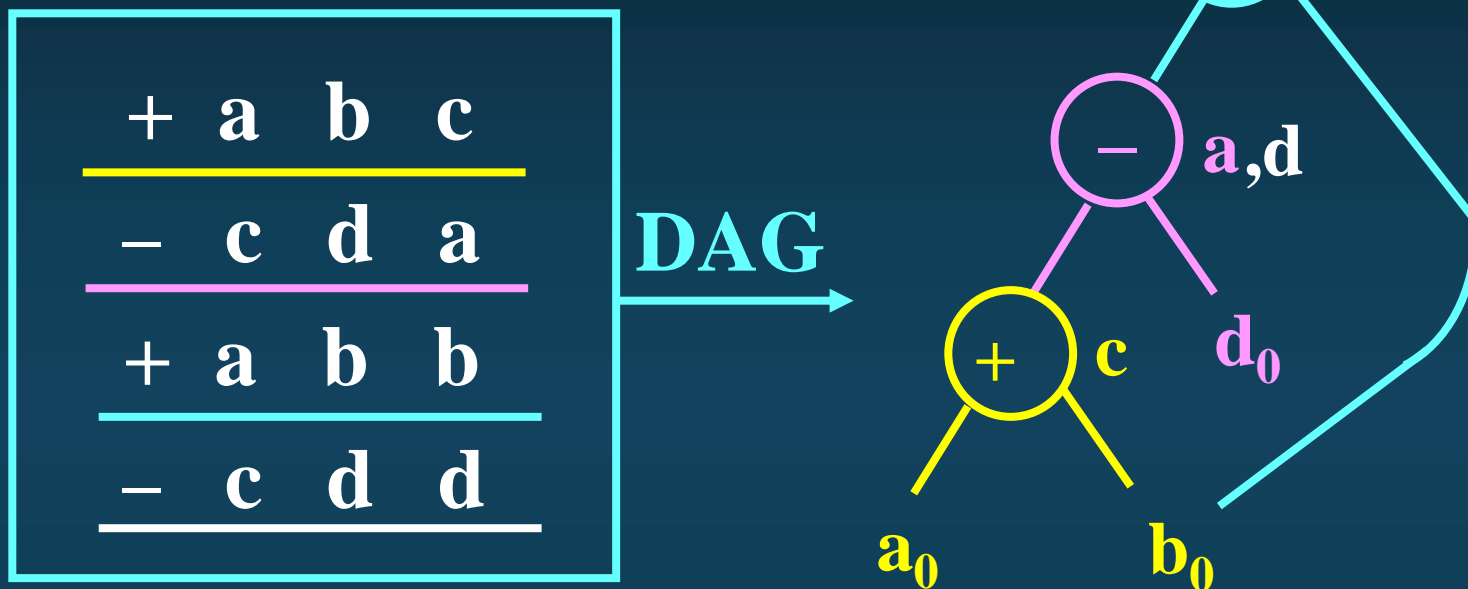
2. 内部结点

标记: 一个运算符号。写在结点下面。

代表: 利用后续结点运算出来的值。

3. 图中各个结点可能附加一个或多个标识符,表示这些标识符具有该结点所代表的值。写在结点右面。

例8.3 设有基本块如下



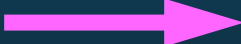
★ 注意:

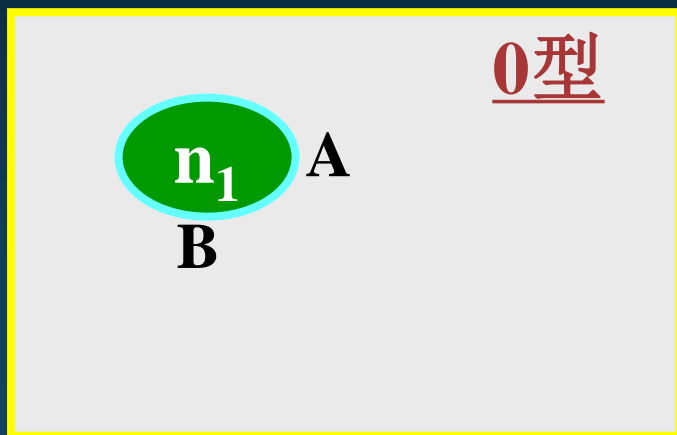
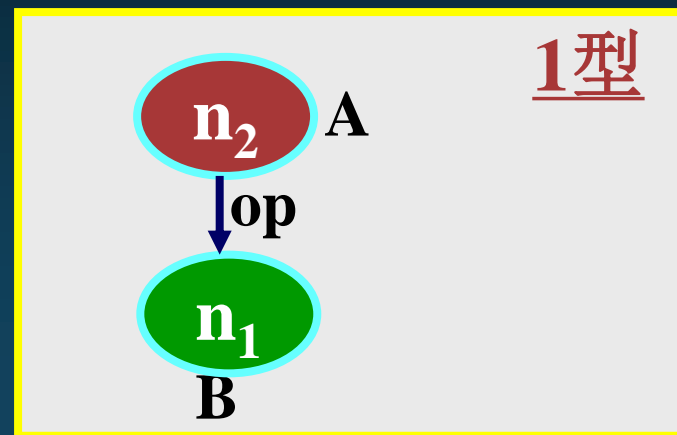
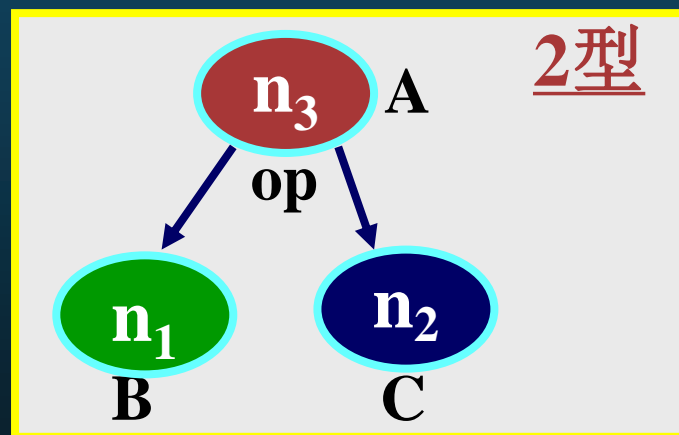
流图的一个结点是一个基本块，可用DAG表示。流图确认的是基本块之间的关系，DAG确认的是基本块内各四元式间的关系。

■ 常见四元式与DAG结点对应关系 P231

0型  一个结点(定值语句)

1型  二个结点 (单目运算
且定值)

2型  三个结点(双目运算、
取数组元素且定值，
条件句)

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

■ 常见四元式简化为下述三种情况

(1) $A = B$ 0型

(2) $A = \text{op } B$ 1型

(3) $A = B \text{ op } C$ 2型

■ 算法8.1 （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

输入： 一个基本块 B_i

输出： 含有下列信息的基本块 B_i 的DAG:

- (1) 叶结点、内部结点按统一标记;
- (2) 每个结点有一个标识符表（可空）;

算法：

对基本块中每一四元式依次执行以下步骤

1. 构造叶结点;
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
4. 捕捉可能的无用赋值; //删除

例8.4 设有一个基本块的语句序列如下

$$(1) \quad T_0 = 3.14$$

$$(2) \quad T_1 = 2 * T_0$$

$$(3) \quad T_2 = R + r$$

$$(4) \quad A = T_1 * T_2$$

$$(5) \quad B = A$$

$$(6) \quad T_3 = 2 * T_0$$

$$(7) \quad T_4 = R + r$$

$$(8) \quad T_5 = T_3 * T_4$$

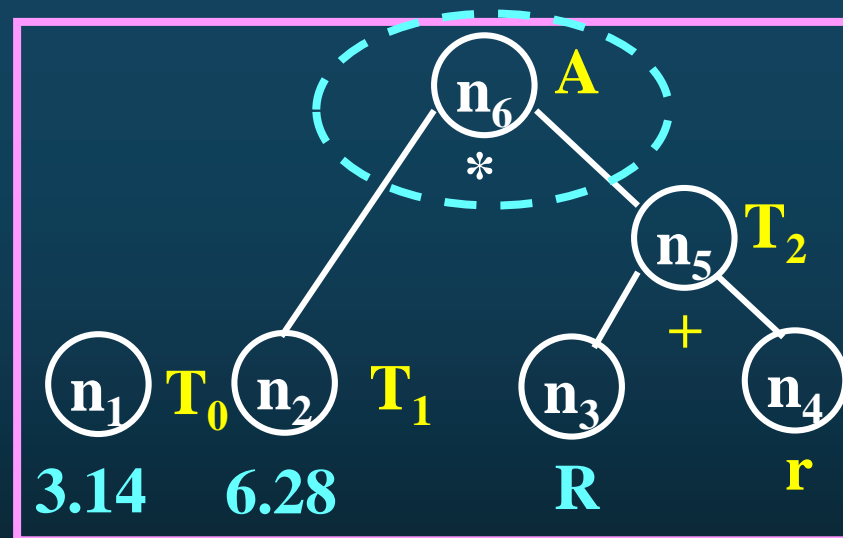
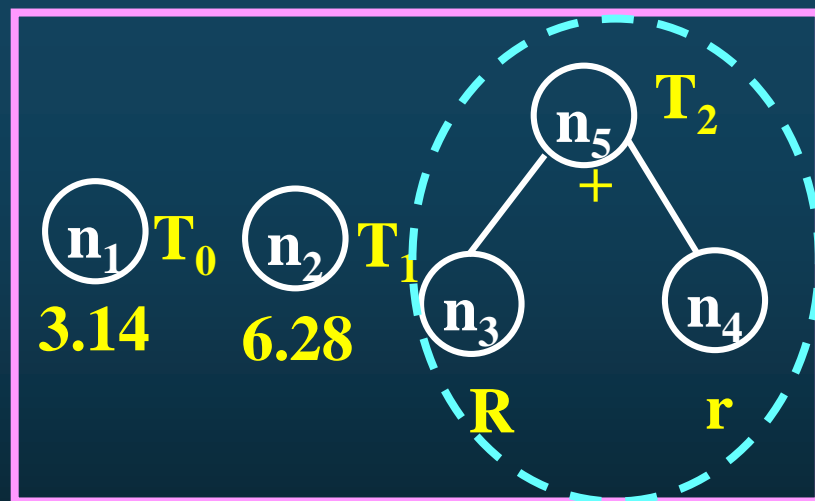
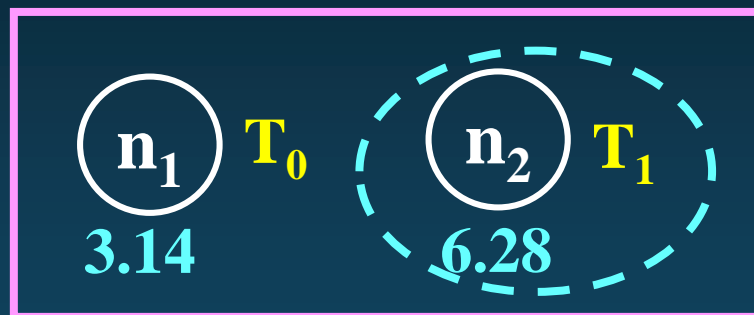
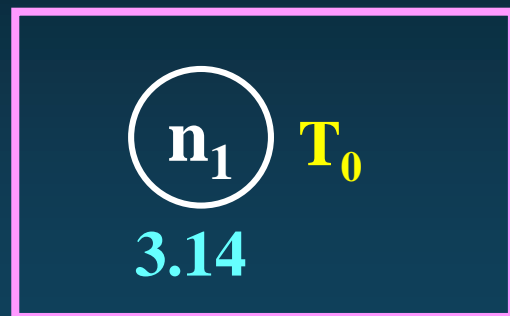
$$(9) \quad T_6 = R - r$$

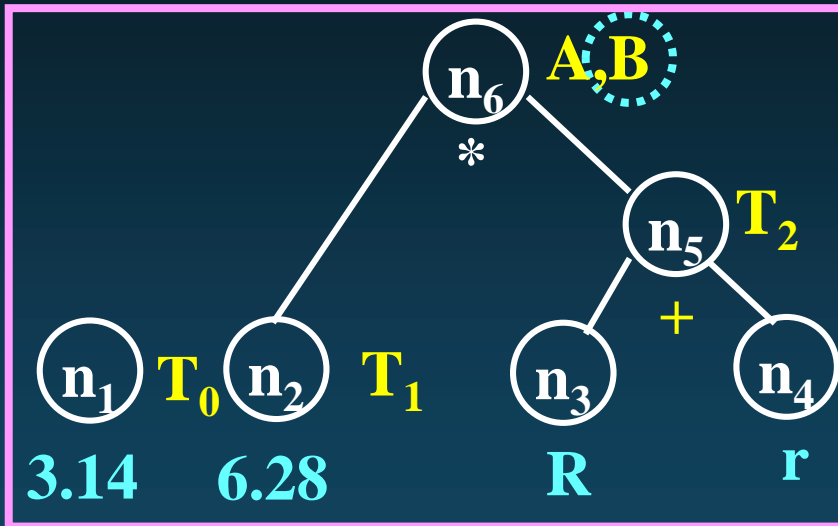
$$(10) \quad B = T_5 * T_6$$

P233_ 例8.4

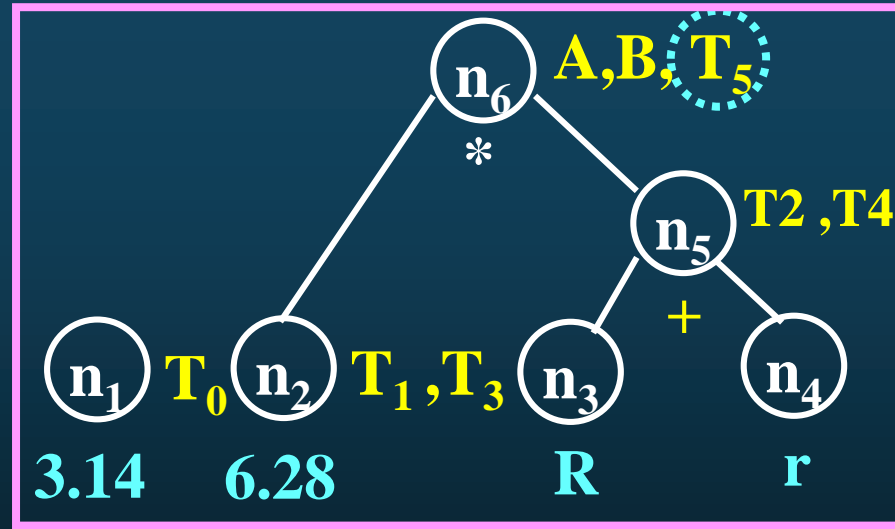
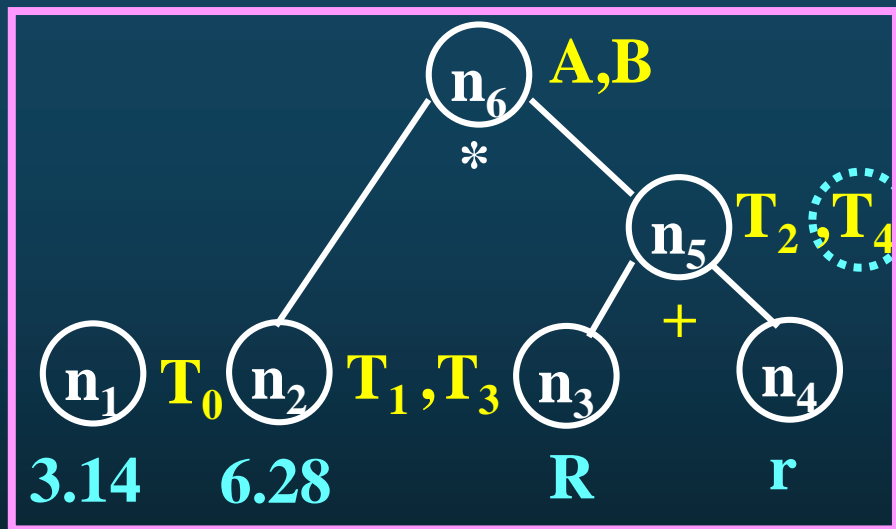
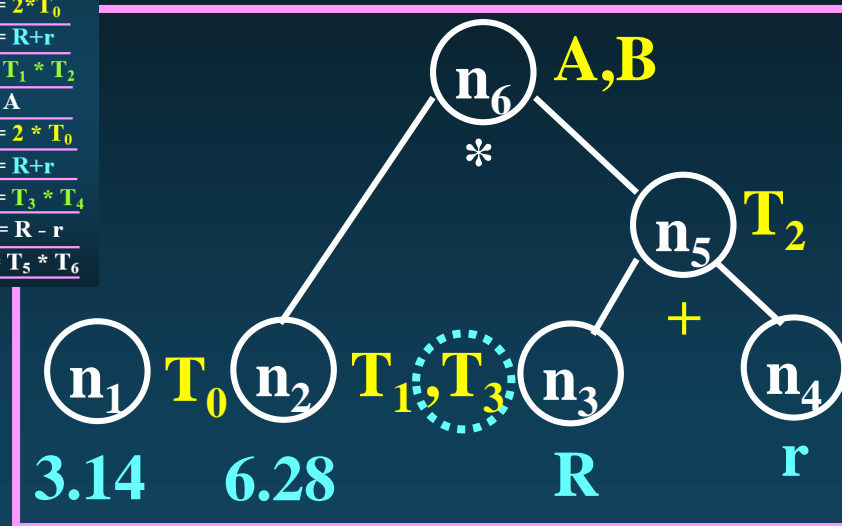
解：构造DAG的过程如下：

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

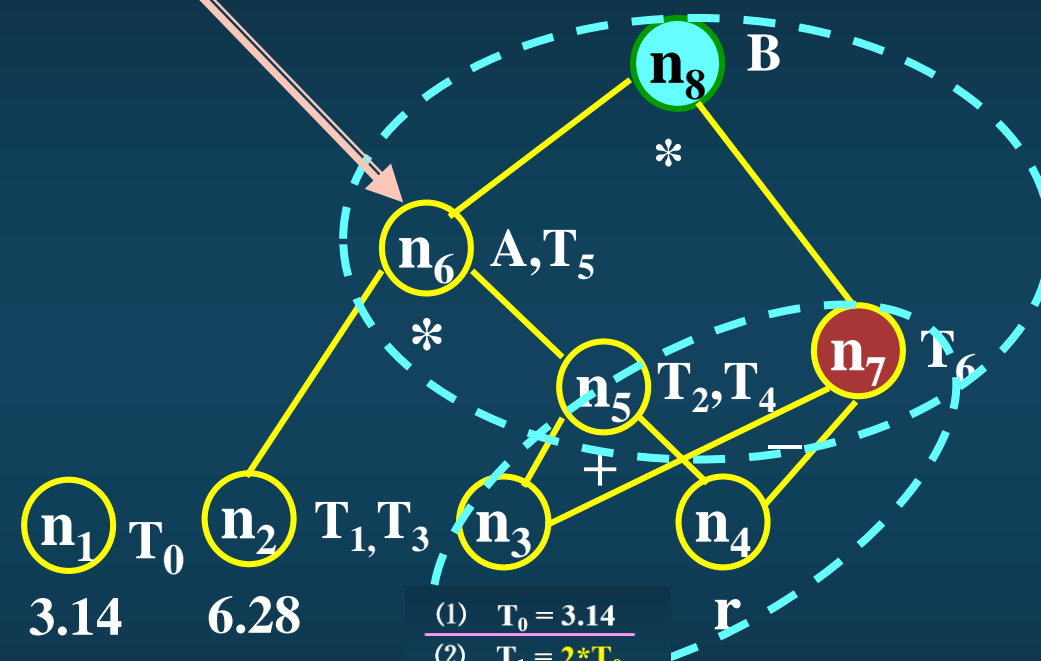




- (1) $T_0 = 3.14$
- (2) $T_1 = 2 * T_0$
- (3) $T_2 = R + r$
- (4) $A = T_1 * T_2$
- (5) $B = A$
- (6) $T_3 = 2 * T_0$
- (7) $T_4 = R + r$
- (8) $T_5 = T_3 * T_4$
- (9) $T_6 = R - r$
- (10) $B = T_5 * T_6$



原附在 n_6 的B删除



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

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

■ 本节思路

- **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为一个循环：

1. 强连通性。

流图中若存在任意两个节点之间必有一条通路，则通路上的任何节点都属于该循环。

2. 入口惟一。

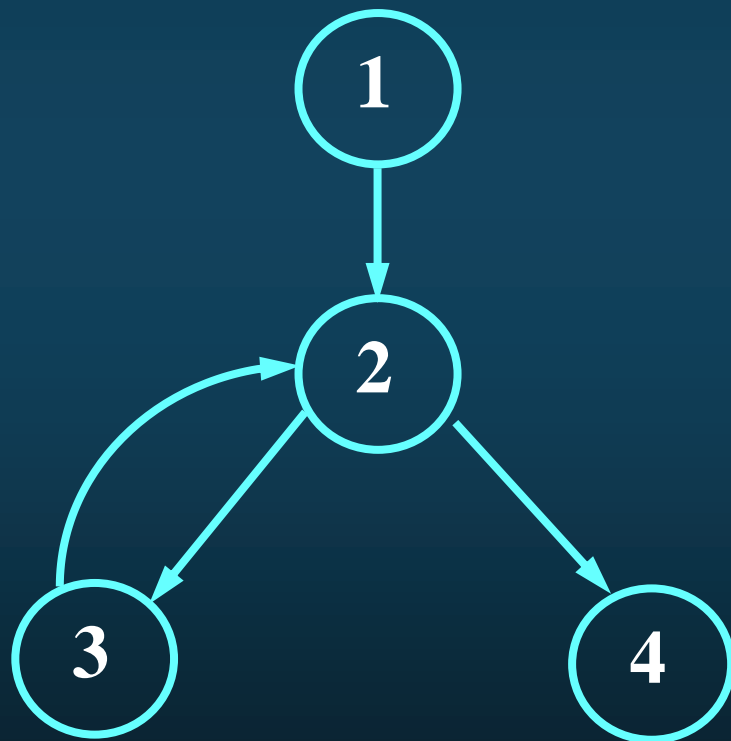
入口是流图的首结点或结点序列外某结点有一条有向边引到它。

■ 定义8.2 （循环）

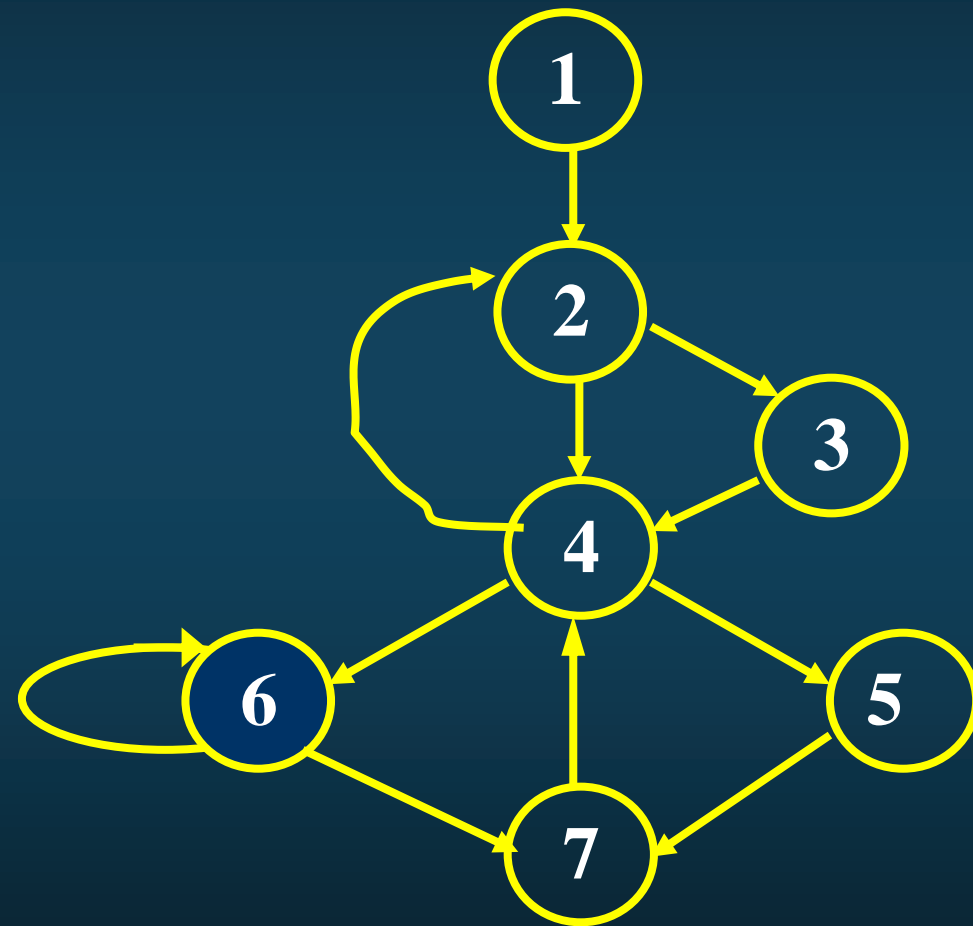
程序流图中具有惟一入口结点的强连通子图。

例如右图，

{2, 3}是循环 { 强连通性成立
惟一结点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8.3 （必经结点）

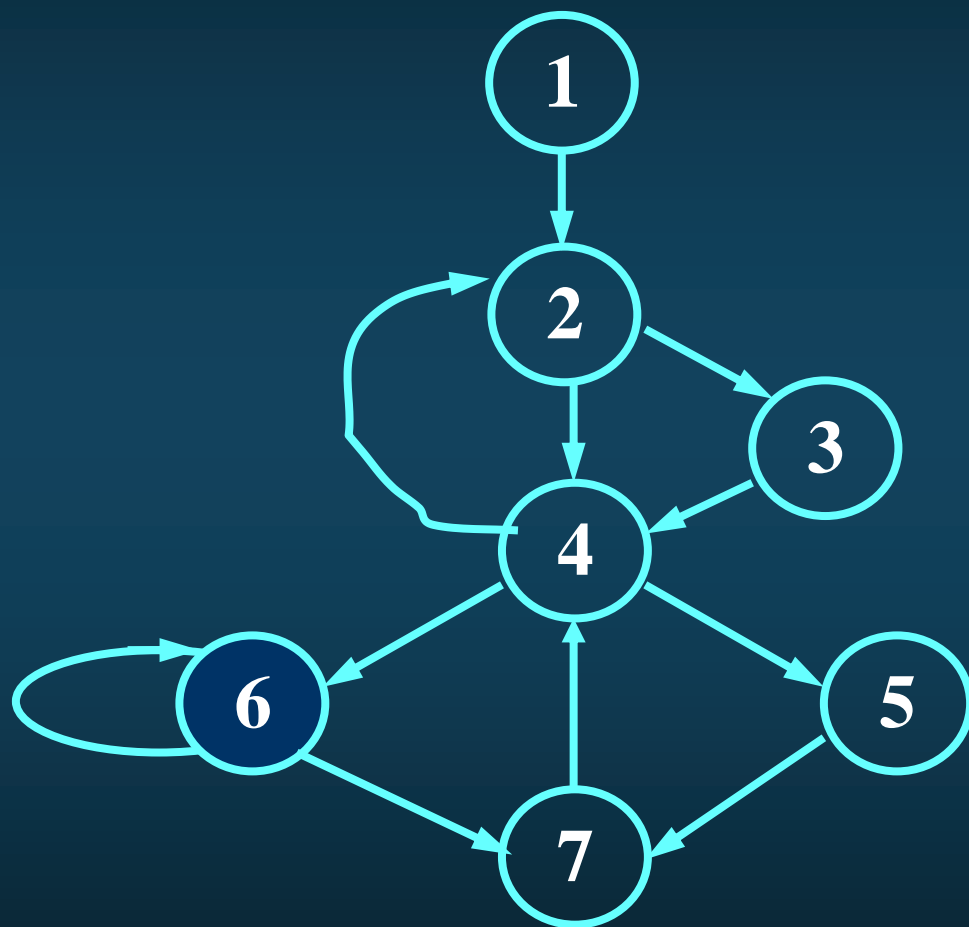
在程序流图G中， n_i 和 n_j 为任意结点。若从 n_0 出发，到达 n_j 的**任何**一条通路都必经过 n_i ，则称 n_i 是 n_j 的必经结点，记作 n_i **DOM** n_j 。

■ 定义8.4 （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**: $a \text{ DOM } a$
 - (2) **传递性**: 如果 $a \text{ DOM } b$, $b \text{ DOM } c$,
则有: $a \text{ DOM } c$ 。
 - (3) **反对称性**: 若有 $a \text{ DOM } b$, $b \text{ DOM } a$,
则有: $a = b$ 。

例8.5 设有如下流图



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

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

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

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

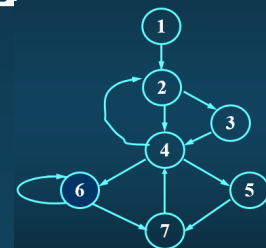
$$D(5) = \{1, 2, 4, 5\}$$

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

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

■ 定义8.5 (回边)

设 $a \rightarrow b$ 是流图 G 中一条有向边, 如果 $b \text{ DOM } a$, 则称 $a \rightarrow b$ 是流图 G 中的一条回边。记作 $\langle a, b \rangle$ 。



例7.5 流图中存在有向边 $6 \rightarrow 6$, $7 \rightarrow 4$ 和 $4 \rightarrow 2$ 。

并且有

皆为回边

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

则 $6 \text{ DOM } 6$,

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

则 $4 \text{ DOM } 7$,

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

则 $2 \text{ DOM } 4$ 。

■ 利用回边求出流图中的循环：

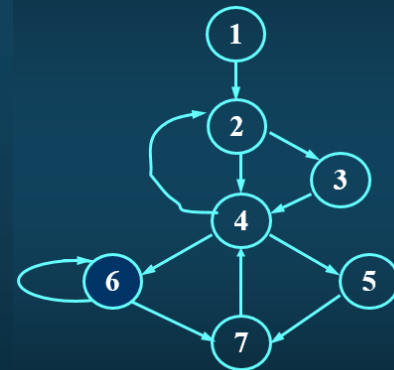
若 $\langle n, d \rangle$ 是一回边，则由结点 d 、结点 n 以及所有通路到达 n 而该通路不经过 d 的**所有结点**序列构成一个循环 L ， d 是循环 L 的惟一入口。

例8.5 流图中的循环：

$$\langle 6, 6 \rangle_{\text{loop}} = \{ 6 \}$$

$$\langle 7, 4 \rangle_{\text{loop}} = \{ 4, 5, 6, 7 \}$$

$$\langle 4, 2 \rangle_{\text{loop}} = \{ 2, 3, 4, 5, 6, 7 \}$$



例

➤ **summary** (查找循环步骤)

1. 确定G的D(n);
2. 由D(n)找回边;
3. 通过回边确定循环。

一. 局部优化

1. 基本块定义 { 入口
出口

2. 实施 — DAG { 构造: 中间code → DAG
重建: 已优化 code ←

二. 循环优化

技术准备 { 控制流分析
数据流分析

控制流分析 → Loop

中间code → 基本块 → G → D(n) → 回边

数据流分析: 中间code + 控制流 → 采集 → 优化所需信息