



编译原理

第九章 代码生成



哈尔滨工业大学 陈鄞 单丽莉



提纲

9.1 代码生成器的主要任务

9.2 一个简单的目标机模型

9.3 指令选择

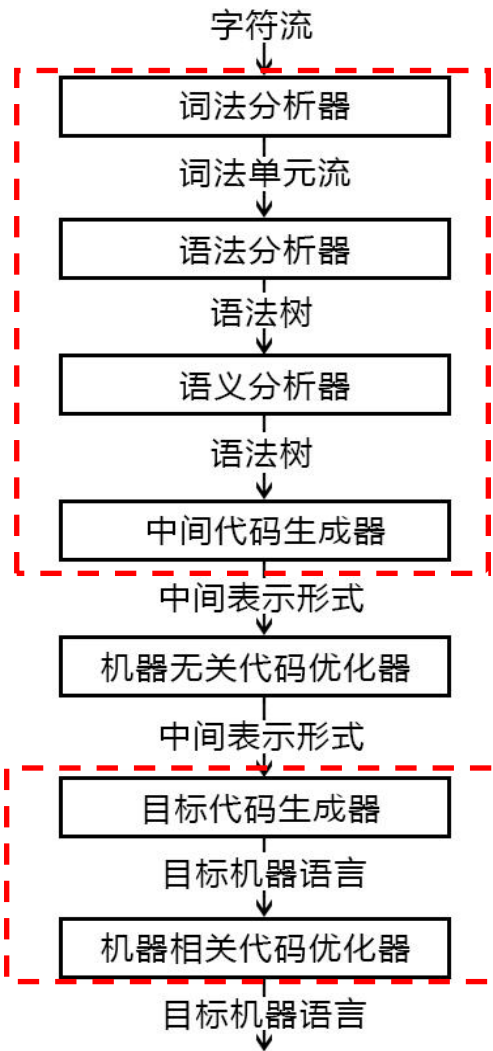
9.4 寄存器选择

9.5 窥孔优化

编译器的结构

分析部分/
前端(front end):
与源语言相关

综合部分/
后端(back end):
与目标语言相关



中间表示IR
符号表
数据流信息

目标代码生成器

目标代码

代码生成器的主要任务

➤ 指令选择

➤ 选择适当的 **目标机指令或指令序列** 来实现 **中间表示(IR)语句**

➤ 例：

➤ 三地址语句

➤ $x = y + z$

➤ 目标代码

➤ **LD** **R0**, **y** /* 把y的值**加载**到寄存器**R0**中 */

➤ **ADD** **R0**, **R0**, **z** /* **z****加**到**R0**上 */

➤ **ST** **x**, **R0** /* 把**R0**的值**保存**到**x**中 */

代码生成器的主要任务

➤ 指令选择

➤ 选择适当的 **目标机指令或指令序列** 来实现 **中间表示(IR)语句**

➤ 例：

三地址语句序列

➤ $a = b + c$

➤ $d = a + e$

目标代码

➤ *LD R0, b* // $R0 = b$

➤ *ADD R0, R0, c* // $R0 = R0 + c$

➤ *ST a, R0* // $a = R0$

➤ *LD R0, a* // $R0 = a$

➤ *ADD R0, R0, e* // $R0 = R0 + e$

➤ *ST d, R0* // $d = R0$

代码生成器的主要任务

➤ 指令选择

- 选择适当的 **目标机指令或指令序列** 来实现 **中间表示(IR)语句**

➤ 寄存器分配和指派

- 把哪个值放在哪个寄存器中

➤ 指令排序

- 按照什么顺序来安排指令的执行



提纲

9.1 代码生成器的主要任务

9.2 一个简单的目标机模型

9.3 指令选择

9.4 寄存器选择

9.5 窥孔优化

一个简单的目标机模型

➤ 三地址机器模型

➤ 加载、保存、运算、跳转等操作

➤ 内存按字节寻址

➤ n 个通用寄存器 $R0, R1, \dots, Rn-1$

➤ 假设所有的运算分量都是整数

➤ 指令之间可能有一个标号

目标机器的主要指令

- 指令格式：运算符 目标地址, 运算分量1, 运算分量2
- 加载指令 *LD dst, addr*
 - *LD r, x*
 - *LD r₁, r₂*
- 保存指令 *ST x, r*
- 运算指令 *OP dst, src1, src2* //假设必须是寄存器
- 无条件跳转指令 *BR L*
- 条件跳转指令 *Bcond r, L*
 - 例: *BLTZ r, L* // if $r < 0$ jump to L

寻址模式

➤ 变量名 x

➤ 例: $LD\ R1, \underline{x}$

➤ $R1 = contents(\underline{x})$

寻址模式

➤ 变量名 x

➤ $a(r)$

➤ a 是一个变量（内存位置）， r 是一个寄存器作为下标

➤ $a(r)$ 表示一个内存地址： a + 下标寄存器 r 中的值

➤ 这种寻址方式对于数组访问是很有用的

➤ 数组的基地址为 a ，某元素基于数组基地址的偏移地址为 r

➤ 例： $LD\ R1, \underline{a(R2)}$

➤ $R1 = \underline{contents(a + contents(R2))}$

寻址模式

➤ 变量名 x

➤ $a(r)$

➤ $c(r)$

➤ c 是一个整数， r 是下标寄存器，存放一个内存地址

➤ $c(r)$ 表示一个内存地址：寄存器 r 中的值 + 整数 c

➤ 这种寻址方式可以用于沿指针取值的寻址， r 的值即指针值， c 表示与指针值的偏移

➤ 例：LD $R1$, 100 (R2)

➤ $R1 = \text{contents}(\text{contents}(R2) + 100)$

寻址模式

➤ 变量名 x

➤ $a(r)$

➤ $c(r)$

➤ $*r$

➤ 间接寻址

➤ 寄存器 r 存放的是一个内存地址 A ，内存地址 A 的位置存放了另一个内存地址 B ， $*r$ 即表示地址 B

➤ 例： $LD\ R1, \underline{*R2}$

➤ $R1 = \text{contents}(\text{contents}(\text{contents}(\underline{R2})))$

地址 A

地址 B

寻址模式

➤ 变量名 x

➤ $a(r)$

➤ $c(r)$

➤ $*r$

➤ $*c(r)$

➤ c 是一个整数，在寄存器 r 中存放一个地址 A ， $A+c$ 的内存位置上存放另一个地址 B ， $*c(r)$ 即表示地址 B

➤ 例： $LD\ R1, \underline{*100(R2)}$

➤ $R1 = \text{contents}(\underline{\text{contents}(\text{contents}(R2) + 100)})$

寻址模式

➤ 变量名 x

➤ $a(r)$

➤ $c(r)$

➤ $*r$

➤ $*c(r)$

➤ $\#c$

➤ 立即数 c

➤ 例: $LD\ R1, \#100$

➤ $R1 = 100$

A 3D metallic staircase structure, composed of several rectangular blocks, is positioned on the left side of the slide. The background is a light blue surface covered with a pattern of binary code (0s and 1s) in a darker blue color. The staircase has a modern, industrial look with a brushed metal texture.

提纲

9.1 代码生成器的主要任务

9.2 一个简单的目标机模型

9.3 指令选择

9.4 寄存器选择

9.5 窥孔优化

运算语句的目标代码

➤ 三地址语句

➤ $x = y - z$

➤ 目标代码

➤ *LD R1, y* // $R1 = y$

➤ *LD R2, z* // $R2 = z$

➤ *SUB R1, R1, R2* // $R1 = R1 - R2$

➤ *ST x, R1* // $x = R1$

尽可能避免使用上面的全部四个指令，如果

✓ 所需的运算分量已经在寄存器中了，就无需加载它们

✓ 运算结果被存于寄存器中使用，可能不需要存放回内存

数组寻址语句的目标代码

➤ 三地址语句

➤ $b = a[i]$

➤ a 是一个实数数组，每个实数占8个字节

➤ 目标代码

➤ ***LD R1 , i // R1 = i***

➤ ***MUL R1 , R1, 8 // R1=R1 * 8***

➤ ***LD R2 , a(R1) // R2=contents (a + contents(R1))***

➤ ***ST b , R2 // b = R2***

数组寻址语句的目标代码

➤ 三地址语句

➤ $a[j] = c$

➤ a 是一个实数数组，每个实数占8个字节

➤ 目标代码

➤ $LD \quad R1, \quad c \qquad \qquad // \quad R1 = c$

➤ $LD \quad R2, \quad j \qquad \qquad // \quad R2 = j$

➤ $MUL \quad R2, \quad R2, 8 \qquad // \quad R2 = R2 * 8$

➤ $ST \quad a(R2), R1 \qquad // \quad contents(a+contents(R2))=R1$

指针存取语句的目标代码

➤ 三地址语句

➤ $x = *p$

➤ 目标代码

➤ $LD\ R1, p$ $//\ R1 = p$

➤ $LD\ R2, 0(R1)$ $//\ R2 = contents\ (0 + contents\ (R1))$

➤ $ST\ x, R2$ $//\ x = R2$

指针存取语句的目标代码

➤ 三地址语句

➤ $*p = y$

➤ 目标代码

➤ *LD R1, p* // $R1 = p$

➤ *LD R2, y* // $R2 = y$

➤ *ST 0(R1), R2* // $\text{contents}(0 + \text{contents}(R1)) = R2$

条件跳转语句的目标代码

➤ 三地址语句

➤ *if $x < y$ goto L*

➤ 目标代码

➤ *LD $R1, x$ // $R1 = x$*

➤ *LD $R2, y$ // $R2 = y$*

➤ *SUB $R1, R1, R2$ // $R1 = R1 - R2$*

➤ *BLTZ $R1, M$ // if $R1 < 0$ jump to M*

*M 是标号为 L 的三地址指令所产生的
目标代码中的第一个指令的标号*

过程调用和返回的目标代码

- 使用静态内存分配方式
- 使用栈式内存分配方式

使用静态内存分配

调用者

➤ 三地址语句

➤ *call callee*

➤ 目标代码

➤ *ST callee.staticArea, #here + 20*

➤ *BR callee.codeArea*

*callee*的活动记录在静态区中的起始地址

*callee*的目标代码在代码区中的起始地址

任务：保存返回地址，跳转到被调用者代码

被调用者

➤ 三地址语句

➤ *return*

➤ 目标代码

➤ *BR *callee.staticArea*

任务：跳转回调用者代码

使用静态内存分配

调用者

- *ST callee.staticArea, #here + 20*
- *BR callee.codeArea*

被调用者

- *BR *callee.staticArea*

目标代码

例： 有下面的三地址代码
：

//调用者 *c*

action₁

call p

action₂

halt

//被调用者 *p*

action₃

return

假定

*c*和*p*的代码从地址100和200开始，
*p*的活动记录起始地址为364

ACTION指令占20个字节

100: ACTION1
120: ST 364, #140
132: BR 200
140: ACTION2
160: HALT
.....
200: ACTION3
220: BR *364
.....
364: 140
.....

调用者 *c*

被调用者 *p*

使用栈式内存分配方式

➤ 三地址语句

➤ *call callee*

➤ 目标代码: 调用序列

➤ *ADD SP, SP, #caller.recordsize*

➤ *ST 0(SP), #here + 16*

➤ *BR callee.codeArea*

➤ 三地址语句

➤ *return*

➤ 目标代码: 返回序列

➤ 被调用过程

➤ *BR *0(SP)*

➤ 调用过程

➤ *SUB SP, SP, #caller.recordsize*

新增任务: 通过维护栈顶指针, 分配和回收活动记录空间

使用栈式内存分配方式

- 目标代码：调用序列
 - *ADD SP, SP, #caller.recordsize*
 - *ST 0(SP), #here + 16*
 - *BR callee.codeArea*

//调用者*m*

action₁

call q

action₂

halt

//被调用者*q*

action₃

return

假定

*m*和*q*的代码从地址100和300开始，
*m*的活动记录大小为常数*msize*，

栈区地址从600开始，
ACTION指令占20个字节

- 目标代码：返回序列
 - 被调用过程
 - *BR *0(SP)*
 - 调用过程
 - *SUB SP, SP, #caller.recordsize*

目标代码

```
100: LD SP #600
108: ACTION1
128: ADD SP, SP, #msize
136: ST 0(SP), #152
144: BR 300
152: SUB SP, SP, #msize
160: ACTION2
180: HALT
.....
300: ACTION3
220: BR *0(SP)
.....
```

调用者
m

被调用者
q



提纲

9.1 代码生成器的主要任务

9.2 一个简单的目标机模型

9.3 指令选择

9.4 寄存器选择

9.5 窥孔优化

三地址语句的目标代码生成

- 对每个形如 $x = y \text{ op } z$ 的三地址指令 I ，执行如下动作
 - 调用函数 `getreg(I)` 来为 x 、 y 、 z 选择寄存器，把这些寄存器称为 R_x 、 R_y 、 R_z
 - 如果 R_y 中存放的不是 y ，则生成指令 “ $LD R_y, y'$ ”。 y' 是存放 y 的内存位置之一
 - 类似的，如果 R_z 中存放的不是 z ，生成指令 “ $LD R_z, z'$ ”
 - 生成目标指令 “ $OP R_x, R_y, R_z$ ”

基本块的收尾处理

- 对于一个在基本块的出口处可能活跃^{活跃}的变量 x ，如果它的地址描述符表明它的值没有存放在 x 的内存位置上，则生成指令“ $ST\ x, R$ ”（ R 是在基本块结尾处存放 x 值的寄存器）

寄存器描述符和地址描述符

➤ 寄存器描述符 (*register descriptor*)

- 记录每个寄存器当前存放的是哪些变量的值

➤ 地址描述符 (*address descriptor*)

- 记录运行时每个名字的当前值存放在哪个或哪些位置

- 该位置可能是寄存器、栈单元、内存地址或者是它们的某个集合

- 这些信息可以存放在该变量名对应的符号表条目中

管理寄存器和地址描述符

- 当代码生成算法生成加载、保存和其他指令时，它必须同时更新寄存器和地址描述符
- 对于指令 “ $LD\ R, x$ ”
 - 修改 R 的寄存器描述符，使之只包含 x
 - 修改 x 的地址描述符，把 R 作为新增位置加入到 x 的位置集合中
 - 从任何不同于 x 的地址描述符中删除 R

管理寄存器和地址描述符

- 当代码生成算法生成加载、保存和其他指令时，它必须同时更新寄存器和地址描述符
- 对于指令 “ $LD\ R, x$ ”
- 对于指令 “ $OP\ R_x, R_y, R_z$ ”
 - 修改 R_x 的寄存器描述符，使之只包含 x
 - 从任何不同于 R_x 的寄存器描述符中删除 x
 - 修改 x 的地址描述符，使之只包含位置 R_x
 - 从任何不同于 x 的地址描述符中删除 R_x

管理寄存器和地址描述符

- 当代码生成算法生成加载、保存和其他指令时，它必须同时更新寄存器和地址描述符
 - 对于指令 “ $LD\ R, x$ ”
 - 对于指令 “ $OP\ R_x, R_y, R_z$ ”
 - 对于指令 “ $ST\ x, R$ ”
 - 修改 x 的地址描述符，使之包含自己的内存位置

管理寄存器和地址描述符

- 当代码生成算法生成**加载**、**保存**和其他指令时，它必须同时更新寄存器和地址描述符
 - 对于指令 “ $LD\ R, x$ ”
 - 对于指令 “ $OP\ R_x, R_y, R_z$ ”
 - 对于指令 “ $ST\ x, R$ ”
 - 对于复制语句 $x=y$ ，如果需要生成加载指令 “ $LD\ R_y, y'$ ” 则
 - 修改 R_y 的寄存器描述符，使之只包含 y
 - 修改 y 的地址描述符，把 R_y 作为新增位置加入到 y 的位置集合中
 - 从任何不同于 y 的变量的地址描述符中删除 R_y
 - 修改 R_y 的寄存器描述符，使之也包含 x
 - 修改 x 的地址描述符，使之只包含 R_y
 - 在任何不同于 R_y 的寄存器描述符中删除 x
- 如果 y 已经在 R_y 中
只做这三步

例

$$t = a - b$$

$$u = a - c$$

$$v = t + u$$

$$a = d$$

$$d = v + u$$

LD R1 , a
LD R2 , b
SUB R2 , R1 , R2

假设寄存器分配:

a, R1

b, R2

t, R2

<i>R1</i>	<i>R2</i>	<i>R3</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>t</i>	<i>u</i>	<i>v</i>
<i>a</i>	<i>t</i>		<i>a, R1</i>	<i>b, R2</i>	<i>c</i>	<i>d</i>	<i>R2</i>		

例

$$t = a - b$$

$$u = a - c$$

$$v = t + u$$

$$a = d$$

$$d = v + u$$

LD R3, c
SUB R1, R1, R3

假设寄存器分配:

a, R1

c, R3

u, R1

<i>R1</i>	<i>R2</i>	<i>R3</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>t</i>	<i>u</i>	<i>v</i>
<i>u</i>	<i>t</i>	<i>c</i>	<i>a, R1</i>	<i>b</i>	<i>c, R3</i>	<i>d</i>	<i>R2</i>	<i>R1</i>	

例

$$t = a - b$$

$$u = a - c$$

$$v = t + u \longrightarrow \text{ADD } R3, R2, R1$$

$$a = d$$

$$d = v + u$$

假设寄存器分配:

$t, R2$

$u, R1$

$v, R3$

$R1$	$R2$	$R3$	a	b	c	d	t	u	v
u	t	v	a	b	$c, R3$	d	$R2$	$R1$	$R3$

例

$$t = a - b$$

$$u = a - c$$

$$v = t + u$$

$$a = d$$



LD R2, d

$$d = v + u$$

假设寄存器分配:

d, R2

a, R2

<i>R1</i>	<i>R2</i>	<i>R3</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>t</i>	<i>u</i>	<i>v</i>
<i>u</i>	<i>d, a</i>	<i>v</i>	<i>R2</i>	<i>b</i>	<i>c</i>	<i>d, R2</i>	<i>R2</i>	<i>R1</i>	<i>R3</i>

例

$$t = a - b$$

$$u = a - c$$

$$v = t + u$$

$$a = d$$

$$d = v + u$$

ADD R1, R3, R1

假设寄存器分配:

$v, R3$

$u, R1$

$d, R1$

<i>R1</i>	<i>R2</i>	<i>R3</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>t</i>	<i>u</i>	<i>v</i>
<i>d</i>	<i>d, a</i>	<i>v</i>	<i>R2</i>	<i>b</i>	<i>c</i>	<i>R1</i>		<i>R1</i>	<i>R3</i>

例:假设在出口处 t , u 和 v 不活跃, a , b , c 和 d 活跃

$$t = a - b$$

$$u = a - c$$

$$v = t + u$$

$$a = d$$

$$d = v + u$$

exit

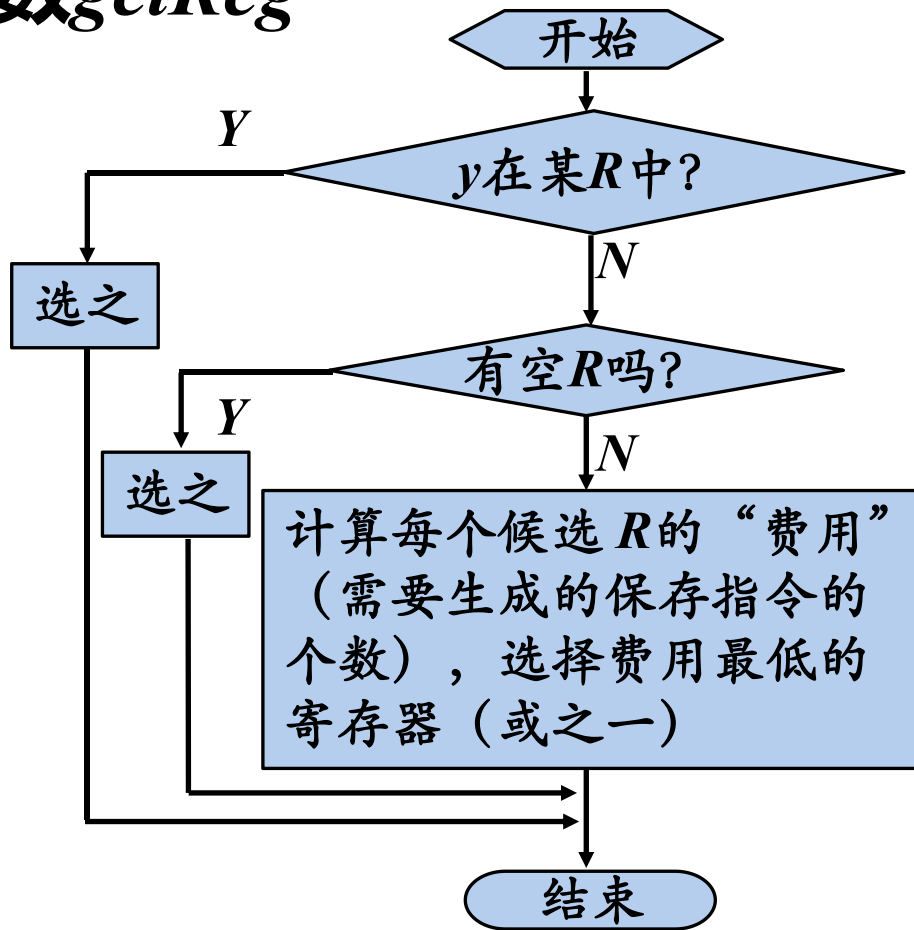
ST $a, R2$
ST $d, R1$

<i>R1</i>	<i>R2</i>	<i>R3</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>t</i>	<i>u</i>	<i>v</i>
<i>d</i>	<i>a</i>	<i>v</i>	<i>R2, a</i>	<i>b</i>	<i>c</i>	<i>R1, d</i>			<i>R3</i>

寄存器选择函数 $getReg$

$x = y \text{ op } z$

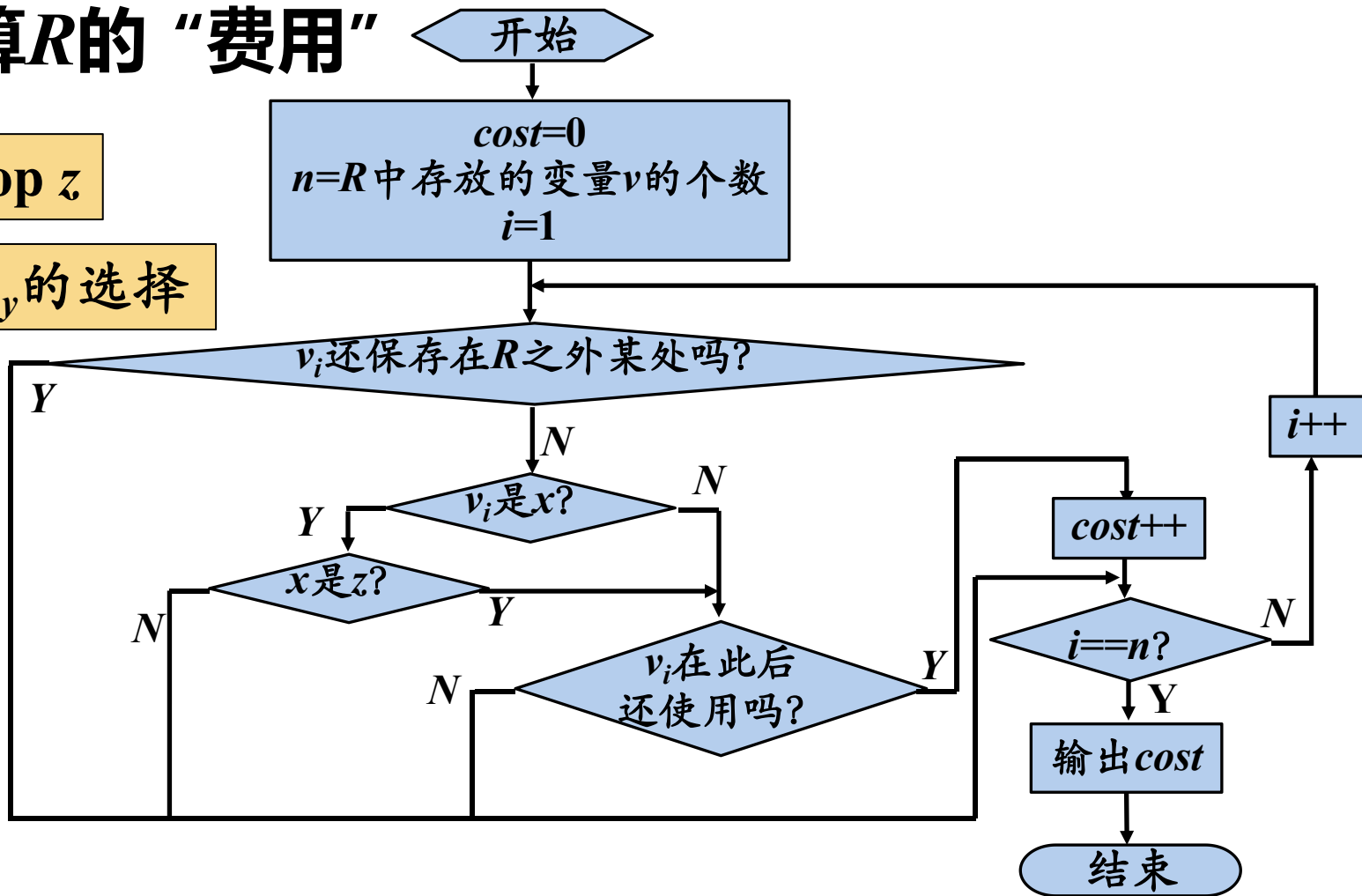
寄存器 R_y 的选择



计算 R 的“费用”

$x = y \text{ op } z$

寄存器 R_y 的选择



寄存器 R_x 的选择

$$x = y \text{ op } z$$

- 选择方法与 R_y 类似，区别之处在于
 - 因为 x 的一个新值正在被计算，因此只存放了 x 的值的寄存器对 R_x 来说总是可接受的，即使 x 就是 y 或 z 之一(因为我们的机器指令允许一个指令中的两个寄存器相同)
 - 如果 y 在指令 I 之后不再使用，且(在必要时加载 y 之后) R_y 仅仅保存了 y 的值，那么， R_y 同时也可以用作 R_x 。对 z 和 R_z 也有类似选择

当 I 是复制指令 $x=y$ 时，选择好 R_y 后，令 $R_x = R_y$



提纲

9.1 代码生成器的主要任务

9.2 一个简单的目标机模型

9.3 指令选择

9.4 寄存器选择

9.5 窥孔优化

窥孔优化

- **窥孔**(*peephole*)是程序上的一个小的滑动窗口
- **窥孔优化**是指在优化的时候，检查目标指令的一个滑动窗口(即窥孔)，并且只要有可能就在窥孔内用更快或更短的指令来替换窗口中的指令序列
- 也可以在**中间代码生成**之后直接应用窥孔优化来提高中间表示形式的质量

具有窥孔优化特点的程序变换的例子

- 冗余指令删除
- 控制流优化
- 代数优化
- 机器特有指令的使用

冗余指令删除

➤ 消除冗余的加载和保存指令

➤ 例

三地址指令序列

➤ $a = b + c$

➤ $d = a + e$

目标代码

➤ *LD* $R0, b$ // $R0 = b$

➤ *ADD* $R0, R0, c$ // $R0 = R0 + c$

➤ *ST* $a, R0$ // $a = R0$

➤ *LD* $R0, a$ // $R0 = a$

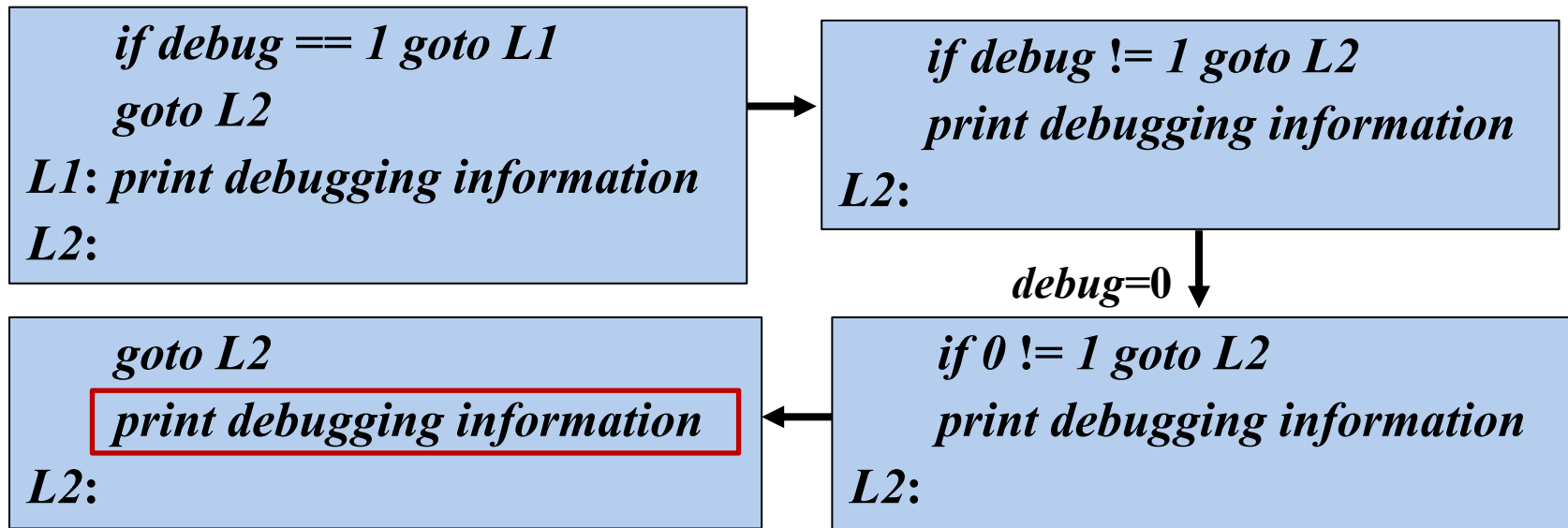
➤ *ADD* $R0, R0, e$ // $R0 = R0 + e$

➤ *ST* $d, R0$ // $d = R0$

如果第四条指令有标号，则不可以删除

冗余指令删除

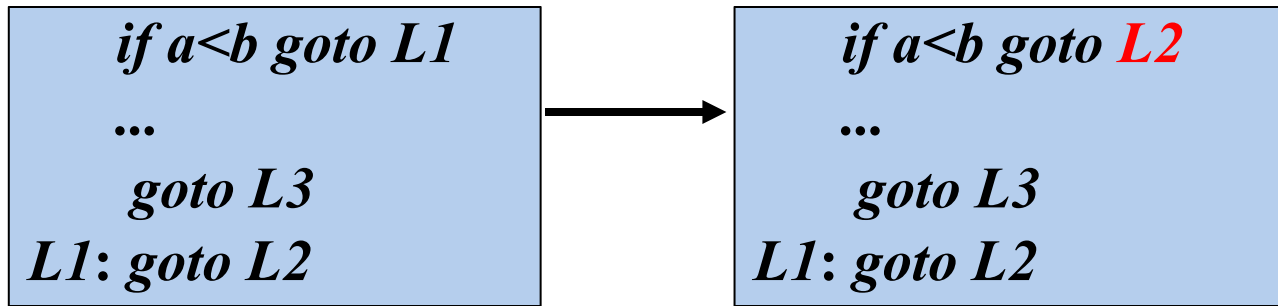
- 消除冗余的加载和保存指令
- 消除不可达代码
 - 一个紧跟在无条件跳转之后的不带标号的指令可以被删除
 - 例



控制流优化

➤ 在代码中出现**跳转到跳转指令的指令**时，某些条件下可以使用**一个跳转指令**来代替

➤ 例



如果不再有跳转到`L1`的指令，并且语句`L1: goto L2`之前是一个**无条件跳转指令**，则可以删除该语句

代数优化

➤ 代数恒等式

➤ 消除窥孔中类似于 $x=x+0$ 或 $x=x*1$ 的运算指令

➤ 强度削弱

- 对于乘数(除数)是2的幂的定点数乘法(除法)，用移位运算实现代价比较低
- 除数为常量的浮点数除法可以通过乘数为该常量倒数的乘法来求近似值

特殊指令的使用

- 充分利用目标系统的某些高效的特殊指令来提高代码效率
- 例如：*INC*指令可以用来替代加1的操作



第九章 代码生成

结束

哈尔滨工业大学 陈鄞

