

## 第九章 代码生成



# 学习内容

---

- 目标机器
- 存储管理
- 基本块、流图
- 简单代码生成算法
- 寄存器分配
- 窥孔优化
- 基本块dag表示、从dag生成代码



## 9.1 设计中的问题

---

### 1. 输入

- 前端生成的中间表示形式

- 线性化表示：后缀表示形式
- 四元式表示：三地址码
- 虚拟机表示：抽象栈机器代码
- 图形化表示：语法树、DAG

- 符号表信息

- 类型检查、类型转换已完成

- 输入是无错误的



# 设计中的问题（续）

---

## 2. 目标程序

### □ 绝对机器语言

- 固定内存地址，可直接执行

### □ 可重定位机器语言

- 多模块连接
- 灵活、分别编译

### □ 汇编语言

- 简单——符号指令、宏的使用



## 设计中的问题（续）

---

### 3. 内存管理

- ❑ 名字→数据地址，与前端协作
- ❑ 符号表
- ❑ 标号→指令地址
- ❑ backpatching技术



## 设计中的问题（续）

---

### 4. 指令选择

- 指令集特性

- 一致性、完整性
- 指令速度、机器特性

- 代码质量：速度、大小

- 丰富的指令集→多种代码生成方式

- 选择最高效方式



## 设计问题（续）

---

### 5. 寄存器分配

- allocation、assignment
- 最优化assignment——NP-完全问题
- register-pair

### 6. 计算顺序

- 调整计算顺序→提高效率
- NP-完全问题



## 设计问题（续）

---

### 7. 代码生成方法

- 正确性是第一位的
- 设计目标：易于实现、测试、维护
- 9.6节：生成算法，9.9节：窥孔优化
- 9.7节：寄存器使用算法——控制流
- 9.10、9.11节：代码选择
- 9.12节：代码生成——树重构





## 9.2 目标机器

- 寄存器:  $R0, R1, \dots, R_{n-1}$
- 指令:  $op \text{ source, destination}$
- 寻址方式

寻址方式	形式	地址	额外开销
绝对	M	M	1
寄存器	R	R	0
索引	$c(R)$	$c + \text{contents}(R)$	1
间接寄存器	$*R$	$\text{contents}(R)$	0
间接索引	$*c(R)$	$\text{contents}(c + \text{contents}(R))$	1

MOV R0, M

MOV 4(R0), M

MOV \*4(R0), M



# 指令开销

---

## ○ 长度

1. MOV R0, R1: 开销1
2. MOV R5, M: 开销2
3. ADD #1, R3: 开销2
4. SUB 4(R0), \*12(R1): 开销3



# 指令开销——翻译方法

○  $a = b + c$

1. MOV b, R0      开销6

ADD c, R0

MOV R0, a

2. MOV b, a      开销6

ADD c, a

3. MOV \*R1, \*R0      开销2

ADD \*R2, \*R0

4. ADD R2, R1      开销3

MOV R1, a

R1: b的值

R2: c的值

R0: a的地址

R1: b的地址

R2: c的地址



## 9.3 运行时存储管理 (※)

---

- 静态分配，栈分配
- 翻译如下三地址码
  - call
  - return
  - halt
  - action

## 9.3.1 静态分配

○ call翻译为

MOV #here, callee.static\_area

GOTO callee.code\_area

○ return翻译为

GOTO \*callee.static\_area

当前代码地址  
被调用函数的

静态区域  
被调用函数的

代码区域



## 例9.1

### 三地址码

/* c的代码 */ action <sub>1</sub> call p action <sub>2</sub> halt
/* p的代码 */ action <sub>1</sub> return

### c的活动记录 (64字节)

0:	return address
8:	arr
56:	i
60:	j

### p的活动记录 (64字节)

0:	return address
8:	buf
84:	n



## 例9.1（续）

---

100: ACTION<sub>1</sub>

120: MOV #140, 364

132: GOTO 200

140: ACTION<sub>2</sub>

160: HALT

...

200: ACTION<sub>3</sub>

220: GOTO \*364

...

/\* 300-363: c的活动记录 \*/

300: /\* 返回地址 \*/

304: /\* 局部数据 \*/

...

/\* 364-451: p的活动记录 \*/

364: /\* 返回地址 \*/

368: /\* 局部数据 \*/



## 9.3.2 栈分配

---

- 使用相对活动记录起始的偏移
- 活动记录的地址——栈寄存器，SP
- “第一个过程”

`MOV #stackstart, SP`

第一个过程的代码

`HALT`





# 栈分配——函数调用和返回

## ○ 调用

```
ADD #caller.recordsize, SP  /* 指向被调函数活  
                           动记录 */
```

```
MOV #here + 16, *SP        /* 保存返回地址 */
```

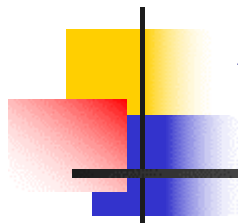
```
GOTO callee.code_area
```

## ○ 返回

```
GOTO *0(SP)
```

## ○ 栈寄存器调整回调用者的活动记录

```
SUB #caller.recordsize, SP
```



## 例9.2

### 三地址码

/\* s的代码 \*/

action<sub>1</sub>

call q

action<sub>2</sub>

halt

/\* p的代码 \*/

action<sub>3</sub>

return

/\* q的代码 \*/

action<sub>4</sub>

call p

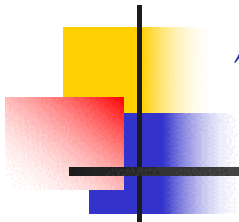
action<sub>5</sub>

call q

action<sub>6</sub>

call q

return



## 例9.2（续）

```
/* s的代码 */
100: MOV #600, SP    /* 初始化栈 */
108: ACTION1
128: ADD #ssize, SP /* 调用序列开始 */
136: MOV #152, *SP  /* 返回地址压栈 */
144: GOTO 300       /* call q */
152: SUB #ssize, SP /* 恢复栈 */
160: ACTION2
180: HALT
...

/* p的代码 */
200: ACTION3
220: GOTO *0(SP)    /* return */
...
```



## 例9.2（续）

```
/* q的代码 */
300: ACTION4 /* 条件转移到456 */
320: ADD #qsize, SP
328: MOV #344, *SP /* 返回地址压栈 */
336: GOTO 200 /* call p */
344: SUB #qsize, SP
352: ACTION5
372: ADD #qsize, SP
380: MOV #396, *SP /* 返回地址压栈 */
388: GOTO 300 /* call q */
396: SUB #qsize, SP
404: ACTION6
424: ADD #qsize, SP
432: MOV #448, *SP /* 返回地址压栈 */
440: GOTO 300 /* call q */
448: SUB #qsize, SP
456: GOTO *0(SP) /* return */
...
600:
```



### 9.3.3 运行时名字的地址

---

○  $x := 0$

- 相对地址12
- 位于静态分配区域，地址static
- $\text{static}[12] := 0$
- $\text{static} = 100 \rightarrow \text{MOV } \#0, 112$
- display方式，地址在寄存器R3中
  - $t_1 := 12 + R3$
  - $*t_1 := 0$
  - $\text{MOV } \#0, 12(R3)$



## 9.4 基本块和流图

### ○ 9.4.1 基本块, **basic block**

□ 连续语句序列, 执行过程中没有分支

□  $t_1 := a * a$

$t_2 := a * b$

$t_3 := 2 * t_2$

$t_4 := t_1 + t_3$

$t_5 := b * b$

$t_6 := t_4 + t_5$

□  $x := y + z$ : 定义 $x$ , 使用(引用)  $y$ 、 $z$



## 算法9.1 基本块的划分

输入 三地址码序列

输出 划分后的基本块

方法

1. 首先确定入口语句（**leader**，基本块的第一个语句）

集，确定规则：

- i. 程序的第一条语句
- ii. 条件转移或无条件转移语句的目的语句
- iii. 条件转移或无条件转移语句之后的语句

1. 每个leader对应的基本块：

leader——下一个leader（或程序尾）



## 例9.3

---

begin

    prod := 0

    i := 1

    do begin

        prod := prod + a[i] \* b[i];

        i := i + 1

    end

    while i <= 20

end





## 例9.3（续）

---

1.  $\text{prod} := 0$
2.  $i := 1$
3.  $t_1 := 4 * i$
4.  $t_2 := a[t_1]$
5.  $t_3 := 4 * i$
6.  $t_4 := b[t_3]$
7.  $t_5 := t_2 * t_4$
8.  $t_6 := \text{prod} + t_5$
9.  $\text{prod} := t_6$
10.  $t_7 := i + 1$
11.  $i := t_7$
12. if  $i \leq 20$  goto (3)



## 9.4.2 基本块的变换

---

- 基本块进行表达式计算
- 等价——计算相同的表达式
- 变换——不改变计算的表达式集合
- 提高代码质量
- 保结构变换
  - structure-preserving transformation
- 代数变换
  - algebraic transformation



## 9.4.3 保结构变换

### 1. 公共子表达式删除

$a := b + c$

$b := a - d$

$c := b + c$

$d := a - d$

→

$a := b + c$

$b := a - d$

$c := b + c$

$d := b$

$d$ 与 $b$ 相同  
 $c$ 与 $a$ 不同!



## 保结构变换（续）

### 2. 无用代码删除

$x := y + z$

$x$ 在后续代码中未被使用，则可将此语句删除

### 3. 重命名临时变量

$t := b + c$ ,  $t$ ——临时变量 →

$u := b + c$ ,  $u$ ——新临时变量，对 $t$ 的引用→对 $u$ 的引用，  
基本块结果不变

定义临时变量的语句都定义新的临时变量→

基本块等价变换，范式基本块



## 保结构变换（续）

### 4. 语句交换

$t_1 := b + c$

$t_2 := x + y$

两个语句交换位置不影响运算结果 $\leftrightarrow$

$x$ 、 $y$ 都不是 $t_1$ ， $b$ 、 $c$ 都不是 $t_2$

**注意：**范式基本块允许所有可能的语句交换



## 9.4.4 代数变换

---

○ 允许改变表达式——代数上等价

□  $x := x + 0$ ——删除

□  $x := x * 1$ ——删除

□  $x := y ** 2 \rightarrow x := y * y$ ——提高性能



## 9.4.5 流图

---

- 基本块间添加控制流信息→程序
- 流图, **flow graph**
  - 节点——基本块
  - 首 (**initial**) 节点——该基本块的入口语句就是程序的第一条语句



## 流图的构造

- 基本块 $B_1$ 到 $B_2$ 有一条有向边 $\leftarrow\rightarrow$ 代码执行序列中 $B_2$ 紧跟在 $B_1$ 之后：
  1.  $B_1$ 的最后一条语句（无）条件转向到 $B_2$ 的第一条语句
  2. 程序中 $B_2$ 紧跟在 $B_1$ 之后，且 $B_1$ 的最后一条语句不是无条件转移语句
- $B_1$ —— $B_2$ 的前驱， $B_2$ —— $B_1$ 的后继





## 例9.4

```
prod := 0  
i := 1
```

```
t1 := 4 * i  
t2 := a [ t1 ]  
t3 := 4 * i  
t4 := b [ t3 ]  
t5 := t2 * t4  
t6 := prod + t5  
prod := t6  
t7 := i + 1  
i := t7  
if i <= 20 goto (3)
```



## 9.4.6 基本块的表示

---

### ○ 记录

- 计数器：四元组（三地址码语句）数目
- 指向leader的指针
- 指向前驱基本块和后继基本块的指针

### ○ 优化时代码改变或移动

- 跳转到语句号 → 跳转到基本块



## 9.4.7 循环 (loop)

- 什么是循环？如何找到循环？
- 循环——满足如下条件的一组节点
  1. 这组节点是强连通的——任何两个节点间都存在一条（完全包含在循环内的）路径
  2. 这组节点具有唯一的一个入口 (entry) ——从循环外的节点到达循环内的节点，唯一的途径是先到达入口节点
- 内部不包含其他循环的循环——内层循环，  
inner loop



## 9.5 下次引用信息

- next-use information
- 名字的使用（use）
  - 三地址码语句i为x赋值
  - 语句j将x作为运算对象，而i到j的控制流路径中无其他对x赋值的语句
  - 语句j使用了语句i计算的x值
- 对语句 $x := y \text{ op } z$ ，确定x、y、z下次使用的位置，以决定寄存器可否释放
- 由后向前扫描基本块



## 9.5.1 计算下次引用信息

### ○ 基本方法

- 每个变量记录下次引用信息和活跃信息
- 假定每个临时变量在基本块出口后非活跃
- 非临时变量在出口后活跃
- 若临时变量跨基本块，假定其活跃

### ○ 算法：当扫描到语句 $i$ : $x := y \text{ op } z$

1. 将 $x$ 、 $y$ 、 $z$ 的下次引用信息和活跃信息附加在语句 $i$ 之上
2. 设置 $x$ 为“非活跃”和“没有下次引用”
3. 设置 $y$ 、 $z$ 为“活跃”，下次引用信息设置为 $i$

不可  
交换！



## 9.5.2 临时名字的存储分配

- 两个临时名字活跃期不重叠  $\rightarrow$  相同地址
- 利用下次使用信息

$t_1$  {  $t_2$  {  $t_1 := a * a$   
 $t_2 := a * b$   
 $t_3$  {  $t_3 := 2 * t_2$   
 $t_4$  {  $t_4 := t_1 + t_3$   
 $t_5$  {  $t_5 := b * b$   
 $t_6 := t_4 + t_5$



$t_1 := a * a$   
 $t_2 := a * b$   
 $t_2 := 2 * t_2$   
 $t_1 := t_1 + t_2$   
 $t_2 := b * b$   
 $t_1 := t_1 + t_2$



## 9.6 一个简单的代码生成器

- 每个中间语言指令  $\rightarrow$  目标语言指令
- 计算结果尽量保存在寄存器，除非
  - a) 需要用寄存器进行其他计算
  - b) 下面语句是函数调用、转移或标号基本块结束！
- $a := b + c$ 
  - $R_i = b, R_j = c \rightarrow \text{ADD } R_j, R_i$ ——开销1
  - $R_i = b \rightarrow \text{ADD } c, R_i$ ——开销2
  - 或  $\rightarrow \text{MOV } c, R_j \quad \text{ADD } R_j, R_i$ ——开销3



## 9.6.1 寄存器描述符和地址描述符

- 代码生成程序用来保存寄存器内容和名字的地址

### 1. 寄存器描述符:

当前每个寄存器内容，分配新寄存器时用

初始：所有寄存器均空

代码生成过程中：保存0个或多个名字值

### 2. 地址描述符:

名字的当前值保存在何处？

寄存器、栈位置或是内存地址

可能在多个位置





## 9.6.2 代码生成算法

○ 输入：基本块，对每个语句  $x := y \text{ op } z$ ：

1. 为计算结果分配位置  $L$ ——getreg

2. 提取  $y$  的值

➤ 查询  $y$  的位置  $y'$  ——地址描述符，寄存器优先

➤  $y' \neq L$  ——MOV  $y'$ ,  $L$

3. 生成计算指令

➤ 查询  $z$  的位置  $z'$

➤ OP  $z'$ ,  $L$

➤ 更新  $x$  的地址描述符和  $L$  的寄存器描述符

4. 若  $y$ 、 $z$  不再被引用，从寄存器描述符删除



## 特殊情况

- 一元运算：省略z的部分
- $x := y$ ，根据y的位置
  - 寄存器L：修改寄存器描述符和地址描述符  
→x也（仅）在L中
  - 内存：getreg，得到的寄存器保存x和y
  - x不再被引用：MOV y, x



## 特殊情况

---

- 基本块出口，将活跃名字值存入内存
  - 寄存器描述符 → 哪些名字保存在寄存器中
  - 地址描述符 → 名字对应的内存地址
  - 活跃信息 → 是否需要保存



## 9.6.3 getreg算法

- $x := y \text{ op } z$ , 为 $x$ 分配位置 $L$ , 效率高 $\rightarrow$ 低
  1. 与 $y$ 占用相同寄存器
    - $y$ 的寄存器 $r$ 不包含其他名字
    - $y$ 不再活跃、不再被引用
  2. 1) 失败, 寻找一个空寄存器
  3. 2) 失败, 寄存器替换
    - $x$ 还会被引用, 或 $op$ 操作需要使用寄存器——数组索引
    - 寻找已用寄存器 $R$ ,  $x$ 替换其中变量
    - $R$ 中变量引用位置最远, 或已保存在内存中
  4.  $x$ 不再被引用, 或无合适寄存器——内存

## 例9.5 $d := (a - b) + (a - c) + (a - c)$

三地址码	目标代码	寄存器描述符	地址描述符
		寄存器均为空	
$t := a - b$	MOV a, R0 SUB b, R0	R0包含t	t在R0中
$u := a - c$	MOV a, R1 SUB c, R1	R0包含t R1包含u	t在R0中 u在R1中
$v := t + u$	ADD R1, R0	R0包含v R1包含u	u在R1中 v在R0中
$d := v + u$	ADD R1, R0 MOV R0, d	R0包d	d在R0中 d在R0和 内存中

## 9.6.4 其他类型语句的代码生成

### ○ 数组和指针

语句	i 在寄存器Ri 中		i 在内存Mi 中		i 在栈中	
	代码	开销	代码	开销	代码	开销
$a := b[i]$	MOV b(Ri), R	2	MOV Mi, R MOV b(R), R	4	MOV Si(A), R MOV b(R), R	4
$a[i] := b$	MOV b, a(Ri)	3	MOV Mi, R MOV b, a(R)	5	MOV Si(A), R MOV b, a(R)	5

语句	p 在寄存器Rp 中		p 在内存Mp 中		p 在栈中	
	代码	开销	代码	开销	代码	开销
$a := *p$	MOV *Rp, a	2	MOV Mp, R MOV *R, R	3	MOV Sp(A), R MOV *R, R	3
$*p := a$	MOV a, *Rp	2	MOV Mp, R MOV a, *R	4	MOV a, R MOV R, *Sp(A)	5



## 9.6.5 条件转移语句

### ○ 两种实现方式

- 寄存器值符合六个条件之一：负数、零、正数、非负数、非零、非正数

if  $x < y$  goto  $z \rightarrow x - y$ , if 负数 goto  $z$

- 条件代码

if  $x < y$  goto  $z \rightarrow$     `CMP x, y`  
  `CJ< z`

$x := y + z \quad \rightarrow$     `MOV y, R0`  
if  $x < 0$  goto  $z$     `ADD z, R0`  
                                  `MOV R0, x`  
                                  `CJ< z`



## 9.7 寄存器分配和指定

- 寄存器操作比内存操作代码短，速度快
- 分配：确定哪些值保存在寄存器中
- 指定：确定每个值具体保存在哪个寄存器中
- 寄存器分组
  - ▣ 基地址、数学运算、栈地址…
  - ▣ 简单、低效





## 9.7.1 全局寄存器分配

- 9.6节，基本块出口，寄存器→内存
  - 避免复制，将最常用的变量保持在寄存器中
  - 跨越基本块边界——全局
  - 循环
  - 使用固定一组寄存器保存每个内存循环中最活跃的变量
  - C运行程序员指定寄存器分配



## 9.7.2 引用计数

- 假定访问寄存器比访问内存节省开销1
- 循环L，定义x
  - ▣ 9.6节：定义之后若有引用，x将保留在寄存器中，而定义之前的引用需访问内存
  - ▣ 优化：x一直保存在寄存器中，定义之前的引用变为访问寄存器，节省开销1

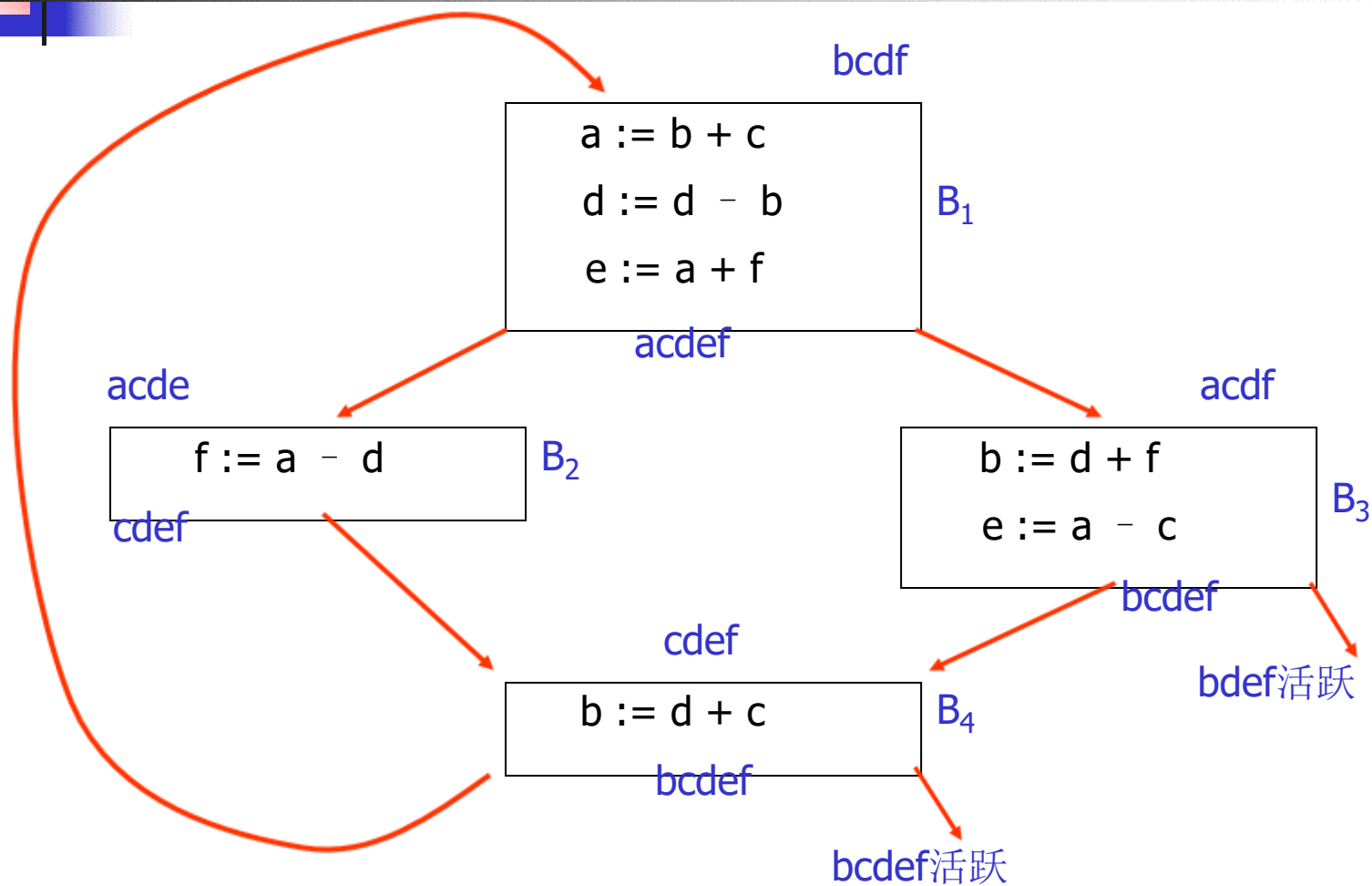


## 节省开销（续）

- 块结束时变量活跃，在块中被定义，后继块中被引用
  - ▣ 9.6节：保存至内存，后继块又需读出到寄存器
  - ▣ 优化：无需保存和读出，节省开销2

$$\sum_{L \text{ 中的块 } B} (\text{use}(x, B) + 2 * \text{live}(x, B))$$

## 例9.6





## 例9.6（续）

- R0, R1, R2

- 对变量a

- 只在B<sub>1</sub>出口活跃,

- $use(a, B_1) = use(a, B_4) = 0,$

- $use(a, B_2) = use(a, B_3) = 1$

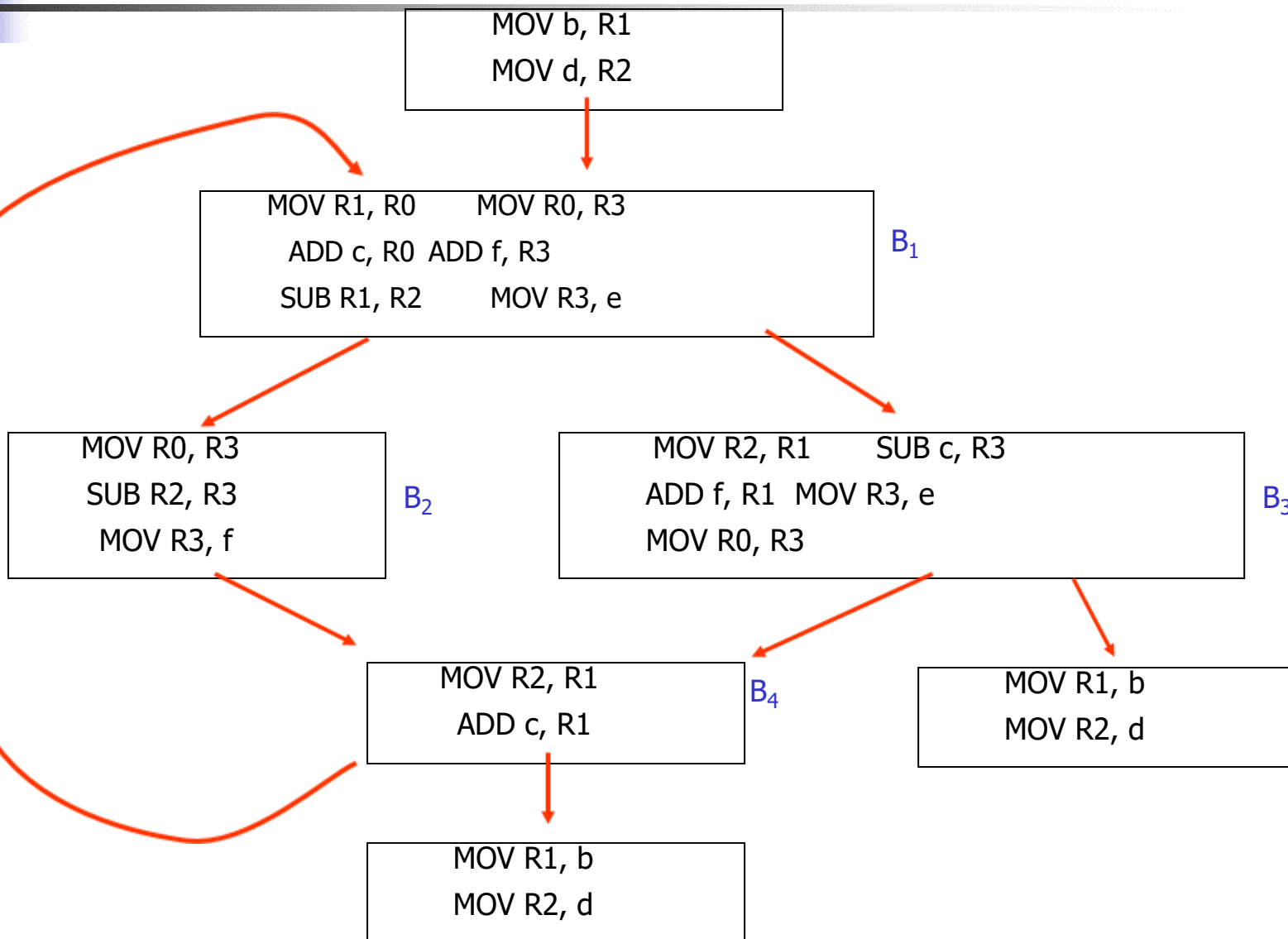
- 总共节省4

- b, c, d, e, f——6, 3, 6, 4, 4

- 可将a、b、d保存在R0、R1、R2中

$$\sum_{L中的B} 2 * live(a, B) = 2$$
$$\sum_{L中的B} use(a, B) = 2$$

## 例9.6（续）





## 9.7.3 外层循环的寄存器分配

- 内层循环相同思想
- $L_1$  包含  $L_2$ ,  $x$  在  $L_2$  中分配了寄存器, 则在  $L_1 - L_2$  不必再分配
- $x$  在  $L_1$  中分配了寄存器, 而  $L_2$  中没有, 则在  $L_2$  入口需保存, 出口需读出
- $x$  在  $L_2$  中分配了寄存器, 而  $L_1$  中没有, 则在  $L_2$  入口需读取, 出口需保存



## 9.7.4 图着色法进行寄存器分配

### ○ 两次扫描

1. 假定寄存器数目是无限的，选择目标机器指令翻译中间代码——每个变量一个寄存器，符号寄存器

2. 分配物理寄存器

#### ➤ 寄存器冲突图，register-interference graph

- 节点——符号寄存器， $R_1$ — $R_2$ ， $R_2$ 定义的位置上 $R_1$ 活跃
- $k$ ——可用物理寄存器数，用 $k$ 个颜色为图着色
- 相邻节点不同颜色——干扰变量使用不同寄存器
- 启发式算法
  - »  $n$ 邻居数 $<k$ ，去掉 $n \rightarrow G'$ ， $G'$ 可 $k$ 着色 $\rightarrow G$ 可 $k$ 着色
  - » 最终——空图（成功）或所有节点邻居数 $\geq n$ （失败）





## 9.8 基本块的DAG表示

- 构造方法

1. 叶：标记变量或常量

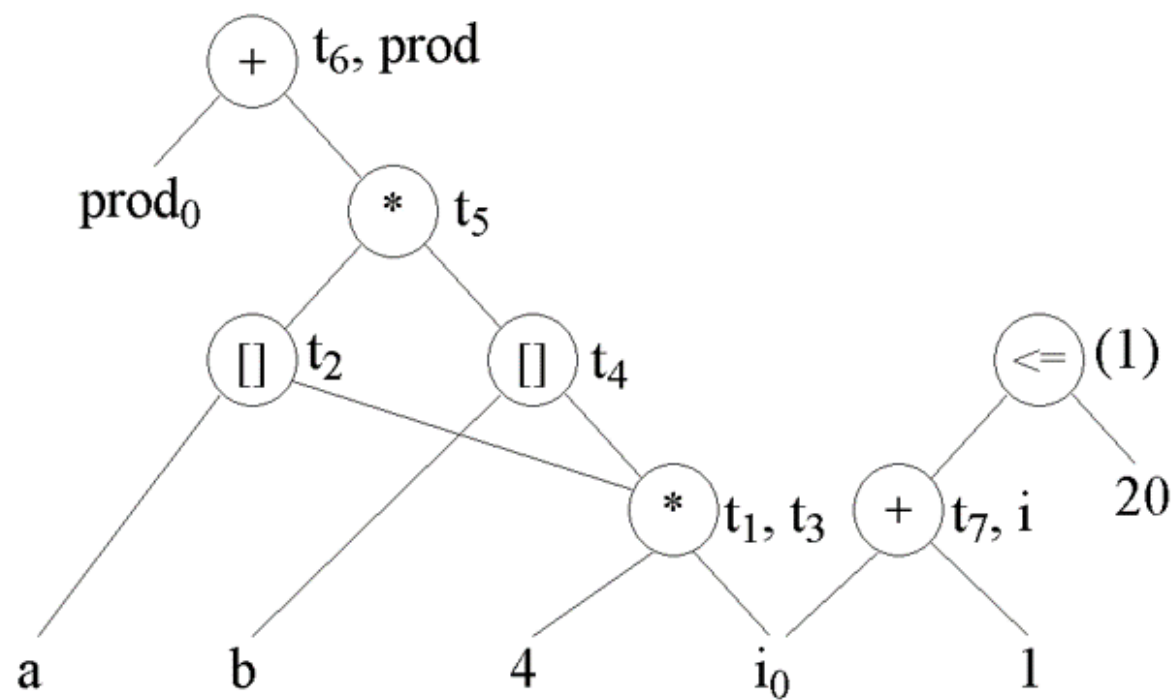
左值/右值

初始值——下标0

2. 内部节点：标记为运算符

3. 为节点标记标识符——计算结果保存在标识符中

1.  $t_1 := 4 * i$
2.  $t_2 := a[t_1]$
3.  $t_3 := 4 * i$
4.  $t_4 := b[t_3]$
5.  $t_5 := t_2 * t_4$
6.  $t_6 := \text{prod} + t_5$
7.  $\text{prod} := t_6$
8.  $t_7 := i + 1$
9.  $i := t_7$
10. if  $i \leq 20$  goto (1)





## 9.8.1 dag的构造

---

算法9.2:

输入 一个基本块

输出 基本块对应的dag，包含如下信息:

1. 每个节点有一个标记，叶节点标记为标识符，  
内部节点标记为操作符
2. 每个节点有一个附着标识符列表



## 算法9.2: dag的构造 (续)

方法:

$\text{node}(\text{identifier})$ : 标识符 $\text{identifier}$ 相关联的节点

对基本块每个语句做步骤(1)-(3)

初始, 没有节点,  $\text{node}$ 函数均为未定义

三种语句: (i)  $x := y \text{ op } z$

(ii)  $x := \text{op } y$

(iii)  $x := y$

### 1. 创建孩子节点

- 若 $\text{node}(y)$ 未定义, 创建一个叶节点, 标记为 $y$ ,  $\text{node}(y)$ 的值设置为此节点。对(i)类语句,  $z$ 同样处理



## 算法9.2: dag的构造 (续)

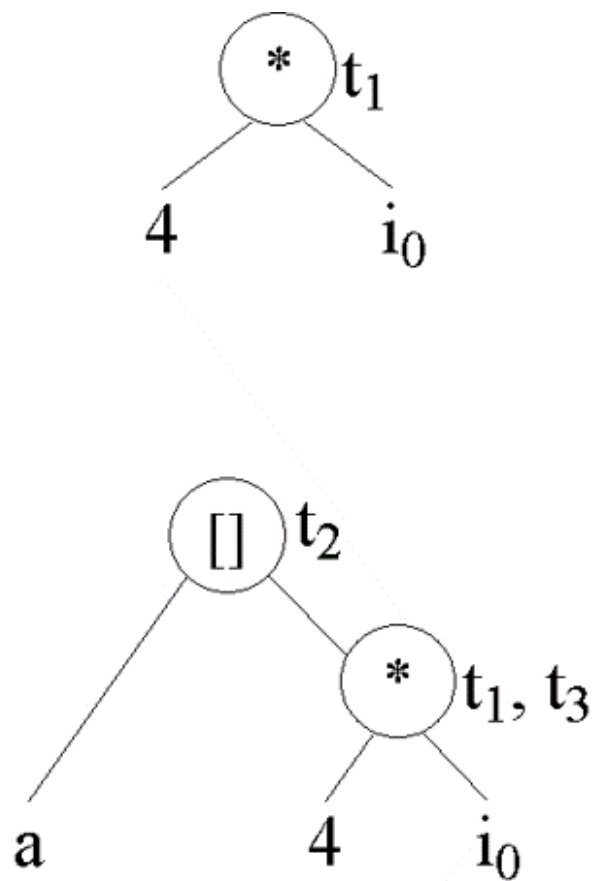
### 1. 创建父节点

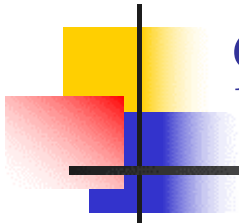
- 对(i)类语句, 检查是否有节点标记为`op`, 且左孩子为`node(y)`, 右孩子为`node(z)` (检查公共子表达式), 若没有, 创建这样的节点。令`n`表示此节点。
- 对(ii)类语句, 检查是否有标记为`op`的节点, 其单一孩子节点为`node(y)`, 若没有, 创建这样的节点。令`n`表示此节点。
- 对(iii)类语句, 令`n=node(y)`

### 2. 附着列表的更新

- 将`x`从节点`node(x)`的附着列表删除, 添加到`n`的附着列表中, 并将`node(x)`的值设置为`n`

## 例9.8





## 9.8.2 dag的应用

1. 公共子表达式的自动检查
  2. 哪些标识符在基本块中被使用
    - 步骤(1), 创建的叶节点对应的标识符
  3. 哪些语句S, 其计算的变量值可能在基本块外被使用
    - S对x赋值, 创建节点n, x附着在n上 (步骤2)
    - dag构造完毕后, 若仍有 $\text{node}(x) == n$ , 则S计算的x值在基本块外可被使用
- 例9.9
- 例9.8中所有所有语句都满足3



## dag的其他应用

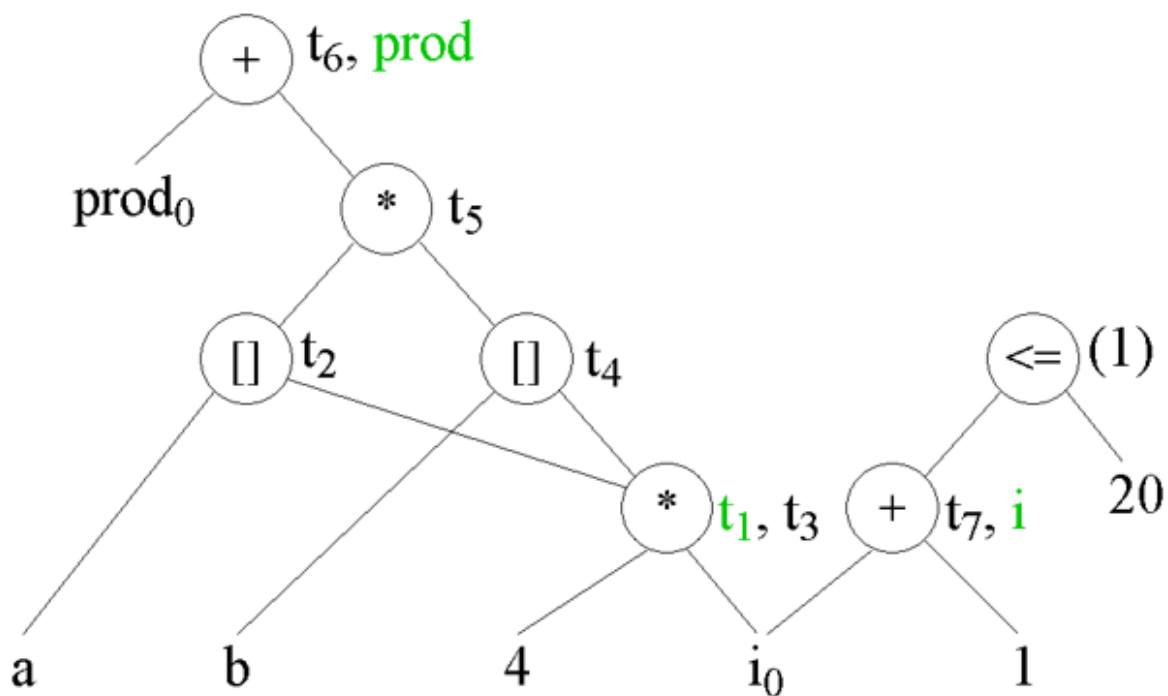
---

- 提取公共子表达式、 $x:=y$ 避免不必要的复制——重新构造优化的三地址码
- 对每个节点关联的标识符，选定一个



## 例9.10

1.  $t_1 := 4 * i$
2.  $t_2 := a[t_1]$
3.  $t_4 := b[t_1]$
4.  $t_5 := t_2 * t_4$
5.  $prod := prod + t_5$
6.  $i := i + 1$
7. if  $i \leq 20$  goto (1)





### 9.8.3 数组、指针和函数调用

○ $x := a[i]$	➔	$x := a[i]$
$a[j] := y$		$z := x$
$z := a[i]$		$a[j] := y$
若 $i = j$ , $y \neq a[i]$ , 错误!		

- 注销 [] 节点, 不附加额外的标识符
- $*p := w$ , 类似, 可能需要注销所有节点
- 函数调用, 注销所有节点
- 额外边——保持原三地址码语句顺序



## 哪些语句顺序应予以保持

1. 前：数组元素的赋值语句  
后：同一数组的元素的计算、赋值语句
2. 前：数组元素的计算语句  
后：同一数组的元素的赋值语句
3. 前：函数调用、利用指针间接赋值  
后：使用任何标识符
4. 前：任何标识符的计算  
后：函数调用、利用指针间接赋值



## 9.9 窥孔（Peephole）优化

- 窥孔：目标代码一个小的、移动的窗口  
➔ 更小、更快的代码
- 1. 消除冗余指令
- 2. 控制流优化
- 3. 代数优化
- 4. 利用机器的特性



## 9.9.1 冗余的Load和Store指令

---

1. MOV R0, a

2. MOV a, R0

- 删除(2)
- 例外：(2)有标号
- 保证(1)、(2)在同一个基本块

## 9.9.2 不可达代码

#define debug 0	→	if debug = 1 goto L1
...		goto L2
if (debug) {		L1: 打印调试信息
打印调试信息 }		L2:
→	if debug <> 1 goto L2	
	打印调试信息	
	L2:	
→	if 0 <> 1 goto L2	} 可删除
	打印调试信息	
	L2:	



## 9.9.3 控制流优化

goto L1



goto L2

...

...

L1: goto L2

L1: goto L2

if a < b goto L1



if a < b goto L2

...

...

L1: goto L2

L1: goto L2

goto L1



if a < b goto L2

...

goto L3

L1: if a < b goto L2

...

L3:

L3:



## 其他优化方法

### ○ 9.9.4 代数优化

- $x := x + 0$ ,  $x := x * 1$

### ○ 9.9.5 强度削弱

- 开销高的指令  $\rightarrow$  等价的开销低的指令

- $x^2$ : 乘方函数  $\rightarrow x * x$

### ○ 9.9.6 利用机器特性

- 特殊机器指令高效实现某些操作

- $i := i + 1 \rightarrow \text{inc } i$



## 9.10 利用dag生成目标代码

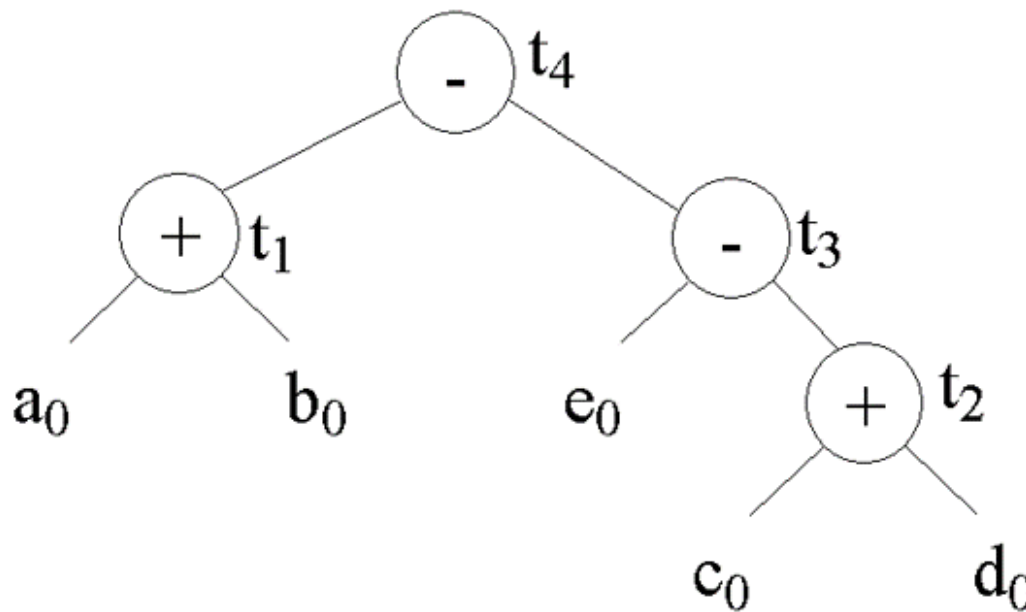
### 9.10.1 重排顺序: $(a + b) - (e - (c + d))$

$t_1 := a + b$

$t_2 := c + d$

$t_3 := e - t_2$

$t_4 := t_1 - t_3$



MOV a, R0

ADD b, R0

MOV c, R1

ADD d, R1

MOV R0, t<sub>1</sub>

MOV e, R0

SUB R1, R0

MOV t<sub>1</sub>, R1

SUB R0, R1

MOV R1, t<sub>4</sub>



## 重排顺序（续）

$t_2 := c + d$

$t_3 := e - t_2$

$t_1 := a + b$

$t_4 := t_1 - t_3$

MOV c, R0

ADD d, R0

MOV e, R1

SUB R0, R1

MOV a, R0

ADD b, R0

SUB R1, R0

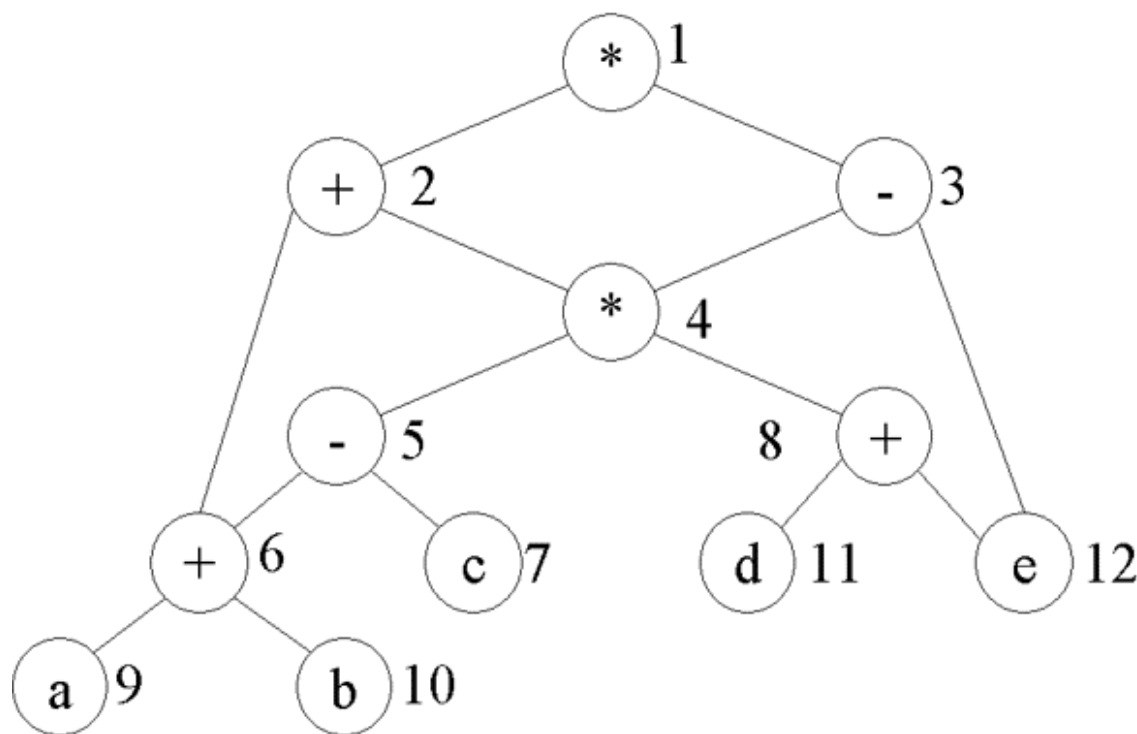
MOV R0,  $t_4$



## 9.10.2 一个启发式重排序算法

- 计算紧接在左孩子之后 ( $t_4$ ,  $t_1$ ) ——共用一个寄存器
- 算法（按逆序给出重排之后的顺序）：
  1. **while** 存在未列出的内部节点 **do begin**
  2. 选择一个未列出的节点 $n$ ，其父节点都已列出;
  3. 列出 $n$ ;
  4. **while**  $n$ 的最左孩子 $m$ 的所有父节点都已列出，且它不是叶节点 **do begin**
  5.     列出 $m$ ;
  6.      $n := m$ ;
  - end**
  - end**

## 例9.11



$t_8 := d + e$

$t_6 := a + b$

$t_5 := t_6 - c$

$t_4 := t_5 * t_8$

$t_3 := t_4 - e$

$t_2 := t_6 + t_4$

$t_1 := t_2 * t_3$



### 9.10.3 树结构的最优排序

---

- dag→树，计算最优顺序的算法
  1. 自底向上为每个节点标记一个整数——不存储中间结果的条件下，计算所需最少寄存器数目
  2. 由标记确定顺序，对树进行遍历，生成目标代码

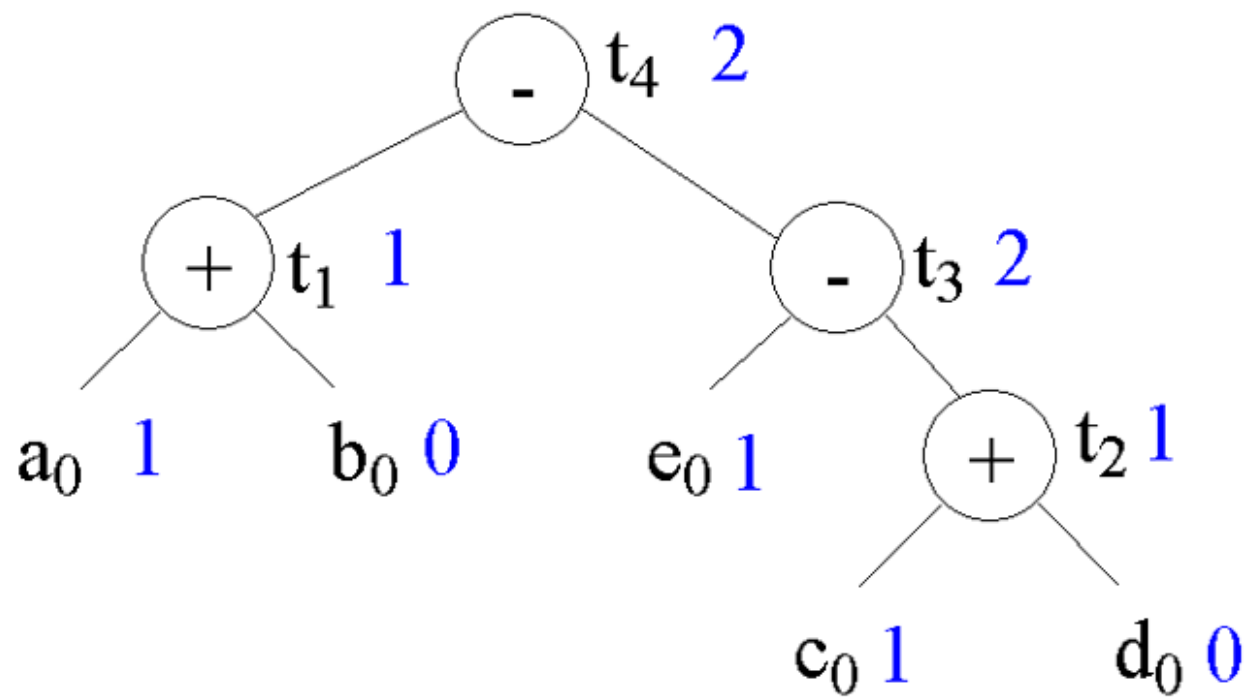


## 9.10.4 标记算法

1. **if**  $n$ 为叶节点 **then**
2.     **if**  $n$ 为其父节点的最左孩子 **then**
3.          $\text{label}(n) := 1$
4.     **else**  $\text{label}(n) := 0$
5.     **else begin**
6.         令 $n_1, n_2, \dots, n_k$ 是 $n$ 的孩子按标号排序后的  
        结果, 即 $\text{label}(n_1) \geq \text{label}(n_2) \geq \dots \geq \text{label}(n_k)$
7.          $\text{label}(n) := \max(\text{label}(n_i) + i - 1), 1 \leq i \leq k$
8.     **end**

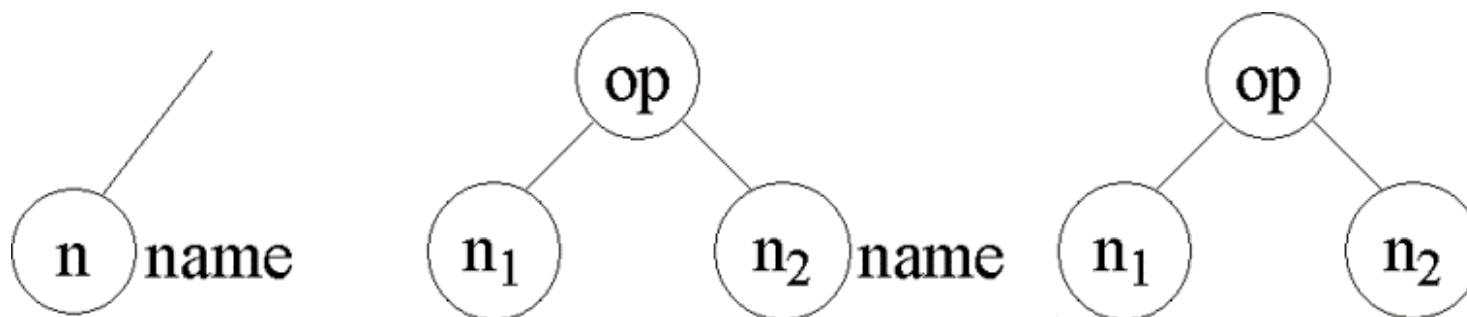
$$\text{label}(n) = \begin{cases} \max(l_1, l_2), & l_1 \neq l_2 \\ l_1 + 1, & l_1 = l_2 \end{cases}$$

## 例9.12

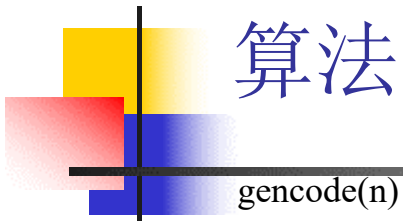


## 9.10.5 利用标记树生成代码

- `gencode(n)`: 生成以 $n$ 为根的标记树 $T$ 的代码, 结果在寄存器 $R0$ 中
- `rstack`: 可用寄存器栈,  $R0, R1, \dots, R(r-1)$
- `swap(rstack)`: 交换栈顶两个寄存器
- `tstack`: 临时存储栈,  $T0, T1, T2, \dots$
- 算法处理的几种情况:







# 算法

gencode(n)

{

if (n为表示运算对象name的左叶节点且为其父节点的最左孩子)

print 'MOV' || name || ',' || top(rstack);

else if (n是内部节点, 运算符为op, 左、右孩子为 $n_1$ 、 $n_2$ )

{

if (label( $n_2$ ) == 0) { /\* 情况1 \*/

令name为 $n_2$ 表示的运算对象;

gencode( $n_1$ );

print op || name || ',' || top(rstack);

} else if (1 ≤ label( $n_1$ ) < label( $n_2$ ) and label( $n_1$ ) < r) { /\* 情况2 \*/

swap(rstack);

gencode( $n_2$ );

R = pop(rstack); /\*  $n_2$ 的值在R中 \*/

gencode( $n_1$ );

print op || R || ',' || top(rstack);

push(rstack, R);

swap(rstack);

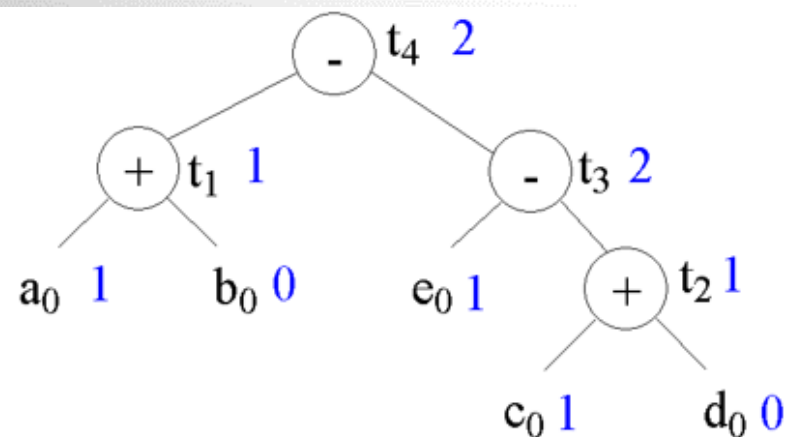


## 算法（续）

```
    } else if (1 <= label(n2) <= label(n1) and label(n2) < r) { /* 情况3 */  
        gencode(n1);  
        R = pop(rstack);          /* n1的值在R中 */  
        gencode(n2);  
        print op || top(rstack) || ',' || R;  
        push(rstack, R);  
    } else { /* 情况4 */  
        gencode(n2);  
        T = pop(tstack);  
        print 'MOV' || top(rstack) || ',' || T;  
        gencode(n1);  
        push(tstack, T);  
        print op || T || ',' || top(rstack);  
    }  
}  
}
```

## 例9.13

gencode(t<sub>4</sub>) [R<sub>1</sub> R<sub>0</sub>] 情况2  
gencode(t<sub>3</sub>) [R<sub>0</sub> R<sub>1</sub>] 情况3  
gencode(e) [R<sub>0</sub> R<sub>1</sub>] 情况0  
    print MOV e, R<sub>1</sub>  
gencode(t<sub>2</sub>) [R<sub>0</sub>] 情况1  
gencode(c) [R<sub>0</sub>] 情况0  
    print MOV c, R<sub>0</sub>  
    print ADD d, R<sub>0</sub>  
    print SUB R<sub>0</sub>, R<sub>1</sub>  
gencode(t<sub>1</sub>) [R<sub>0</sub>] 情况1  
gencode(a) [R<sub>0</sub>] 情况0  
    print MOV a, R<sub>0</sub>  
    print ADD b, R<sub>0</sub>  
print SUB R<sub>1</sub>, R<sub>0</sub>





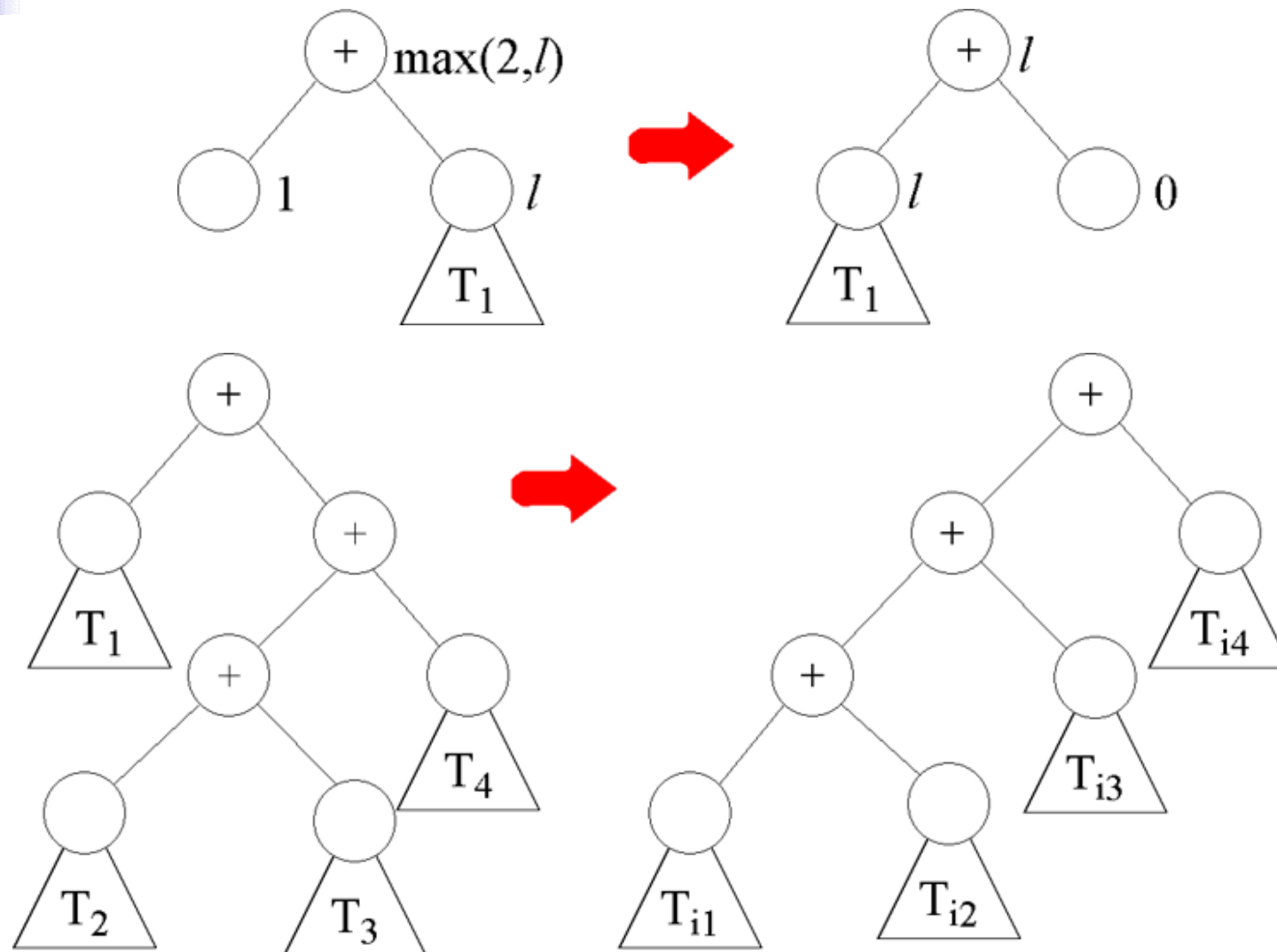
## 9.10.6 多寄存器运算

- 乘、除、函数调用
- 修改标记算法

$$label(n) = \begin{cases} \max(2, l_1, l_2), & l_1 \neq l_2 \\ l_1 + 1, & l_1 = l_2 \end{cases}$$

- “寄存器对”情况
  - 避免使用swap——情况2

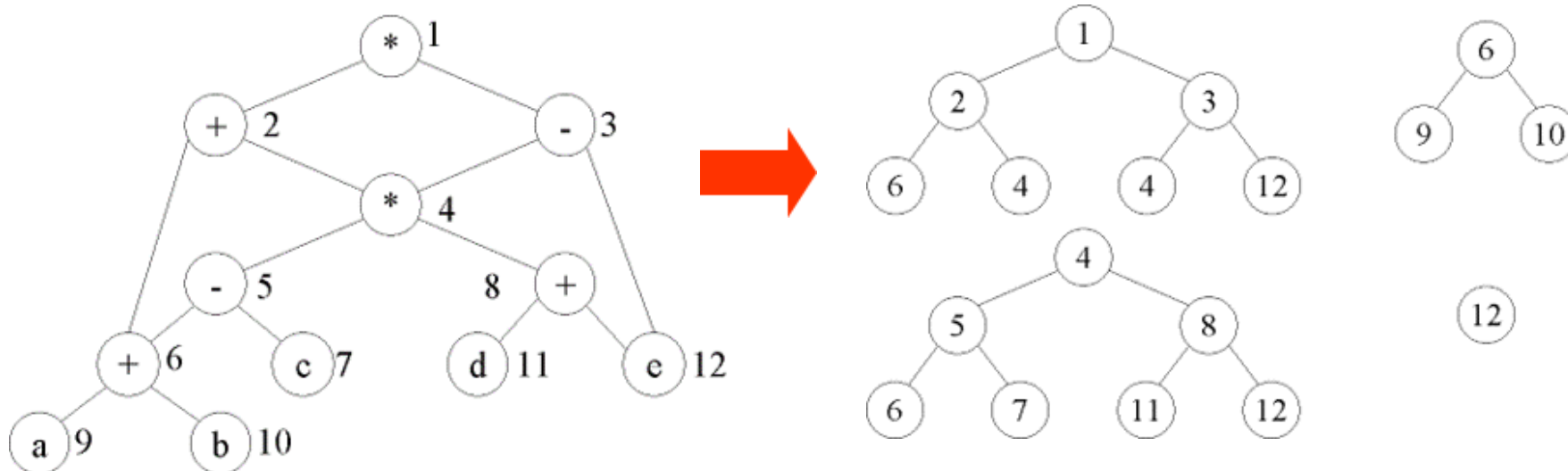
## 9.10.7 利用代数特性优化



## 9.10.8 公共子表达式的处理

○ NP-完全问题

○ dag  $\rightarrow$  森林，只有叶节点可为共享节点



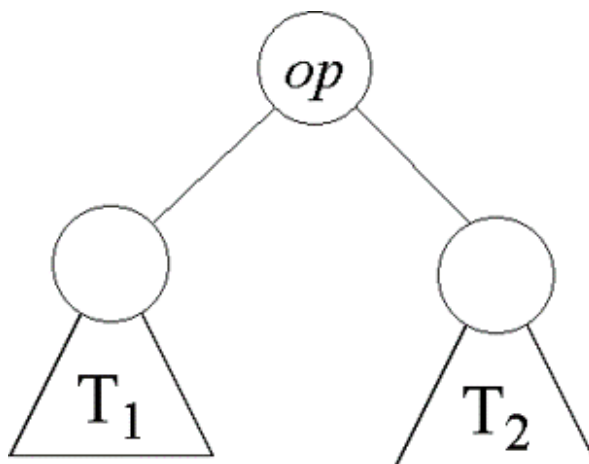


## 9.11 动态规划代码生成算法

- 一条指令使用两个以上寄存器
- $R_i := E$ 
  - $E$ 包含一个或多个寄存器，有一个是 $R_i$
- $\text{ADD } R_0, R_1 \rightarrow R_1 := R_1 + R_0$   
 $\text{ADD } *R_0, R_1 \rightarrow R_1 := R_1 + \text{ind } R_0$
- load指令:  $R_i := M$
- store指令:  $M := R_i$
- 寄存器间拷贝:  $R_i := R_j$

## 9.11.2 动态规划算法原理

- $E=E_1+E_2$ ,  $E$ 的最优程序可由 $E_1$ 、 $E_2$ 的最优程序组合而成
- 连续特性
  - ▣  $T_1$ 、 $T_2$ 的计算是连续的、完整的, 不交叉







## 9.11.4 动态规划算法

1. 对表达式树T的每个节点n，计算以n为根的子树（子表达式）的最优计算开销
  - $C[i]$ : 利用i个寄存器计算表达式的开销
  - $R := E$ 
    - 考虑所有子表达式的不同计算顺序
    - “+” E的指令的计算开销
    - 开销最小者  $\rightarrow C[i]$
2. 遍历T，根据 $C[i]$ 确定哪些子表达式结果必须保存到内存
3. 生成代码



## 例9.14

- 目标机器

- 两个寄存器R0、R1

- 指令集（开销均为1）：

$R_i := M_j$

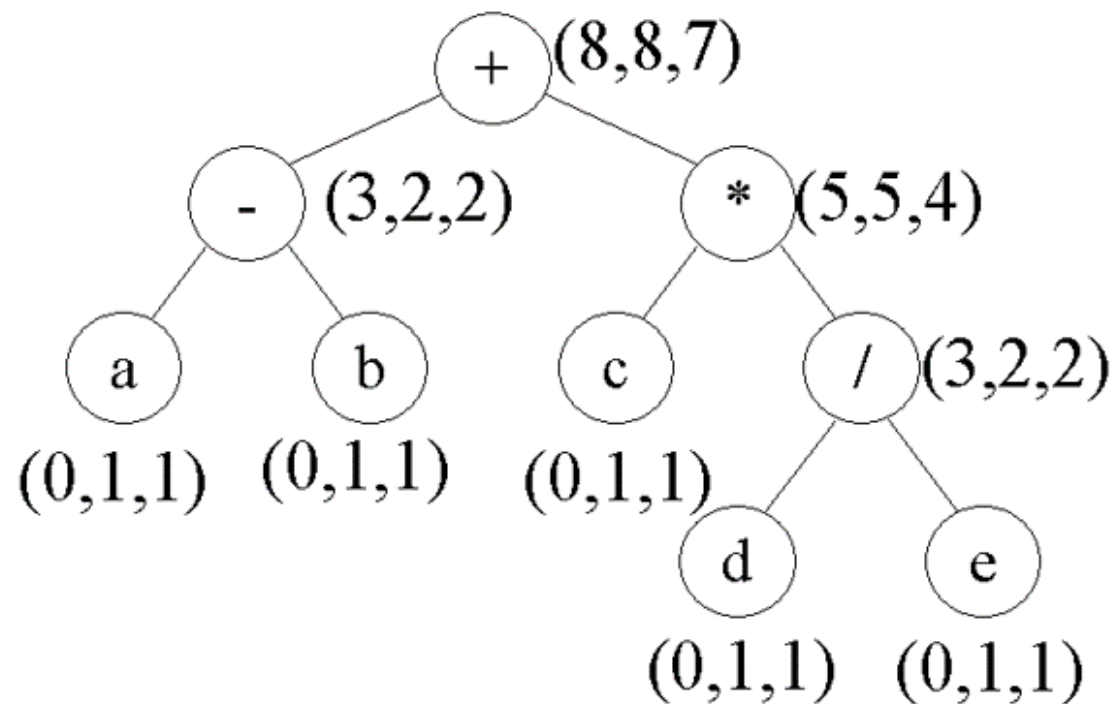
$R_i := R_i \text{ op } R_j$

$R_i := R_i \text{ op } M_j$

$R_i := R_j$

$M_i := R_j$

## 例9.14（续）



### ○ 叶节点a

- 结果→内存，C[0]=0（已经在内存中）
- 结果→寄存器，一个可用寄存器，C[1]=1
- C[2]=1



## 例9.14（续）

### ○ 根节点

#### □ 使用一个寄存器—— $R0 := R0 + M$

- 最小开销：右子树→内存，左子树→寄存器  
 $5+2+1=8$

#### □ 使用两个寄存器

- 左子树——使用两个寄存器，右子树——一个，  
开销 $2+5+1=8$
- 左子树——1，右子树——2， $2+4+1=7$
- 左子树——2，右子树——内存， $2+5+1=8$



## 例9.14（续）

---

○ 最终结果：

$R0 := c$

$R1 := d$

$R1 := R1 / e$

$R0 := R0 * R1$

$R1 := a$

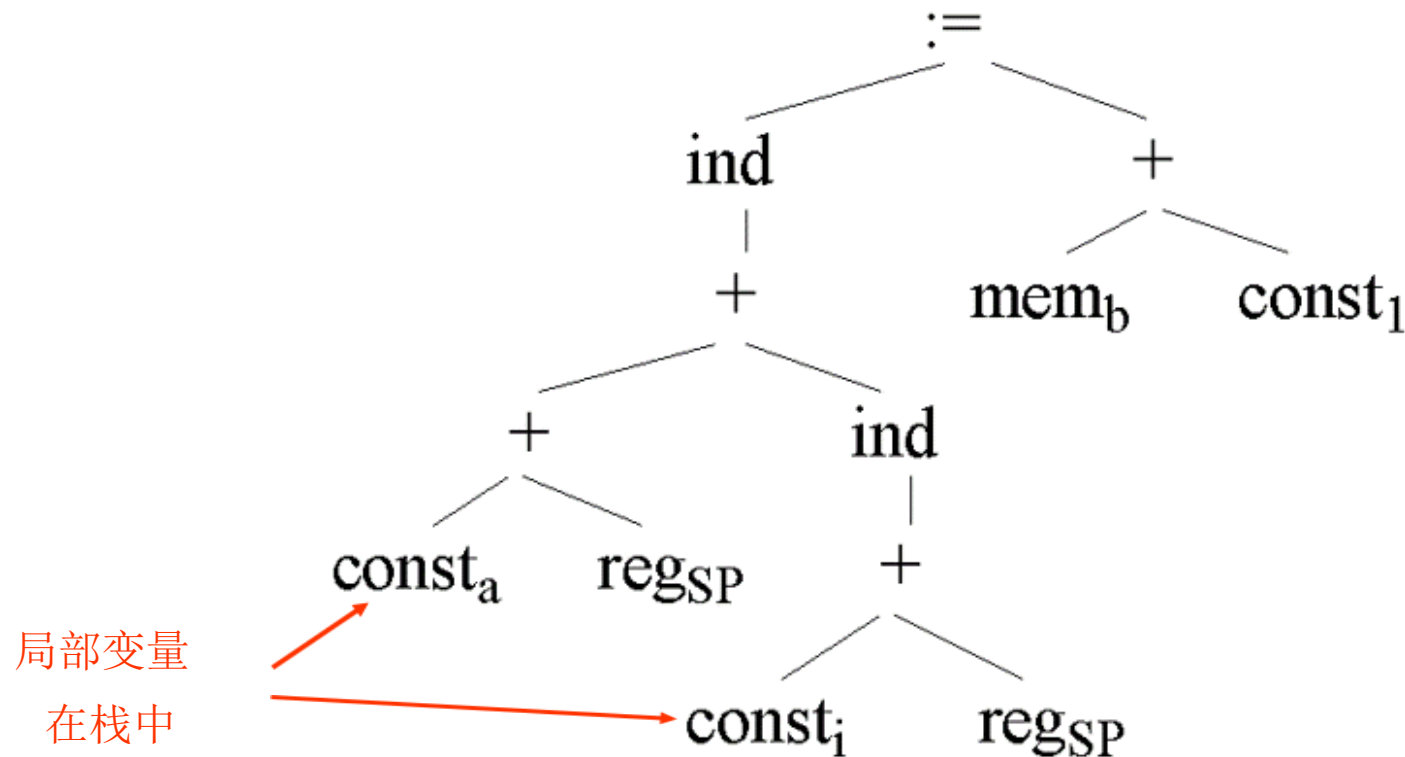
$R1 := R1 - b$

$R1 := R1 + R0$

## 9.12 代码生成程序自动生成器

### 9.12.1 tree rewriting

例9.15:  $a[i] := b + 1$  对应的输入树



# 代码生成算法

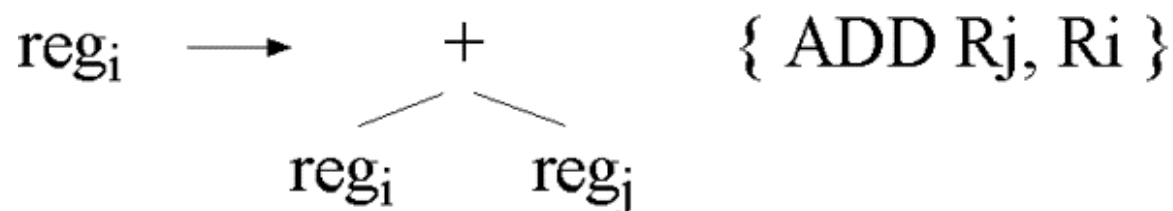
- 改写规则——树  $\rightarrow$  单个节点

$\text{replace} \leftarrow \text{template} \{ \text{action} \}$

单个节点          树          语义动作

- 一组改写规则——树翻译模式

- 模板——机器指令，动作——翻译序列



- 输入树中子树与模板匹配  $\rightarrow$   
将其替换为  $\text{reg}_i$ ，输出  $\text{ADD } R_j, R_i$

## 例9.16（改写规则例）

$\text{reg}_i \longrightarrow \text{const}_c \quad \{ \text{MOV } \#c, \text{Ri} \}$

$\text{reg}_i \longrightarrow \text{mem}_a \quad \{ \text{MOV } a, \text{Ri} \}$

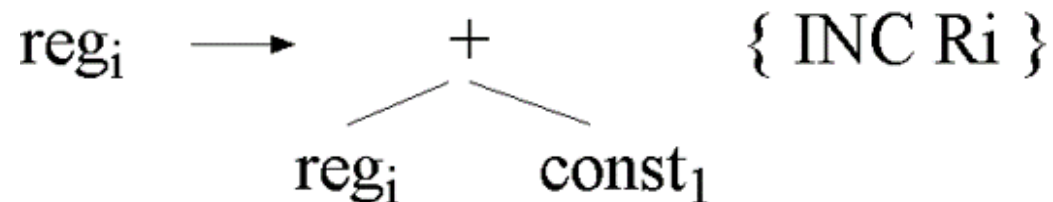
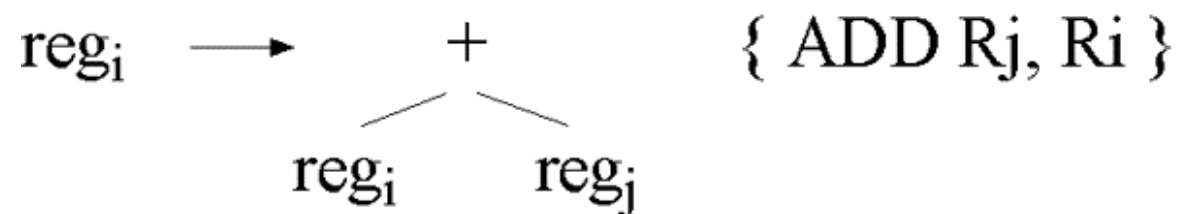
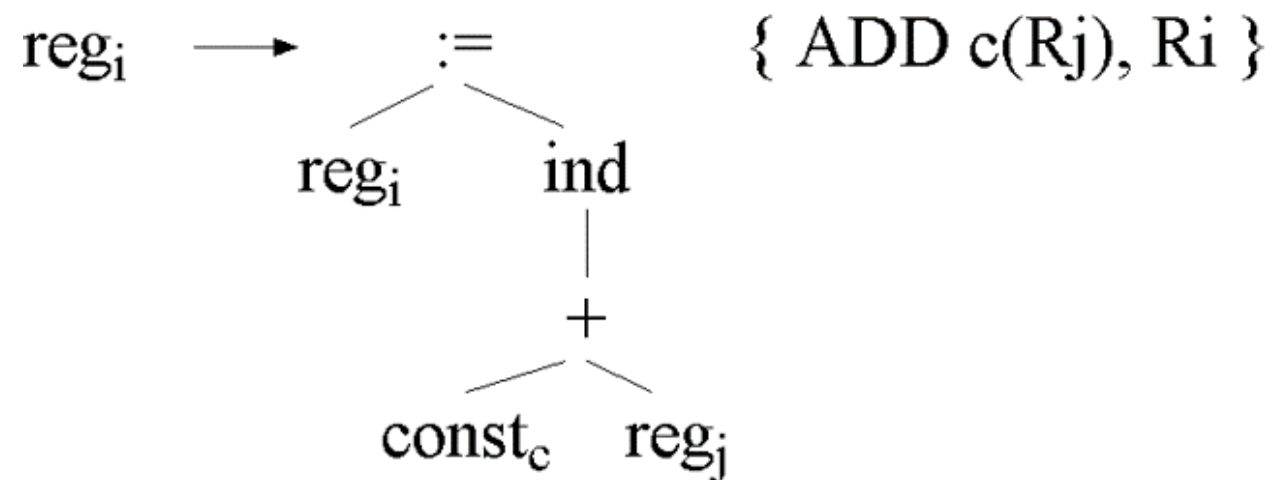
$\text{mem} \longrightarrow \begin{array}{c} := \\ \swarrow \quad \searrow \\ \text{mem}_a \quad \text{reg}_i \end{array} \quad \{ \text{MOV } \text{Ri}, a \}$

$\text{mem} \longrightarrow \begin{array}{c} := \\ \swarrow \quad \searrow \\ \text{ind} \quad \text{reg}_j \\ | \\ \text{reg}_i \end{array} \quad \{ \text{MOV } \text{Rj}, * \text{Ri} \}$

$\text{reg}_i \longrightarrow \begin{array}{c} \text{ind} \\ | \\ + \\ \swarrow \quad \searrow \\ \text{const}_c \quad \text{reg}_j \end{array} \quad \{ \text{MOV } c(\text{Rj}), \text{Ri} \}$

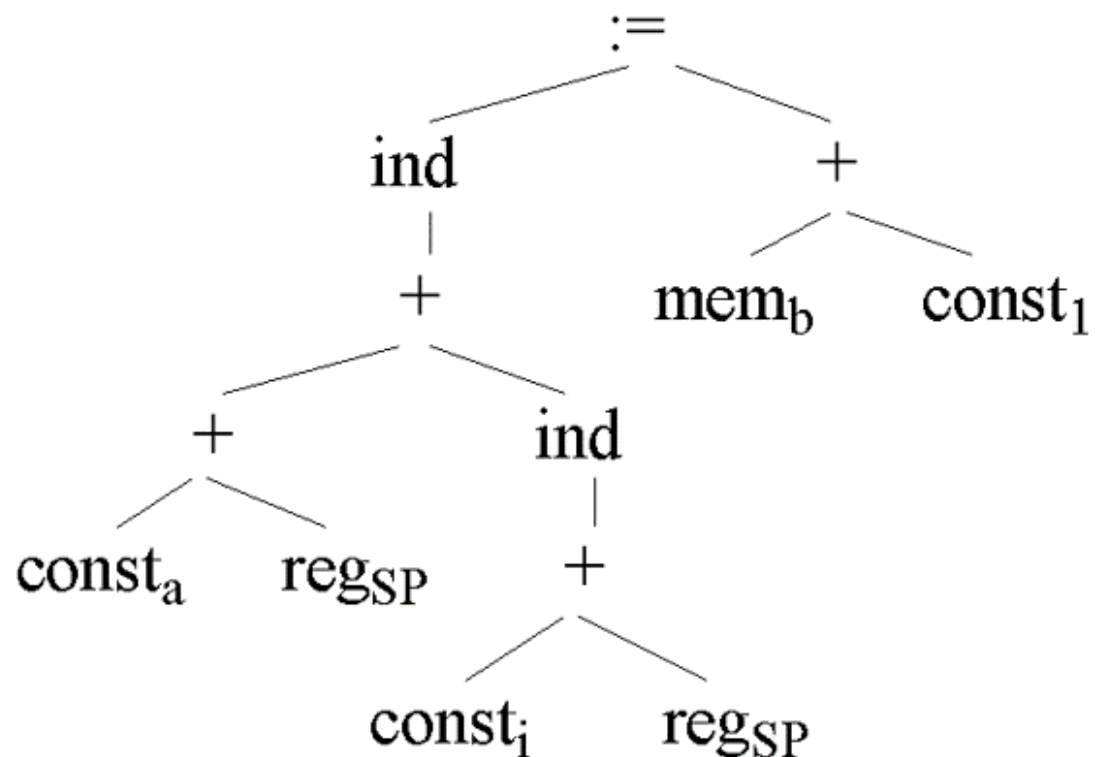


## 例9.16 (续)



## 例9.17 由改写规则进行翻译

○ 输入

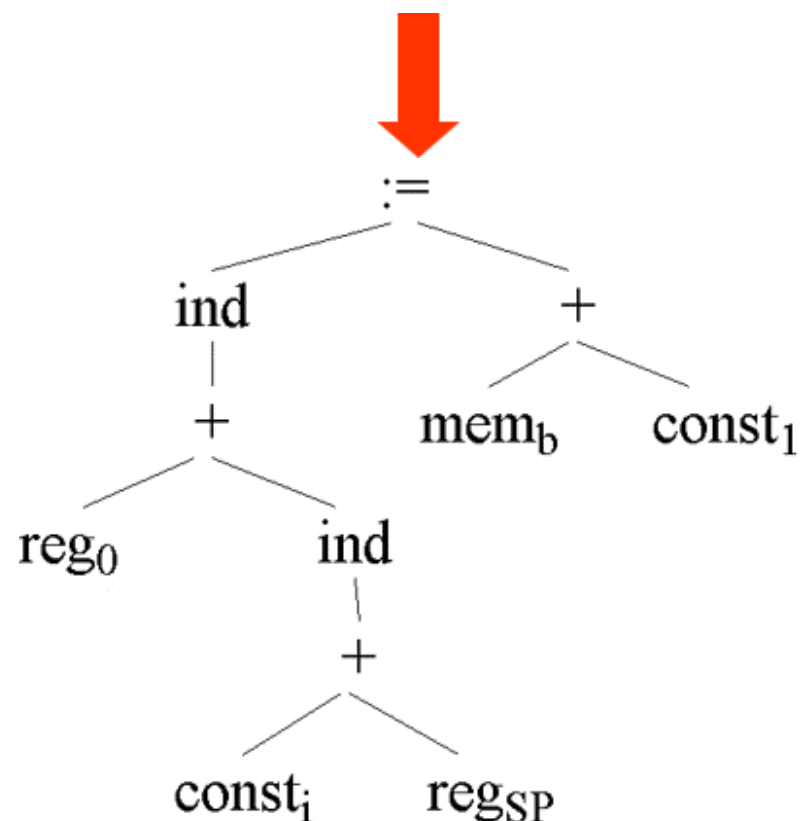


## 例9.17（续）

(1)  $\text{reg}_0 \rightarrow \text{const}_a$     { MOV #a, R0 }

(7)  $\text{reg}_0 \rightarrow +$     { ADD SP, R0 }

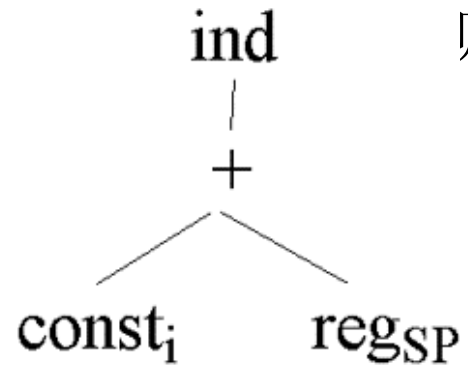
$\text{reg}_0$      $\text{reg}_{\text{SP}}$



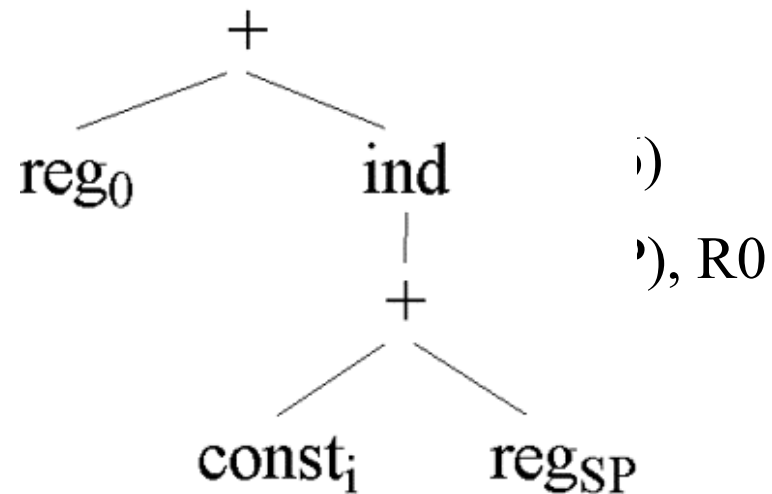
## 例9.17（续）

可对子树

则(5)



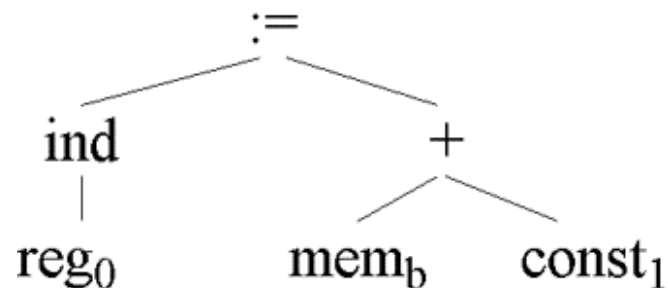
也可对子树



)  
) , R0

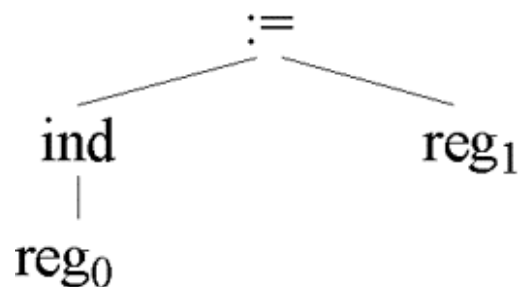


## 例9.17（续）



对 $\text{mem}_b$ 应用规则(2): MOV b, R1

再应用规则(8): INC R1



最后应用规则(4): MOV R1, \*R0



## 例9.17（续）

---

MOV #a, R0

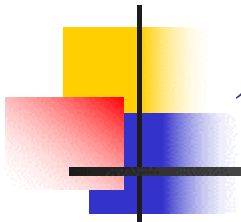
ADD SP, R0

ADD i(SP), R0

MOV b, R1

INC R1

MOV R1, \*R0



## 几个问题

- 如何匹配？——习题3.32，用字符串表示模板
- 匹配顺序和多匹配问题——结合动态规划算法
- 实现方法
  - ▣ 深度优先遍历输入树，同时进行匹配
  - ▣ 可利用动态规划算法获取最优匹配



## 9.12.2 利用语法分析的模式匹配

### ○ 输入树的前缀表示法

$:= \text{ind} + + \text{const}_a \text{reg}_{\text{SP}} \text{ind} + \text{const}_i \text{reg}_{\text{SP}} + \text{mem}_b \text{const}_1$

### ○ 对它进行语法分析，利用翻译模式进行

翻译

1.  $\text{reg}_i \rightarrow \text{const}_c \quad \{ \text{MOV } \#c, \text{Ri} \}$
2.  $\text{reg}_i \rightarrow \text{mem}_a \quad \{ \text{MOV } a, \text{Ri} \}$
3.  $\text{mem} \rightarrow := \text{mem}_a \text{reg}_i \quad \{ \text{MOV } \text{Ri}, a \}$
4.  $\text{mem} \rightarrow := \text{ind reg}_i \text{reg}_j \quad \{ \text{MOV } \text{Rj}, * \text{Ri} \}$
5.  $\text{reg}_i \rightarrow \text{ind} + \text{const}_c \text{reg}_j \quad \{ \text{MOV } c(\text{Rj}), \text{Ri} \}$
6.  $\text{reg}_i \rightarrow + \text{reg}_i \text{ind} + \text{const}_c \text{reg}_j \quad \{ \text{ADD } c(\text{Rj}), \text{Ri} \}$
7.  $\text{reg}_i \rightarrow + \text{reg}_i \text{reg}_j \quad \{ \text{ADD } \text{Rj}, \text{Ri} \}$
8.  $\text{reg}_i \rightarrow + \text{reg}_i \text{const}_1 \quad \{ \text{INC } \text{Ri} \}$





# 语法分析方法进行代码生成

## ○ 二义性

- 规约—规约冲突：选择较长的规约
- 移进—规约冲突：移进

## ○ 优点

- 实现可靠、高效
- 移植方便
- 可方便实现利用机器特性的高效代码生成

## ○ 缺点

- 语法分析限制了计算顺序：左→右
- 文法可能过于庞大
- 避免无法分析和无限循环问题



## 9.12.3 语义检查

- 输入树叶节点：类型+下标
- 机器指令对下标可能有限制
- 在语义动作中添加检查部分

