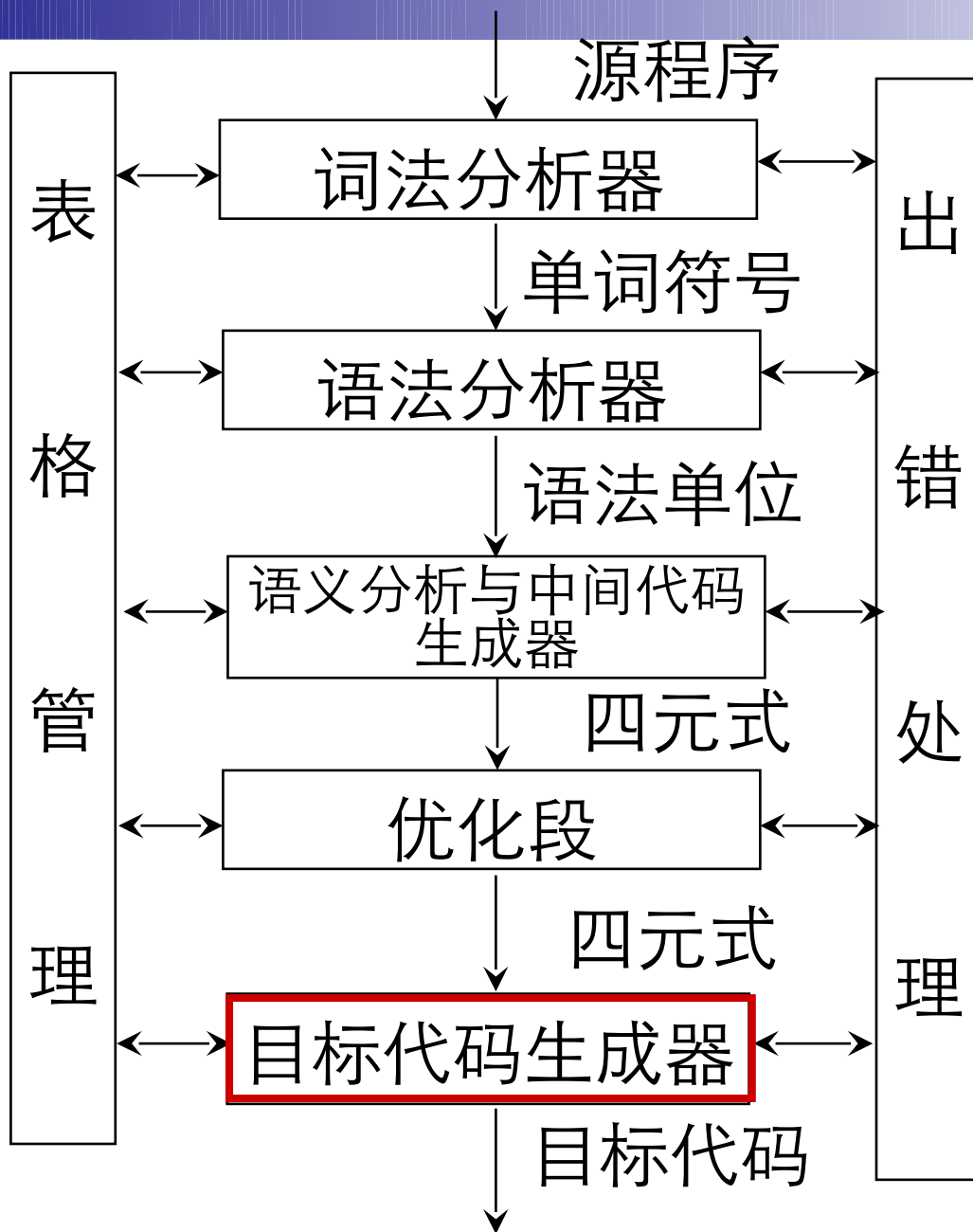




编译原理

第十一章 代码生成

编译程序总框



第十一章 代码生成

- 基本问题
- 目标机器模型
- 一个简单代码生成器

第十一章 代码生成

- 基本问题
- 目标机器模型
- 一个简单代码生成器

代码生成

■ 代码生成

- 把语法分析后或优化后的中间代码变换成目标代码

■ 目标代码的三种形式

- **绝对指令代码**：能够立即执行的机器语言代码，所有地址已经定位
- **可重新定位指令代码**：待装配的机器语言模块，执行时，由连接装配程序把它们和某些运行程序连接起来，转换成能执行的机器语言代码
- **汇编指令代码**：尚须经过汇编程序汇编，转换成可执行的机器语言代码

代码生成

- 代码生成着重考虑的问题
 - 如何使生成的目标代码较短
 - 如何充分利用计算机的寄存器，减少目标代码中访问存贮单元的次数
 - 如何充分利用计算机的指令系统的特点

11.1 基本问题

■ 代码生成器的输入

- 代码生成器的输入包括源程序的中间表示，以及符号表中的信息
- 类型检查

■ $x := y + i * j$

其中 x 、 y 为实型； i 、 j 为整型，产生的三地址代码为：

$T_1 := i \text{ int} * j$

$T_3 := \text{inttoreal } T_1$

$T_2 := y \text{ real} + T_3$

11.1 基本问题

■ 目标程序

- 绝对机器代码、可重定位机器语言、汇编语言
- 采用汇编代码作为目标语言

■ 指令选择

- $a := a + 1$
 - INC a
 - LD $R0, a$
ADD $R0, \#1$
ST $R0, a$

11.1 基本问题

■ 寄存器分配

- 在寄存器分配期间，为程序的某一点选择驻留在寄存器中的一组变量
- 在随后的寄存器指派阶段，挑出变量将要驻留的具体寄存器

■ 计算顺序选择

第十一章 代码生成

- 基本问题
- 目标机器模型
- 一个简单代码生成器

11.2 目标机器模型

- 考虑一个抽象的计算机模型
 - 具有多个通用寄存器，他们既可以作为累加器，也可以作为变址器
 - 运算必须在某个寄存器中进行
 - 含有四种类型的指令形式

类型	指令形式	意义 (设 op 是二目运算符)
直接地址型	op R_i, M	$(R_i) \text{ op } (M) \Rightarrow R_i$
寄存器型	op R_i, R_j	$(R_i) \text{ op } (R_j) \Rightarrow R_i$
变址型	op $R_i, c(R_j)$	$(R_i) \text{ op } ((R_j)+c) \Rightarrow R_i$
间接型	op $R_i, *M$ op $R_i, *R_j$	$(R_i) \text{ op } ((M)) \Rightarrow R_i$ $(R_i) \text{ op } ((R_j)) \Rightarrow R_i$
op 包括一般计算机上常见的一些运算符, 如 $\Rightarrow R_i$		

ADD(加)、SUB(减)、MUL(乘)、DIV(除)

如果 op 是一目运行符, 则 “op R_i, M ” 的意义为: $\text{op } (M) \Rightarrow R_i$, 其余类型可类推。

指 令	意 义
LD R_i, B	把 B 单元的内容取到寄存器 R，即 $(B) \Rightarrow R_i$
ST R_i, B	把寄存器 R_i 的内容存到 B 单元，即 $(R_i) \Rightarrow B$
J X	无条件转向 X 单元
CMP A, B	比较 A 单元和 B 单元的值，根据比较情况把机器内部特征寄存器 CT 置成相应状态。CT 占两个二进制位。根据 $A < B$ 或 $A = B$ 或 $A > B$ 分别置 CT 为 0 或 1 或 2。
J < X	如 CT=0 转 X 单元
J \leq X	如 CT=0 或 CT=1 转 X 单元
J = X	如 CT=1 转 X 单元
J \neq X	如 CT \neq 1 转 X 单元
J > X	如 CT=2 转 X 单元
J \geq X	如 CT=2 或 CT=1 转 X 单元

第十一章 代码生成

- 基本问题
- 目标机器模型
- 一个简单代码生成器

11.3 一个简单代码生成器

- 不考虑代码的执行效率，目标代码生成是不难的，例如：

$$A := (B + C) * D + E$$

翻译为四元式：

$$T_1 := B + C$$

$$T_2 := T_1 * D$$

$$T_3 := T_2 + E$$

$$A := T_3$$

👉 假设只有一个寄存器可供使用

- 四元式

$T_1 := B + C$

$T_2 := T_1 * D$

$T_3 := T_2 + E$

$A := T_3$

- 目标代码：

LD R_0 , B

ADD R_0 , C

ST R_0 , T_1
LD R_0 , T_1

MUL R_0 , D

ST R_0 , T_2
LD R_0 , T_2

ADD R_0 , E

ST R_0 , T_3
LD R_0 , T_3

ST R_0 , A

- 假设

T_1 , T_2 , T_3
在基本块之后
不再引用

LD R_0 , B

ADD R_0 , C

MUL R_0 , D

ADD R_0 , E

ST R_0 , A

11.3 一个简单代码生成器

- 四元式的中间代码变换成目标代码
- 在一个基本块的范围内考虑如何充分利用寄存器
 - 尽可能留：在生成计算某变量值的目标代码时，尽可能让该变量保留在寄存器中
 - 尽可能用：后续的目标代码尽可能引用变量在寄存器中的值，而不访问内存
 - 及时腾空：在离开基本块时，把存在寄存器中的现行的值放到主存中

11.3.1 待用信息

- 如果在一个基本块内，四元式 i 对 A 定值，四元式 j 要引用 A 值，而从 i 到 j 之间没有 A 的其他定值，那么，我们称 j 是四元式 i 的变量 A 的待用信息，即下一个引用点

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

$j: D := A \text{ op } E$

- 假设在变量的符号表登记项中含有记录待用信息和活跃信息的栏。

待用信息和活跃信息的表示

- (x, x) 表示变量的待用信息和活跃信息

- 第 1 元

- i 表示待用信息, \wedge 表示非待用

- 第 2 元

- y 表示

- 在符号表

- 表示后面

变量名	初始状态→信息链 (待用 / 活跃信息 栏)
T	$(\wedge, y) \rightarrow (3, y) \rightarrow (\wedge, \wedge)$
A	$(\wedge, \wedge) \rightarrow (2, y) \rightarrow (1, y)$
B	$(\wedge, \wedge) \rightarrow (1, y)$
C	$(\wedge, \wedge) \rightarrow (2, y)$
U	$(\wedge, \wedge) \rightarrow (4, y) \rightarrow (3, y) \rightarrow (\wedge, \wedge)$
V	$(\wedge, \wedge) \rightarrow (4, y) \rightarrow (\wedge, \wedge)$

待用信息和活跃信息的表示

- (x, x) 表示变量的待用信息和活跃信息
 - 第 1 元
 - i 表示待用信息, \wedge 表示非待用
 - 第 2 元
 - y 表示活跃, \wedge 表示非活跃
- 在符号表中, $(x, x) \rightarrow (x, x)$
 - 表示后面的符号对代替前面的符号对
- 不特别说明, 所有说明变量在基本块出口之后均为非活跃变量

■ 例：基本块

1. $T \leftarrow A - B$

2. $U := A - C$

3. $V := T + U$

4. $W := V + U$

- 设 W 是基本块出口之后的活跃变量。

序号	四元式	左值	左操作数	右操作数
(1)	T:=A-B	(3,y)	(2,y)	(^,^)
(2)	U:=A-C	(3,y)	(^,^)	(^,^)
(3)	V:=T+U	(4,y)	(^,^)	(4,y)
(4)	W:=V+U	(^,y)	(^,^)	(^,^)

计算待用信息和活跃信息

■ 计算待用信息和活跃信息的算法步骤

1. 开始时，把基本块中各变量的符号表登记项中的待用信息栏填为“非待用”，并根据该变量在基本块出口之后是不是活跃的，把其中的活跃信息栏填为“活跃”或“非活跃”；

变量名	待用 / 活跃信息栏
T	$(\wedge, \wedge) \rightarrow (3, y)$
A	$(\wedge, \wedge) \rightarrow (2, y)$
B	$(\wedge, \wedge) \rightarrow (1, y)$

序号	四元式	左值	左操作数	右操作数
(1)	T:=A-B	(3,y)	(2,y)	(\wedge, \wedge)
(2)	U:=A-C	(3,y)	(\wedge, \wedge)	(\wedge, \wedge)
(3)	V:=T+U	(4,y)	(\wedge, \wedge)	(4,y)
(4)	W:=V+U	(\wedge, y)	(\wedge, \wedge)	(\wedge, \wedge)

- 2. 从基本块出口到入口由后向前依次处理各个四元式。对每一个四元式 $i: A := B \text{ op } C$ ，依次执行：
 - 1) 把符号表中变量 A 的待用信息和活跃信息附加到四元式 i 上
 - 2) 把符号表中 A 的待用信息和活跃信息分别置为“非待用”和“非活跃”
 - 3) 把符号表中变量 B 和 C 的待用信息和活跃信息附加到四元式 i 上
 - 4) 把符号表中 B 和 C 的待用信息均置为 i，活跃信息均置为“活跃”

例：基本块

1. $T := A - B$

2. $U := A - C$

3. $V := T + U$

4. $W := V + U$

设 W 是基本块出口之后的活跃变量。

建立待用信息链表与活跃变量信息链表如下：

附加在四元式上的待用 / 活跃信息表

序号	四元式	左值	左操作数	右操作数
(4)	W:=V+U	(^,y)	(^,^)	(^,^)
(3)	V:=T+U	(4,y)	(^,^)	(4,y)
(2)	U:=A-C	(3,y)	(^,^)	(^,^)
(1)	T:=A-B	(3,y)	(2,y)	(^,^)

变量名	初始状态→信息链 (待用 / 活跃信息
栏) T	(^,^) → (3,y) → (^,^)
A	(^,^) → (2,y) → (1,y)
B	(^,^) → (1,y)
C	(^,^) → (2,y)
U	(^,^) → (4,y) → (3,y) → (^,^)
V	(^,^) → (4,y) → (^,^)
W	(^,y) → (^,^)

附加在四元式上的待用 / 活跃信息表

序号	四元式	左值	左操作数	右操作数
(4)	W:=V+U	(^,y)	(^,^)	(^,^)
(3)	V:=T+U	(4,y)	(^,^)	(4,y)
(2)	U:=A-C	(3,y)	(^,^)	(^,^)
(1)	T:=A-B	(3,y)	(2,y)	(^,^)

序号	四元式	左值	左操作数	右操作数
(1)	T:=A-B	(3,y)	(2,y)	(^,^)
(2)	U:=A-C	(3,y)	(^,^)	(^,^)
(3)	V:=T+U	(4,y)	(^,^)	(4,y)
(4)	W:=V+U	(^,y)	(^,^)	(^,^)

V	(^,^) \rightarrow (4,y) \rightarrow (^,^)
W	(^,y) \rightarrow (^,^)

寄存器描述和变量地址描述

- 寄存器描述数组 **RVALUE**

- 动态记录各寄存器的使用信息
- $RVALUE[R] = \{A, B\}$

- 变量地址描述数组 **AVALUE**

- 动态记录各变量现行值的存放位置
- $AVALUE[A] = \{R1, R2, A\}$

寄存器描述和变量地址描述

- 寄存器的分配局限于基本块范围之内
 - 处理完基本块中所有四元式，对现行值在寄存器中的每个变量，如它在基本块之后是活跃的，则把它在寄存器中的值存放到它的主存单元中
- 对于四元式 $A:=B$
 - 如果 B 的现行值在某寄存器 R_i 中，则无须生成目标代码
 - 只须在 $RVALUE(R_i)$ 中增加一个 A ，（即把 R_i 同时分配给 B 和 A ），并把 $AVALUE(A)$ 改为 R_i

小结

- 目标代码形式
- 代码生成着重考虑的问题
- 一个简单代码生成器
 - 待用信息
 - 活跃信息
 - 寄存器描述信息
 - 变量地址描述信息



编译原理

第十一章 代码生成

第十一章 代码生成

- 目标代码形式
- 代码生成着重考虑的问题
- 一个简单代码生成器
 - 待用信息
 - 活跃信息
 - 寄存器描述信息
 - 变量地址描述信息
 - 代码生成算法

代码生成算法

- 对每个四元式： $i: A := B \text{ op } C$ ，依次执行：
 1. 以四元式： $i: A := B \text{ op } C$ 为参数，调用函数过程 $\text{GETREG}(i: A := B \text{ op } C)$ ，返回一个寄存器 R ，用作存放 A 的寄存器。
 2. 利用 $\text{AVALUE}[B]$ 和 $\text{AVALUE}[C]$ ，确定 B 和 C 现行值的存放位置 B' 和 C' 。如果其现行值在寄存器中，则把寄存器取作 B' 和 C'

代码生成算法

3. 如果 $B' \neq R$, 则生成目标代码:

LD R, B'

op R, C'

否则生成目标代码 op R, C'

如果 B' 或 C' 为 R , 则删除
AVALUE[B] 或 AVALUE[C] 中的 R 。

4. 令 $AVALUE[A] = \{R\}$, $RVALUE[R] = \{A\}$ 。

代码生成算法

5. 若 B 或 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，且其现行值在某寄存器 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 ，使得该寄存器不再为 B 或 C 占用。

及时腾空

寄存器分配

- 寄存器分配: $\text{GETREG}(i: A := B \text{ op } C)$ 返回一个用来存放 A 的值的寄存器

1. 尽可能用 B 独占的寄存器
2. 尽可能用空闲寄存器
3. 抢占非空闲寄存器

- 寄存器分配: $\text{GETREG}(i: A:=B \text{ op } C)$ 返回一个用来存放 A 的值的寄存器

1. 尽可能用 B 独占的寄存器
2. 尽可能用空闲寄存器
3. 抢占非空闲寄存器

1 如果 B 的现行值在某个寄存器 R_i 中, $\text{RVALUE}[R_i]$ 中只包含 B , 此外, 或者 B 与 A 是同一个标识符, 或者 B 的现行值在执行四元式 $A:=B \text{ op } C$ 之后不会再引用, 则选取 R_i 为所需要的寄存器 R , 并转 4;

■ 寄存器分配： $\text{GETREG}(i: A:=B \text{ op } C)$ 返回一个用来存放 A 的值的寄存器

1. 尽可能用 B 独占的寄存器
2. 尽可能用空闲寄存器
3. 抢占非空闲寄存器

1 如果 B 的现行值在某个寄存器 R_i 中， $\text{RVALUE}[R_i]$ 中只包含 B ，此外，或者 B 与 A 是同一个标识符，或者 B 的现行值在执行四元式 $A:=B \text{ op } C$ 之后不会再引用，则选取 R_i 为所需要的寄存器 R ，并转 4；

2 如果有尚未分配的寄存器，则从中选取一个 R_i 为所需要的寄存器 R ，并转 4；

1. 尽可能用 B 所在的寄存器
2. 尽可能用空闲寄存器
3. 抢占非空闲寄存器

3 从已分配的寄存器中选取一个 R_i 为所需要的寄存器 R 。最好使得 R_i 满足以下条件：

占用 R_i 的变量的值也同时存放在该变量的贮存单元中，或者在基本块中要在最远的将来才会引用到或不会引用到。

要不要为 R_i 中的变量生成存数指令？

要不要为 R_i 中的变量生成存数指令？

对 $RVALUE[R_i]$ 中每一变量 M ，如果 M 不是 A ，或者如果 M 是 A 又是 C ，但不是 B 并且 B 也不在 $RVALUE[R_i]$ 中，则

3. 如果 $B' \neq R$ ，则生成目标代码：

LD R, B'

op R, C'

否则生成目标代码 op R, C'

如果 B' 或 C' 为 R ，则删除 $AVALUE[B]$ 或 $AVALUE[C]$ 中的 R

否则令 $AVALUE[M] = \{M\}$

(3) 删除 $RVALUE[R_i]$ 中的 M

4 给出 R ，返回。

例：基本块

1. $T := A - B$
2. $U := A - C$
3. $V := T + U$
4. $W := V + U$

设 W 是基本块出口之后的活跃变量，只有 R_0 和 R_1 是可用寄存器，生成的目标代码和相应的 RVALUE 和 AVALUE:

序号	四元式	左值	左操作数	右操作数
(4)	$W:=V+U$	(\wedge, y)	(\wedge, \wedge)	(\wedge, \wedge)
(3)	$V:=T+U$	$(4, y)$	(\wedge, \wedge)	$(4, y)$
(2)	$U:=A-C$	$(3, y)$	(\wedge, \wedge)	(\wedge, \wedge)
(1)	$T:=A-B$	$(3, y)$	$(2, y)$	(\wedge, \wedge)

$U:=A - C$ LD R_1, A R_0 含有 T T 在 R_0 中
 SUB R_1, C R_1 含有 U U 在 R_1 中

$V:=T + U$ ADD R_0, R_1 R_0 含有 V V 在 R_0 中
 R_1 含有 U U 在 R_1 中

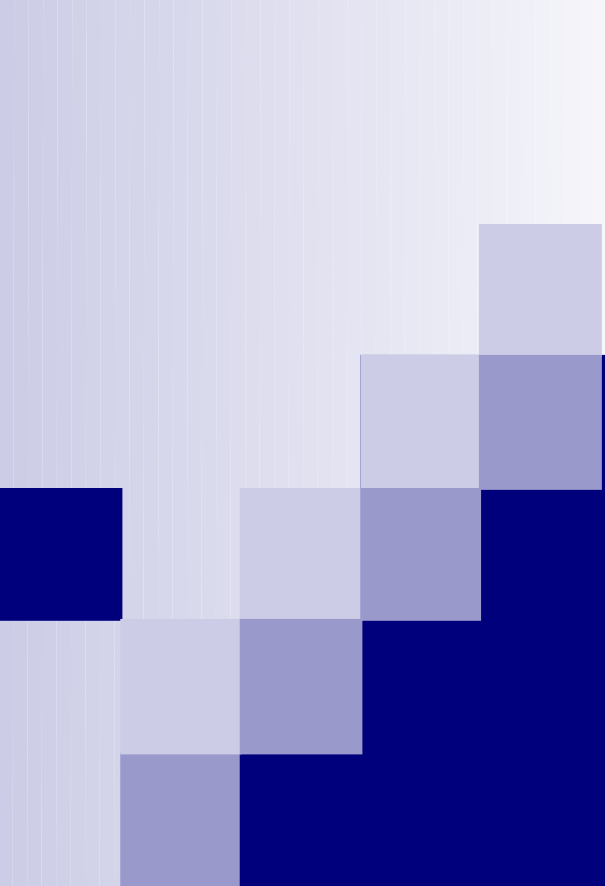
$W:=V + U$ ADD R_0, R_1 R_0 含有 W W 在 R_0 中
 ST R_0, W

本章小结

- 目标代码形式
- 代码生成着重考虑的问题
- 一个简单代码生成器
 - 待用信息
 - 活跃信息
 - 寄存器描述信息
 - 变量地址描述信息
 - 代码生成算法

作业

- P327-1



编译原理

习题课 (3)

第六章 属性文法和语法制导翻译

- 属性文法
- 基于属性文法的处理方法
- S- 属性文法的自下而上计算
- L- 属性文法和自顶向下翻译

属性文法

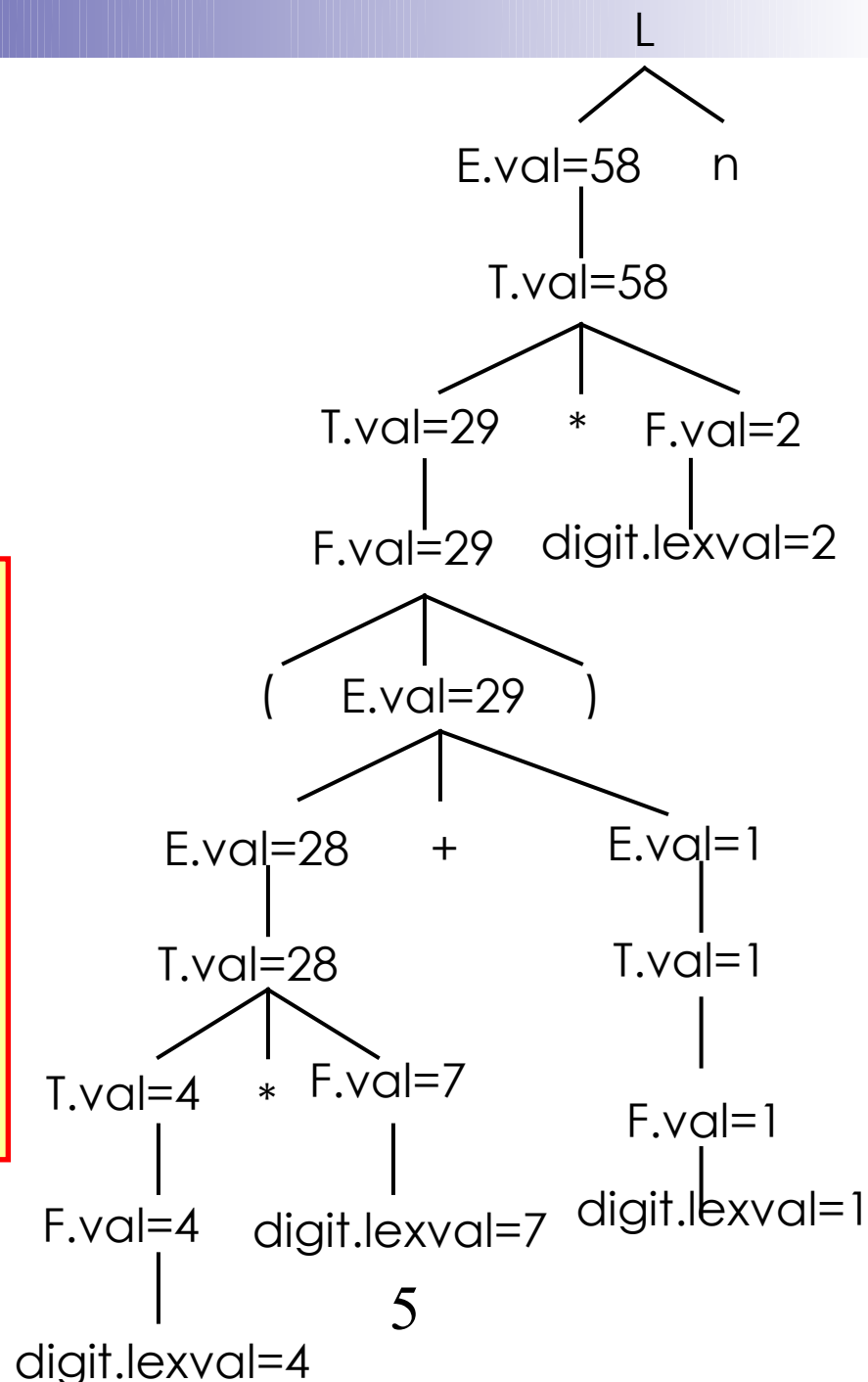
- 在上下文无关文法的基础上，为每个文法符号（终结符或非终结符）配备若干相关的“值”（称为属性）、对于文法的每个产生式都配备了一组属性的计算规则——语义规则
 - 综合属性：“自下而上”传递信息
 - 继承属性：“自上而下”传递信息

基于属性文法的处理方法

- 依赖图
- 树遍历
- 一遍扫描
 - L – 属性文法适合于一遍扫描的自上而下分析
 - S – 属性文法适合于一遍扫描的自下而上分析

P164-1 按照表 6.1 所示的属性文法，构造表达式 $(4*7+1)*2$ 的附注语法树

产生式	语义规则
$L \rightarrow En$	<code>print(E.val)</code>
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \text{digit}$	$F.val := \text{digit.lexval}$



P164-2. 对表达式 $((a)+(b))$

(1) 按照表 6.4 所示的属性
的抽象语法树;

(2) 按照图 6.17 所示的翻
达式的抽象语法树。

$$E \rightarrow T \{ R.i := T.val \}$$

$$R \{ E.val := R.s \}$$

$$R \rightarrow +$$

$$T \{ R_1.i := R.i + T.val \}$$

$$R_1 \{ R.s := R_1.s \}$$

$$R \rightarrow -$$

$$T \{ R_1.i := R.i - T.val \}$$

$$R_1 \{ R.s := R_1.s \}$$

$$R \rightarrow \varepsilon \{ R.s := R.i \}$$

$$T \rightarrow (E) \{ T.val := E.val \}$$

$$T \rightarrow \text{num} \{ T.val := \text{num.val} \}$$

产生式

语义规则

$E \rightarrow E1 + T$ $E.nptr := \text{mknode}(' + ', E1.nptr, T.nptr)$

$E \rightarrow E1 - T$ $E.nptr := \text{mknode}(' - ', E1.nptr, T.nptr)$

$E \rightarrow T$ $E.nptr := T.nptr$

$T \rightarrow (E)$ $T.nptr := E.nptr$

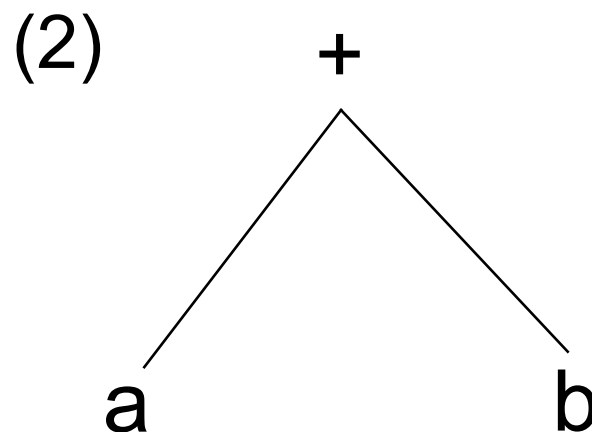
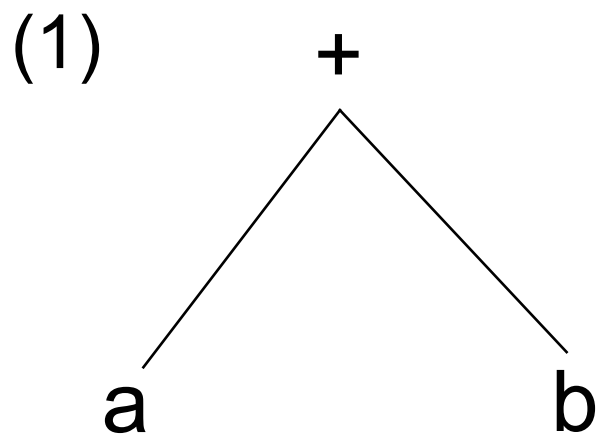
$T \rightarrow \text{id}$ $T.nptr := \text{mkleaf}(\text{id}, \text{id.entry})$

$T \rightarrow \text{num}$ $T.nptr := \text{mkleaf}(\text{num}, \text{num.val})$

P164-2. 对表达式 $((a)+(b))$:

(1) 按照表 6.4 所示的属性文法构造该表达式的抽象语法树;

(2) 按照图 6.17 所示的翻译模式, 构造该表达式的抽象语法树。



P165-5. 下列文法对整型常数和实型常数施用加法运算符 + 生成表达式；当两个整型数相加时，结果仍为整型数，否则，结果为实型数：

$$E \rightarrow E+T \mid T$$

$$T \rightarrow \text{num.num} \mid \text{num}$$

(1) 试给出确定每个子表达式结果类型的属性文法；

(2) 扩充 (1) 的属性文法，使之把表达式翻译成后缀形式，同时也能确定结果的类型。应该注意使用一元运算符 `inttoreal` 把整型数转换成实型数，以便使后缀形如 `inttoreal num` 和 `num` 运算符的两个操作数具有相同的类型。
■ 思路：对 `E` 和 `T` 设置综合属性 `type`，表表达式的类型。

P165-5. 下列文法对整型常数和实型常数施用加法运算符 + 生成表达式；当两个整型数相加时，结果仍为整型数，否则，结果为实型数：

$$E \rightarrow E + T \mid T$$
$$T \rightarrow \text{num} . \text{ num} \mid \text{num}$$

(1) 试给出确定每个子表达式结果类型的属性文法


$$\begin{array}{ll} \dot{E} \rightarrow E_1 + T & \text{if } (E_1.\text{type} = \text{int}) \text{ and } (T.\text{type} = \text{int}) \\ & \text{then } E.\text{type} := \text{int} \\ & \text{else } E.\text{type} := \text{real} \end{array}$$
$$E \rightarrow T \quad E.\text{type} := T.\text{type}$$
$$T \rightarrow \text{num} . \text{ num} \quad T.\text{type} := \text{real}$$
$$T \rightarrow \text{num} \quad T.\text{type} := \text{int}$$

P165-5. 下列文法对整型常数和实型常数施用加法运算符 + 生成表达式；当两个整型数相加时，结果仍为整型数，否则，结果为实型数：

$$E \rightarrow E + T \mid T$$

$$T \rightarrow \text{num} . \text{ num} \mid \text{num}$$


- (1) 试给出确定每个子表达式结果类型的属性文法；
- (2) 扩充 (1) 的属性文法，使之把表达式翻译成后缀形式，同时也能确定结果的类型。应该注意使用一元运算符 `inttoreal` 把整型数转换成实型数，以便使后缀形如加法运算符的两个操作数具有相同的类型。



```

E → E1 + T
    if (E1.type = int) and (T.type = int )
    then begin
        E.type := int
        E.code := E1.code || T.code || +
    end
    else if (E1.type = real) and (T.type = real)
    then begin
        E.type := real;
        E.code := E1.code || T.code || +
    End
    else if (E1.type = int)
    then begin
        E.type := real;
        E.code := E1.code || inttoreal || T.code || +
    End
    else begin
        E.type := real;
        E.code := E1.code || T.code || inttoreal || +
    end

```



$E \rightarrow T$

$E.type := T.type;$

$E.code := T.code$

$T \rightarrow num.num$

$T.type := real$

$E.code := num.num$

$T \rightarrow num$

$T.type := int$

$E.code := num$

P165-7. 下列文法由开始符号 S 产生一个二进制数，令综合属性 val 给出该数的值：

$$S \rightarrow L.L \mid L$$
$$L \rightarrow LB \mid B$$
$$B \rightarrow 0 \mid 1$$

试设计求 $S.val$ 的属性文法，其中，已知 B 的综合属性 c ，给出由 B 产生的二进位的结果值。例如，输入 101.101 时， $S.val=5.625$ ，其中第一个二进位的值是 4，最后一个二进位的值是 0.125。

- 思路：对 L 设置综合属性 val 计算二进制串 L 的值、设置综合属性 $length$ 计算 L 的长度， $L.val/2^{L.length}$ 即为小数部分 L 的值

$S \rightarrow L_1.L_2$ $S.val := L_1.val + (L_2.val / 2^{L_2.length})$

$S \rightarrow L$ $S.val := L.val$

$L \rightarrow L_1B$ $L.val := 2 * L_1.val + B.c;$
 $L.length := L_1.length + 1$

$L \rightarrow B$ $L.val := B.c;$
 $L.length := 1$

$B \rightarrow 0$ $B.c := 0$

$B \rightarrow 1$ $B.c := 1$

翻译

产生式

$E \rightarrow TR$

$R \rightarrow \text{addop } T R_1 \mid \varepsilon$

$T \rightarrow \text{num}$

语义规则

`print(addop.lexeme)`

`print(num.val)`

- **语义规则**：给出了属性计算的定义，没有属性计算的次序等实现细节
- **翻译模式**：给出了使用语义规则进行计算的次序，这样就可把某些实现细节表示出来
- 在翻译模式中，和文法符号相关的属性和语义规则（这里我们也称**语义动作**），用花括号 $\{\}$ 括起来，插入到产生式右部的合适位置上

$E \rightarrow TR$

$R \rightarrow \text{addop } T \{ \text{print(addop.lexeme)} \} R_1 \mid \varepsilon$

$T \rightarrow \text{num} \{ \text{print(num.val)} \}$

P165-11. 设下列文法生成变量的类型说明:

$$D \rightarrow \text{id } L$$
$$L \rightarrow , \text{id } L \mid : T$$
$$T \rightarrow \text{integer} \mid \text{real}$$

(1) 构造一个翻译模式，把每个标识符的类型存入符号表；参考例 6.2。

(2) 由 (1) 得到的翻译模式，构造一个预测翻译器。

- 思路：对 D,L,T 设置综合属性 type, 过程 addtype (id.entry,type) 用来将标识符 id 的类型 type 填入到符号表中

$D \rightarrow \text{id } L$

$D.\text{type} = L.\text{type} ;$
 $\text{addtype}(\text{id.entry}, L.\text{type})$

$L \rightarrow , \text{id } L_1$

$L.\text{type} = L_1.\text{type};$
 $\text{addtype}(\text{id.entry}, L_1.\text{type})$

$L \rightarrow :T$

$L.\text{type} = T.\text{type}$

$T \rightarrow \text{integer}$

$T.\text{type} = \text{integer}$

$T \rightarrow \text{real}$

$T.\text{type} = \text{real}$

第七章 语义分析和中间代码产生

- 中间语言
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

中间语言

- 后缀式，逆波兰表示
- 图表示： DAG、抽象语法树
- 三地址代码
 - 三元式
 - 四元式
 - 间接三元式

P217-1. 给出下面表达式的逆波兰表示 (后缀式) :

$a^*(-b+c)$ not A or not (C or not D)

$a+b^*(c+d/e)$ (A and B) or (not C or D)

$-a+b^*(-c+d)$ (A or B) and (C or not D and E)

if $(x+y)^*z=0$ then $(a+b)\uparrow c$ else $a\uparrow b\uparrow c$

$a^*(-b+c)$

$a+b^*(c+d/e)$

$-a+b^*(-c+d)$

not A or not (C or not D)

(A and B) or (not C or
D)

(A or B) and (C or not D
and E)

if $(x+y)^*z=0$ then
 $(a+b)^\uparrow c$ else $a^\uparrow b^\uparrow c$

$ab@c+^*$

$abcde/+^*+$

$a@bc@d+^*+$

A not C D not or not or

A B and C not D or or

A B or C D not E and or
and

$xy+z^*0= ab+c^\uparrow abc^\uparrow^\uparrow$ if-
then-else

P217-3. 请将表达式 $-(a+b)*(c+d)-(a+b+c)$ 分别表示成三元式、间接三元式和四元式序列。

三元式序列：

- (1) +, a, b
- (2) -, (1), -
- (3) +, c, d
- (4) *, (2), (3)
- (5) +, a, b
- (6) +, (5), c
- (7) -, (4), (6)

间接三元式序列：

三元式表

- (1) +, a, b
- (2) -, (1), -
- (3) +, c, d
- (4) *, (2), (3)
- (5) +, (1), c
- (6) -, (4), (5)

间接码表

- (1)
- (2)
- (3)
- (4)
- (1)
- (5)
- (6)

P217-3. 请将表达式 $-(a+b)*(c+d)-(a+b+c)$ 分别表示成三元式、间按三元式和四元式序列。

四元式序列：

(1)	+	a	b	T1
(2)	-	T1	-	T2
(3)	+	c	d	T3
(4)	*	T2	T3	T4
(5)	+	a	b	T5
(6)	+	T5	c	T6
(7)	-	T4	T6	T7

赋值语句的翻译

- 简单算术表达式及赋值语句
- 数组元素的引用
- 产生有关类型转换的指令

P218-4. 按 7.3 节所说的办法，写出下面赋值句

$$A:=B*(-C+D)$$

的自下而上语法制导翻译过程。给出所产生的三地址代码。

步骤	输入串	栈	PLACE	四元式
(1)	$A:=B*(-C+D)$			
(2)	$:=B*(-C+D)$	i	A	
(3)	$B*(-C+D)$	i:=	A-	
(4)	$*(-C+D)$	i:=I	A-B	
(5)	$*(-C+D)$	i:=E	A-B	
(6)	$(-C+D)$	i:=E*	A-B-	
(7)	$-C+D)$	i:=E*(A-B--	
(8)	$C+D)$	i:=E*(-	A-B---	
(9)	$+D)$	i:=E*(-I	A-B---C	25

P218-4. 按 7.3 节所说的办法, 写出下面赋值句

$$A:=B*(-C+D)$$

的自下而上语法制导翻译过程。给出所产生的三地址代码。

步骤	输入串	栈	PLACE	四元式
(9)	+D)	i:=E*(-I	A-B---C	
(10)	+D)	i:=E*(-E	A-B---C	(-,C,-,T1)
(11)	+D)	i:=E*(E	A-B---T1	
(12)	D)	i:=E*(E+	A-B---T1-	
(13))	i:=E*(E+I	A-B---T1-D	
(14))	i:=E*(E+E	A-B---T1-D	(+,T1,D,T2)
(15))	i:=E(E	A-B---T2	
(16)		i:=E*(E)	A-B---T2-	
(17)		i:=E+E	A-B---T2	(*,B,T2 ,T3)
(18)		i:=E	A-T3	(:=,T3 ,-,A)
(19)		A		

带数组元素引用的赋值语句翻译模式

(1) $S \rightarrow L := E$

{ if $L.offset = \text{null}$ then /*L 是简单变量 */

emit($L.place := E.place$)

else emit($L.place$ '[' $L.offset$ ']' $:= E.place$)}

(2) $E \rightarrow E_1 + E_2$

{ $E.place := \text{newtemp}$;

emit($E.place := E_1.place + E_2.place$)}

(3) $E \rightarrow (E_1) \{ E.place := E_1.place \}$

(4) $E \rightarrow L$

{ if $L.offset = \text{null}$ then
 $E.place := L.place$

else begin

$E.place := \text{newtemp};$

 emit($E.place :=$ $L.place$ $[L.offset]$)

end

}

$A[i_1, i_2, \dots, i_k]$

$((\dots i_1 n_2 + i_2) n_3 + i_3) \dots n_k + i_k) \times w +$

$\text{base} - ((\dots ((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3) \dots) n_k + \text{low}_k) \times w$

(8) $\text{Elist} \rightarrow \text{id} [E$

{ $\text{Elist.place} := E.\text{place};$

$\text{Elist.ndim} := 1;$

$\text{Elist.array} := \text{id.place} \}$

$A[i_1, i_2, \dots, i_k]$

$((\dots i_1 n_2 + i_2) n_3 + i_3) \dots n_k + i_k) \times w +$

$\text{base} - ((\dots ((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3) \dots) n_k + \text{low}_k) \times w$

(7) $\text{Elist} \rightarrow \text{Elist}_1, E$

```
{  t:=newtemp;
   m:=Elist1.ndim+1;
   emit(t ':=' Elist1.place '*' limit(Elist1.array,m) );
   emit(t ':=' t '+' E.place);
   Elist.place:=t;
   Elist.ndim:=m
   Elist.array:= Elist1.array;
```

}

$A[i_1, i_2, \dots, i_k]$

$((\dots i_1 n_2 + i_2) n_3 + i_3) \dots n_k + i_k) \times w +$

$\text{base} - ((\dots ((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3) \dots) n_k + \text{low}_k) \times w$

(5) $L \rightarrow \text{Elist}$]

{ L.place := newtemp;

emit(L.place := Elist.array ' - ' C);

L.offset := newtemp;

emit(L.offset := w '*' Elist.place) }

(6) $L \rightarrow \text{id}$ { L.place := id.place; L.offset := null }

P218-5. 按照 7.3.2 节所给的翻译模式，把下列赋值句翻译为三地址代码：

$$A[i,j]:=B[i,j]+C[A[k,l]]+D[i+j]$$

设：

A 、 B : 10*20

C 、 D : 20 ,

宽度 $w = 4$

下标从 1 开始

$A[i,j] := B[i,j] + C[A[k,l]] + D[i+j]$

T1 := i * 20

T1 := T1 + j

T2 := A - 84

T3 := 4 * T1

T4 := i * 20

T4 := T4 + j

T5 := B - 84

T6 := 4 * T4

T7 := T5[T6]

(5) L → E list] { L.place := newtemp;

emit(L.place ':=' Elist.array ' - ' C);

L.offset := newtemp;

(1) S → L := E

{ if L.offset = null then /* L 是简单变量 */

emit(L.place ':=' E.place)

else emit(L.place '[' L.offset ']' ':=' E.place) }

(4) E → L

{ if L.offset = null then

E.place := L.place

else begin

E.place := newtemp;

emit(E.place ':=' L.place '[' L.offset '])

end }



$A[i,j] := B[i,j] + C[A[k,l]] + D[i+j]$

T1 := i * 20

T1 := T1 + j

T2 := A - 84

T3 := 4 * T1

T4 := i * 20

T4 := T4 + j

T5 := B - 84

T6 := 4 * T4

T7 := T5[T6]

T8 := k * 20

T8 := T8 + l

T9 := A - 84

T10 := 4 * T8

T11 := T9[T10]

T12 := C - 4

T13 := 4 * T11

T14 := T12[T13]

T15 := T7 + T14

T16 := i + j

T17 := D - 4

T18 := 4 * T16

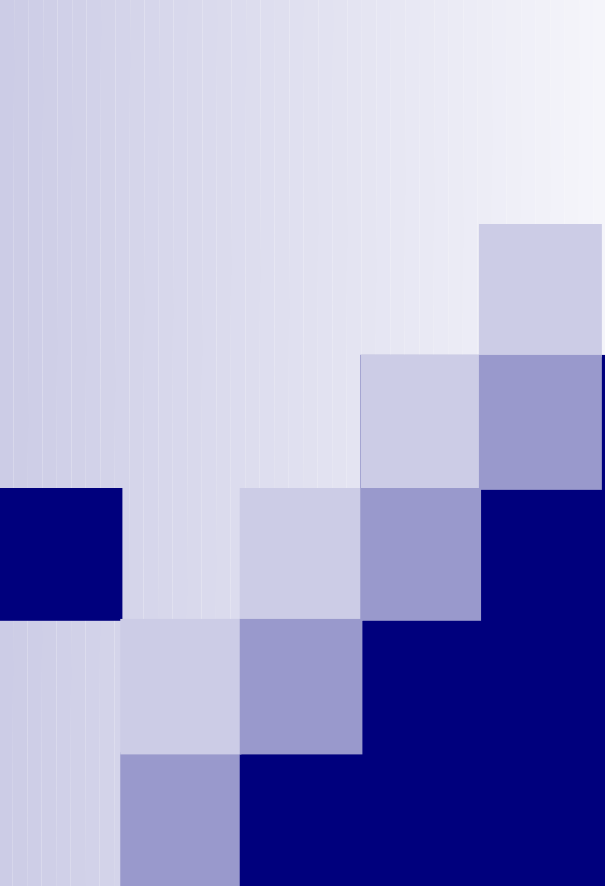
T19 := T17[T18]

T20 := T15 + T19

T2[T3] := T20

小结

- 属性文法和语法制导翻译
 - 属性计算
 - 根据语义设计属性文法
 - 根据语义设计翻译模式
- 语义分析和中间代码产生
 - 表达式的中间表示
 - 后缀式、DAG、抽象语法树、三地址代码
 - 翻译成四元式、构造翻译模式
 - 算术表达式



编译原理

习题课 (4)

第七章 语义分析和中间代码产生

- 翻译成四元式、构造翻译模式
 - 布尔表达式
 - 赋值语句
 - 控制语句

(1) $E \rightarrow E_1 \text{ or } M E_2$
 { backpatch(E_1 .falselist, M.quad);
 E .truelist:=merge(E_1 .truelist, E_2 .truelist);
 E .falselist:= E_2 .falselist }

(2) $E \rightarrow E_1 \text{ and } M E_2$
 { backpatch(E_1 .truelist, M.quad);
 E .truelist:= E_2 .truelist;
 E .falselist:=merge(E_1 .falselist, E_2 .falselist) }

(5) $E \rightarrow \text{id}_1 \text{ relop id}_2$
 { E .truelist:=makelist(nextquad);
 E .falselist:=makelist(nextquad+1);
 emit('j' relop.op ',' id_1 .place ',' id_2 .place ',' '0');
 emit('j, - , - , 0') }

(6) $E \rightarrow \text{id}$
 { E .truelist:=makelist(nextquad);
 E .falselist:=makelist(nextquad+1);
 emit('jnz' ',' id.place ',' '-' ',' '0') ;
 emit('j - , - , 0') }

(3) $E \rightarrow \text{not } E_1$
 { E .truelist:= E_1 .falselist;
 E .falselist:= E_1 .truelist }

(4) $E \rightarrow (E_1)$
 { E .truelist:= E_1 .truelist;
 E .falselist:= E_1 .falselist }

~~(7) $M \rightarrow \epsilon$ { M.quad:=nextquad }~~

(1) $E \rightarrow E_1$ or $M E_2$

```
{ backpatch( $E_1$ .falselist, M.quad);  
   $E$ .truelist:=merge( $E_1$ .truelist,  $E_2$ .truelist);  
   $E$ .falselist:= $E_2$ .falselist }
```

(2) $E \rightarrow E_1$ and $M E_2$

```
{ backpatch( $E_1$ .truelist, M.quad);  
   $E$ .truelist:= $E_2$ .truelist;  
   $E$ .falselist:=merge( $E_1$ .falselist,  $E_2$ .falselist) }
```

(5) $E \rightarrow id_1 \text{ relop } id_2$

```
{  $E$ .truelist:=makelist(nextquad);  
   $E$ .falselist:=makelist(nextquad+1);  
  emit('j' relop.op ','  $id_1$ .place ','  $id_2$ .place ',' '0');  
  emit('j, - , - , 0') }
```

(6) $E \rightarrow id$

```
{  $E$ .truelist:=makelist(nextquad);  
   $E$ .falselist:=makelist(nextquad+1);  
  emit('jnz' ',' id.place ',' ' - ' ',' ' 0') ;
```

(3) $E \rightarrow \text{not } E_1$

```
{  $E$ .truelist:= $E_1$ .falselist;  
   $E$ .falselist:= $E_1$ .truelist }
```

(4) $E \rightarrow (E_1)$

```
{  $E$ .truelist:= $E_1$ .truelist;  
   $E$ .falselist:= $E_1$ .falselist }
```

~~(7) $M \rightarrow \epsilon$ { M .quad:=nextquad }~~

P218-6. 按 7.4.2 节的办法，写出布尔式 $A \text{ or } (B \text{ and not } (C \text{ or } D))$ 的四元式序列。

四元式序列

100. (jnz, A, -, 0)

101. (j, -, -, 102)

102. (jnz, B, -, 104)

103. (j, -, -, 0)

104. (jnz, C, -, 103)

105. (j, -, -, 106)

106. (jnz, D, -, 104)

107. (j, -, -, 100)



falselist



truelist

$S \rightarrow \text{if } E \text{ then } M \ S_1$
 { backpatch(E.truelist, M.quad);
 $S.\text{nextlist} := \text{merge}(E.\text{falselist}, S_1.\text{nextlist})$ }

$S \rightarrow \text{if } E \text{ then } M_1 \ S_1 \ N \ \text{else } M_2 \ S_2$
 { backpatch(E.truelist, $M_1.\text{quad}$);
 backpatch(E.falselist, $M_2.\text{quad}$);
 $S.\text{nextlist} := \text{merge}(S_1.\text{nextlist}, N.\text{nextlist}, S_2.\text{nextlist})$ }

$M \rightarrow \varepsilon$ { $M.\text{quad} := \text{nextquad}$ }

$N \rightarrow \varepsilon$ { $N.\text{nextlist} := \text{makelist}(\text{nextquad}, \dots)$ }

☐ $\text{emit}(j, S, -, -')$ }

☐ 标号与 goto 语句

☐ CASE 语句

■ 过程调用的翻译

$S \rightarrow \text{while } M_1 \ E \ \text{do } M_2 \ S_1$
 { backpatch(E.truelist, $M_2.\text{quad}$);
 backpatch($S_1.\text{nextlist}$, $M_1.\text{quad}$);
 $S.\text{nextlist} := E.\text{falselist}$
 $\text{emit}(j, -, -, 'M_1.\text{quad})$ }

面的语句翻

```
S → if E then M S1
{ backpatch(E.truelist, M.quad);
  S.nextlist := merge(E.falselist, S1.nextlist) }
```

```
S → if E then M1 S1 N else M2 S2
{ backpatch(E.truelist, M1.quad);
  backpatch(E.falselist, M2.quad);
  S.nextlist := merge(S1.nextlist, N.nextlist, S2.nextlist) }
```

```
M → ε { M.quad := nextquad }
```

```
N → ε { N.nextlist := makelist(nextquad, ...
  emit('j, - , - , - ' ) }
```

do A:=A+2

```
S → while M1 E do M2 S1
{ backpatch(E.truelist, M2.quad);
  backpatch(S1.nextlist, M1.quad);
  S.nextlist := E.falselist
  emit('j, - , - , ' M1.quad) }
```

P218-7. 用 7.5.1 节的办法，把下面的语句翻译成四元式序列：

while $A < C$ and $B < D$ do

if $A=1$ then $C:=C+1$

else while $A \leq D$ do $A:=A+2$

100. (j<, A, C, 102)

101. (j, -, -, 0)

102. (j<, B, D, 104)

103. (j, -, -, 101)

104. (j=, A, '1', 106)

105. (j, -, -, 109)

106. (+, C, '1', T1)

107. (:=, T1, -, C)

108. (j, -, -, 100)

109. (j≤, A, D, 111)

110. (j, -, -, 100)

111. (+, A, '2', T2)

112. (:=, T2, -, A)

113. (j, -, -, 109)

114. (j, -, -, 100)

12. Pascal 语言中 for 语句的一般形式为

for v:=initial to final do S

其意义如下:

begin

t1:=initial; t2:=final;

if $t1 \leq t2$ then begin

v:=t1;

S;

while $v \neq t2$ do begin

v:=succ(v);

S;

end

end

end

(1) 设有下列 Pascal 程序:

program

forloop(input,output);

var i, initial, final: integer;

begin

read (initial, final);

for i:=initial to final do


writeln (i)

end

当 initial=MAXINT-

5、 final=MAXINT 时, 该程序的执行结果是什么? 其中 MAXINT 是目标机上能表示的最大整数。

(2) 试构造一个翻译模式, 把 Pascal 语言的 for 语句翻译成四元式。



```
program forloop (input,  
    output);  
var i, initial, final:integer;  
begin  
    read (initial, final);  
    for i:=initial to final do  
        writeln (i)  
    end
```

```
MAXINT – 5  
MAXINT – 4  
MAXINT – 3  
MAXINT – 2  
MAXINT – 1  
MAXINT
```


构造翻译模式

for $v := \text{initial}$ to final do S

其意义如下：

```
begin
  t1:=initial; t2:=final;
  if t1 ≤ t2 then begin
    v:=t1;
    S;
    while v ≠ t2 do begin
      v:=succ(v);
      S;
    end
  end
end
end
```

$S \rightarrow \text{for id} := E_1 \text{ to } E_2 \text{ do } S_1$

改造为：

$S \rightarrow F S_1$

$F \rightarrow \text{for id} := E_1 \text{ to } E_2 \text{ do}$

for v:=initial to final do S

其意义如下:

begin

t1:=initial; t2:=final;

if t1 ≤ t2 then begin

v:=t1;

S;

while v ≠ t2 do begin

v:=succ(v);

S;

end

end

end

F → for id:=E₁ to E₂ do

{

INITIAL=NEWTEMP;

emit(':=,' E₁.PLACE ', -,' INITIAL);

FINAL=NEWTEMP;

emit(':=,' E₂.PLACE ', -,' FINAL);

p:= nextquad+2;

emit('j≤,' INITIAL ', FINAL ', p);

F.nextlist:=makelist(nextquad);

emit('j, - , - , - ');

F.place:=lookup(id.name);

if F.place≠nil then

emit(':=', F.place, '-', INITIAL);

F.quad:=nextquad;

F.final:=FINAL;

}

for v:=initial to final do S

其意义如下:

begin

t1:=initial; t2:=final;

if t1 ≤ t2 then begin

v:=t1;

S;

while v ≠ t2 do begin

v:=succ(v);

S;

end

end

end

$S \rightarrow F S_1$

{

backpatch(S_1 .nextlist, nextquad)

$p := \text{nextquad} + 2;$

emit('j≠,' F.place', ' F.final ', ' p);

$S.\text{nextlist} := \text{merge}(F.\text{nextlist}, \text{make_list}(\text{nextquad}));$

emit('j, - , - , - ');

emit('succ,' F.place ', -, ' F.place);

$\text{emit}('j, - , - , ' F.\text{quad});$

}

```

F → for id := E1 to E2 do
{
    INITIAL = NEWTEMP;
    emit(':=', E1.PLACE, '-', INITIAL);
    FINAL = NEWTEMP;
    emit(':=', E2.PLACE, '-', FINAL);
    p := nextquad + 2;
    emit('j ≤', INITIAL, ',', FINAL, ',', p);
    F.nextlist := makelist(nextquad);
    emit('j, -, -, - ');
    F.place := lookup(id.name);
    if F.place ≠ nil then
        emit(':=', F.place, '-', INITIAL);
    F.quad := nextquad;
    F.final := FINAL;
}

```

```

S → F S1
{
    backpatch(S1.nextlist, nextquad)
    p := nextquad + 2;
    emit('j ≠', F.place, ',', F.final, ',', p);
    S.nextlist := merge(F.nextlist, makelist(nextquad));
    emit('j, -, -, - ');

    emit('succ, ' F.place, '-', F.place);
    emit('j, -, -, ' F.quad);
}

```

构造翻译模式

for $v := \text{initial}$ to final do S

其意义如下:

```
begin
   $t1 := \text{initial}; t2 := \text{final};$ 
  if  $t1 \leq t2$  then begin
     $v := t1;$ 
     $S;$ 
    while  $v \neq t2$  do begin
       $v := \text{succ}(v);$ 
       $S;$ 
    end
  end
end
end
```

$S \rightarrow \text{for } id := E_1 \text{ to } E_2 \text{ do } S_1$

改造为:

$S \rightarrow F \text{ do } M S_1$

$F \rightarrow \text{for } l := E_1 \text{ to } E_2$

$l \rightarrow id$

$M \rightarrow \varepsilon$

构造翻译模式

for $v := \text{initial}$ to final do S

其意义如下:

```
begin
  t1:=initial; t2:=final;
  if t1 ≤ t2 then begin
    v:=t1;
    S;
    while v ≠ t2 do begin
      v:=succ(v);
      S;
    end
  end
end
end
```

$S \rightarrow \text{for } id := E_1 \text{ to } E_2 \text{ do } S_1$

改造为:

$S \rightarrow F \text{ do } M S_1$

$F \rightarrow \text{for } l := E_1 \text{ to } E_2$

$l \rightarrow id$

```
l → id
{
  p:=lookup(id.name);
  if p <> nil then
    l.place := p
  else error
}

M → ε
{
  M.quad := nextquad
}
```

for v:=initial to final do S

其意义如下:

begin

 t1:=initial; t2:=final;

 if $t1 \leq t2$ then begin

 v:=t1;

 S;

 while $v \neq t2$ do begin

 v:=succ(v);

 S;

 end

 end

end

$F \rightarrow \text{for } I := E_1 \text{ to } E_2$

{

 F.falselist:= makelist(nextquad);

 emit('j>,' E₁.place ',' E₂.place ',0');

 emit(I.Place ':='E₁.place);

 F.truelist := makelist(nextquad);

 emit('j,-,-,0');

 F.place := I.place;

 F.end := E₂.place;

}

$S \rightarrow F \text{ do } M S_1$

{

 backpatch(S₁.nextlist, nextquad);

 backpatch(F.truelist, M.quad);

 emit(F.place ':='F.place '+1');

 emit('j<=,' F.place ',' F.end ',' M.quad);

 S.nextlist := F.falselist;

}

```

S → F do M S1
{
    backpatch(S1.nextlist, nextquad);
    backpatch(F.truelist, M.quad);
    emit(F.place ':=' F.place '+' 1);
    emit('j', F.place ',', F.end ',', M.quad);
    S.nextlist := F.falselist;
}

```

```

F → for I := E1 to E2
{
    F.falselist := makelist(nextquad);
    emit('j>', E1.place ',', E2.place ',0');
    emit(I.Place ':=' E1.place);
    F.truelist := makelist(nextquad);
    emit('j,-,-,-');
    F.place := I.place;
    F.end := E2.place;
}

```

```

I → id
{
    p := lookup(id.name);
    if p <> nil then
        I.place := p
    else error
}

M → ε
{
    M.quad := nextquad
}

```


第九章 运行时存储空间组织

- 目标程序运行时的活动
- 运行时存储器的划分
- 静态存储管理
- 一个简单栈式存储分配
- 嵌套过程语言的栈式实现

P270-9. 对于下面的程序：

```
procedure P(X,Y,Z);  
begin  
    Y:=Y+1;  
    Z:=Z+X;  
end P;  
begin  
    A:=2;  
    B:=3;  
    P(A+B,A,A);  
    print A  
end
```

若参数传递的办法分别为（1）传名，（2）传地址，（3）得结果，以及（4）传值，试问，程序执行时所输出的 A 分别是什么？

（1）传名 $A = 9$

（2）传地址
 $A = 8$

（3）得结果
 $A = 7$


（4）传值 $A = 2$

第十章 优化

- 优化概述
- 局部优化
- 循环优化

P306-1. 试把以下程序划分为基本块并作出其程序流图。

```
    read C
    A:=0
    B:=1
L1:  A:=A+B
    if  $B \geq C$  goto L2
    B:=B+1
    goto L1
L2:  write A
    halt
```



```
read C
A: =0
B:=1
L1:A:=A+B
  if  $B \geq C$  goto L2
  B:=B+1
  goto L1
L2:write A
halt
```

1. 求出四元式程序中各个基本块的入口语句：

- 1) 程序第一个语句，或
- 2) 能由条件转移语句或无条件转移语句转移到的语句，或
- 3) 紧跟在条件转移语句后面的语句

read C

A:=0

B:=1

L1:A:=A+B

if $B \geq C$ goto L2

B:=B+1

goto L1

L2:write A

halt

1. 求出四元式程序中各个基本块的入口语句：

- 1) 程序第一个语句，或
- 2) 能由条件转移语句或无条件转移语句转移到的语句，或
- 3) 紧跟在条件转移语句后面的语句

read C

A:=0

B:=1

L1:A:=A+B

if $B \geq C$ goto L2

B:=B+1

goto L1

L2:write A

halt

1. 求出四元式程序中各个基本块的入口语句：

- 1) 程序第一个语句，或
- 2) 能由条件转移语句或无条件转移语句转移到的语句，或
- 3) 紧跟在条件转移语句后面的语句

read C

A:=0

B:=1

L1:A:=A+B

if $B \geq C$ goto L2

B:=B+1

goto L1

L2:write A

halt

1. 求出四元式程序中各个基本块的入口语句：

- 1) 程序第一个语句，或
- 2) 能由条件转移语句或无条件转移语句转移到的语句，或
- 3) 紧跟在条件转移语句后面的语句

read C

A:=0

B:=1

L1:A:=A+B

if $B \geq C$ goto L2

B:=B+1

goto L1

L2:write A

halt

read C

A: =0

B:=1

L1:A:=A+B

if $B \geq C$ goto L2

B:=B+1

goto L1

L2:write A

halt

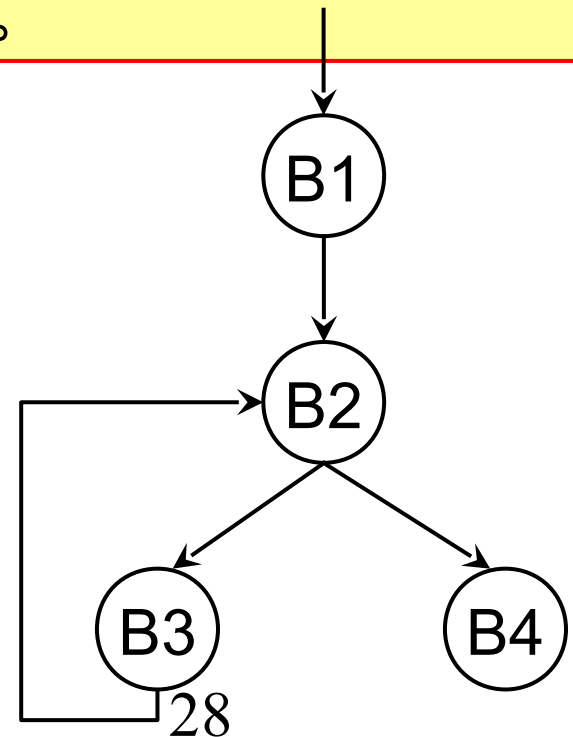
2. 对以上求出的每个入口语句，确定其所属的基本块。它是由该入口语句到下一入口语句（不包括该入口语句）、或到一转移语句（包括该转移语句）、或一停语句（包括该停语句）之间的语句序列组成的。

B1

B2

B3

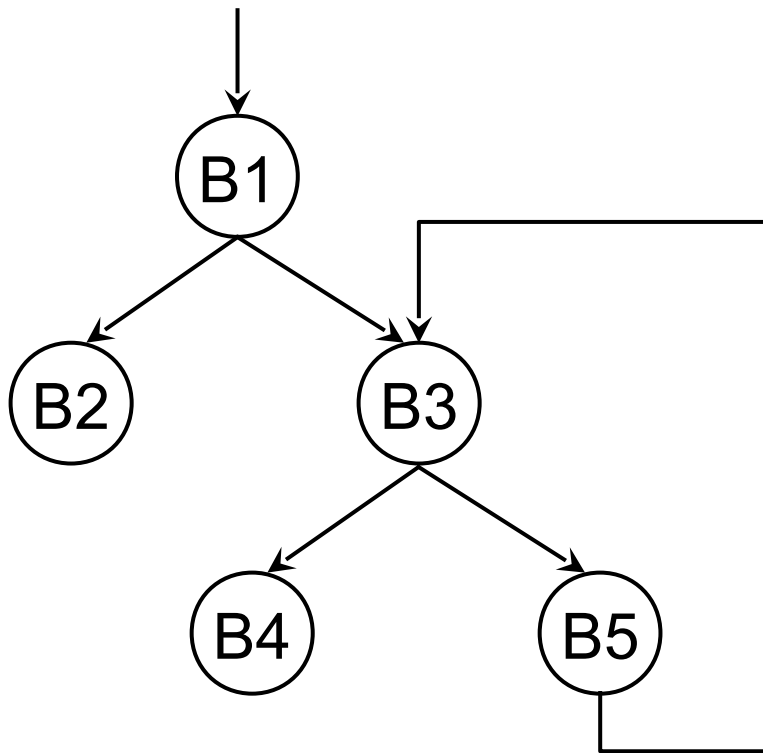
B4



P306-2. ☐ ☐ ☐ ☐ ☐ ☐

☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐

☐ ☐ ☐ ☐



read A, B

F:=1

C:=A*A

D:=B*B

if C<D goto L1

B1

E:=A*A

F:=F+1

E:=E+F

write E

halt

B2

L1: E:=B*B

F:=F+2

E:=E+F

write E

if E>100 goto L2

B3

halt

B4

L2: F:=F-1

goto L1

B5

P306-3. 试对以下基本块 B1 和 B2 :

B1: A:=B*C

D:=B/C

E:=A+D

F:=2*E

G:=B*C

H:=G*G

F:=H*G

L:=F

M:=L

B2: B:=3

D:=A+C

E:=A*C

G:=B*F

H:=A+C

I:=A*C

J:=H+I

K:=B*5

L:=K+J

M:=L

分别应用 DAG 对它们进行优化，并就以下两种情况分别写出优化后的四元式序列：

(1) 假设只有 G、L、M 在基本块后面还要被引用；

(2) 假设只有 L 在基本块后面还要被³⁰引用。

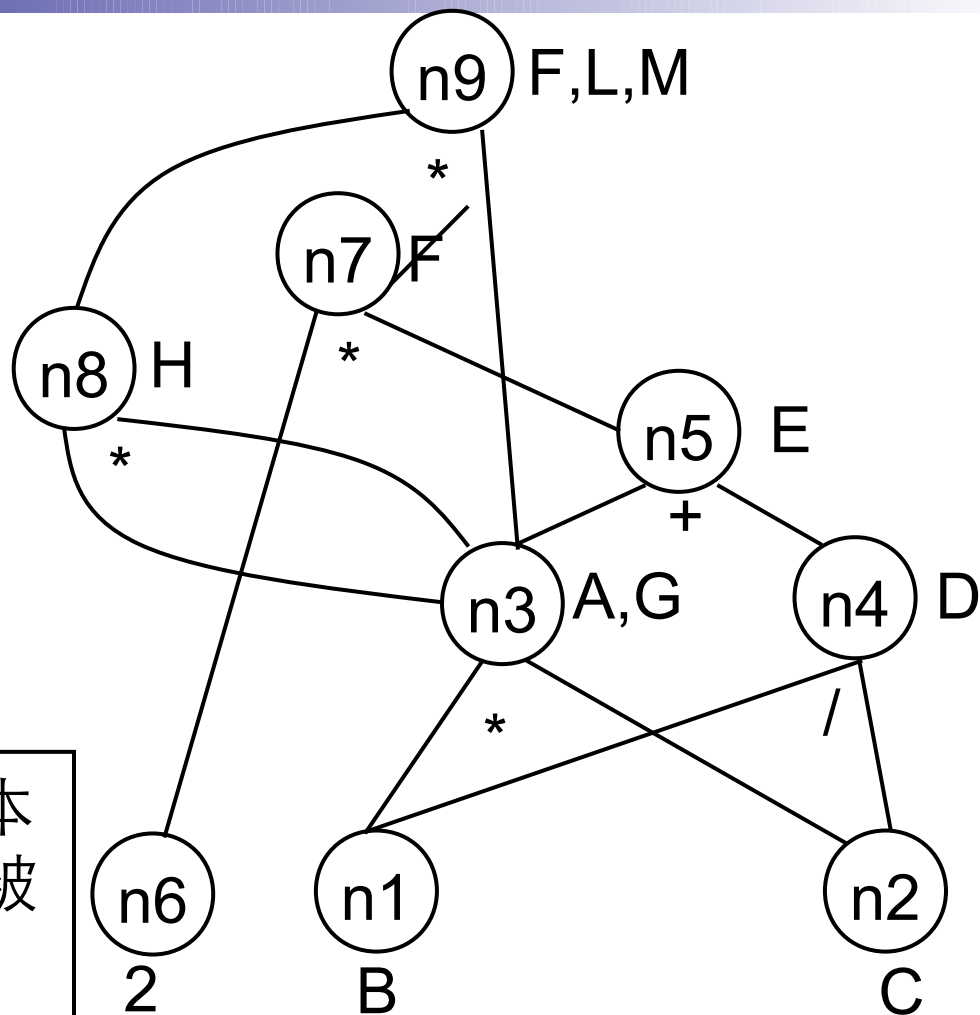
B1: $A := B * C$
 $D := B / C$
 $E := A + D$
 $F := 2 * E$
 $G := B * C$
 $H := G * G$
 $F := H * G$
 $L := F$
 $M := L$

只有 G,L,M
 在基本块后面
 还要被引用:

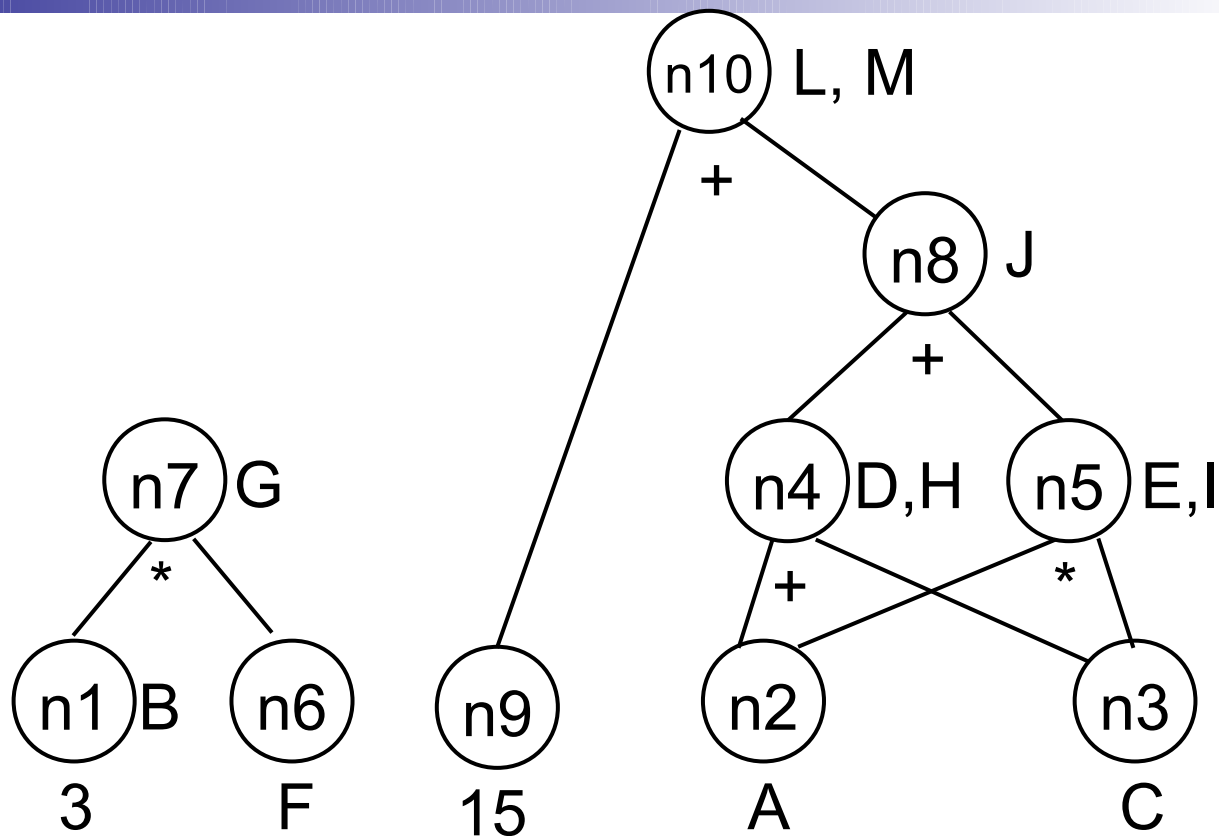
$G := B * C$
 $H := G * G$
 $L := H * G$
 $M := L$

只有 L 在基本
 块后面还要被
 引用:

$G := B * C$
 $H := G * G$
 $L := H * G$



B2: B:=3
 D:=A+C
 E:=A*C
 G:=B*F
 H:=A+C
 I:=A*C
 J:=H+I
 K:=B*5
 L:=K+J
 M:=L



只有 G,L,M 在基本块后面还要被引用

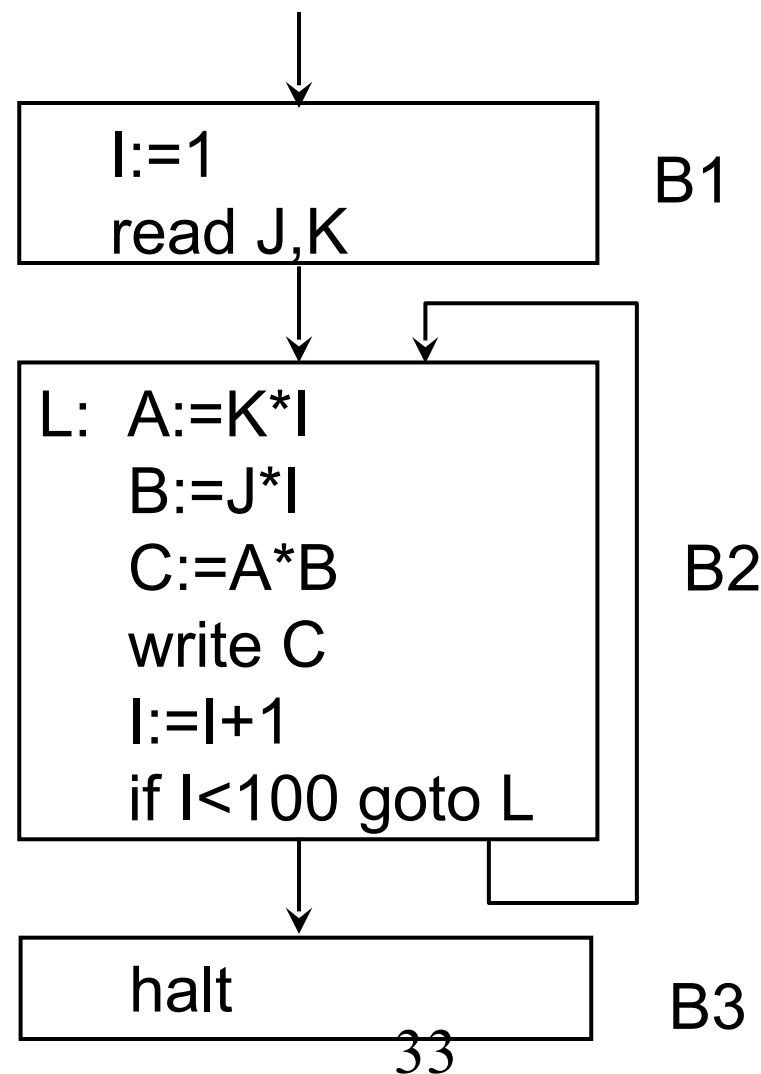
:
 G:=3*F
 D:=A+C
 E:=A*C
 J:=D+E
 L:=15+J

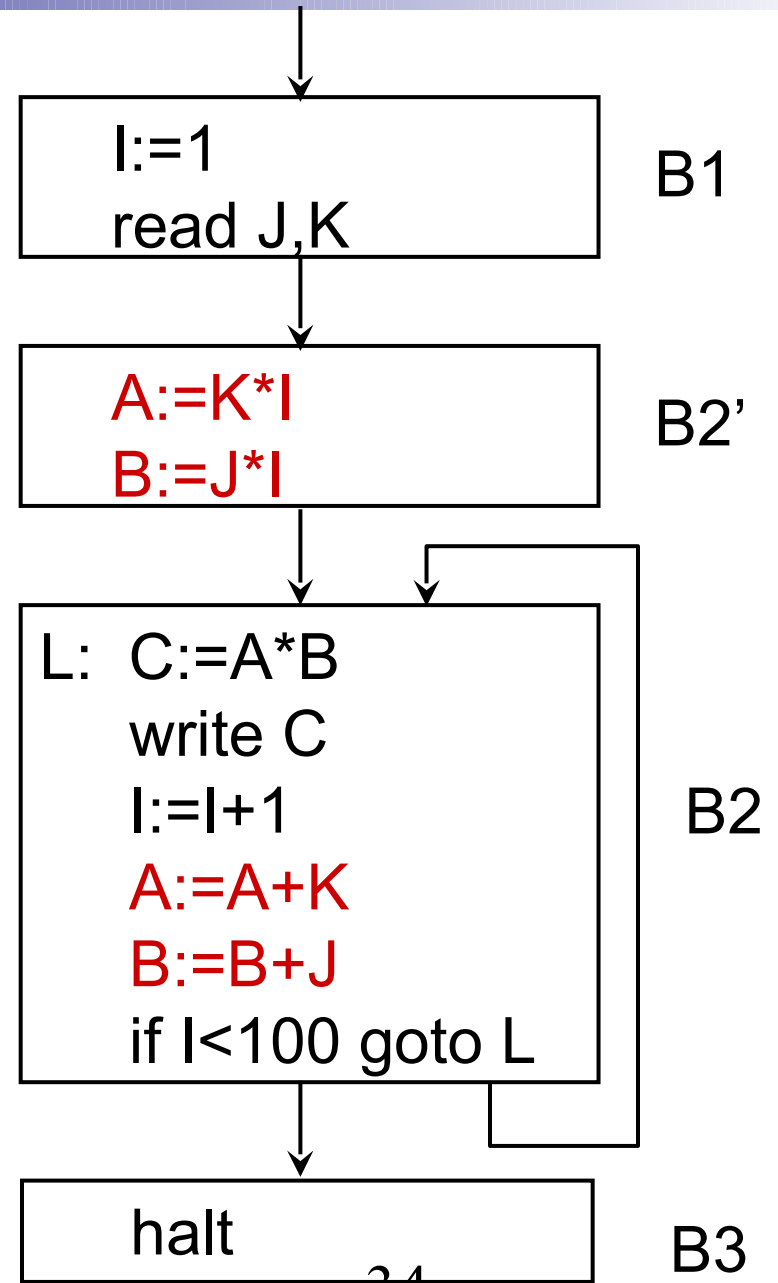
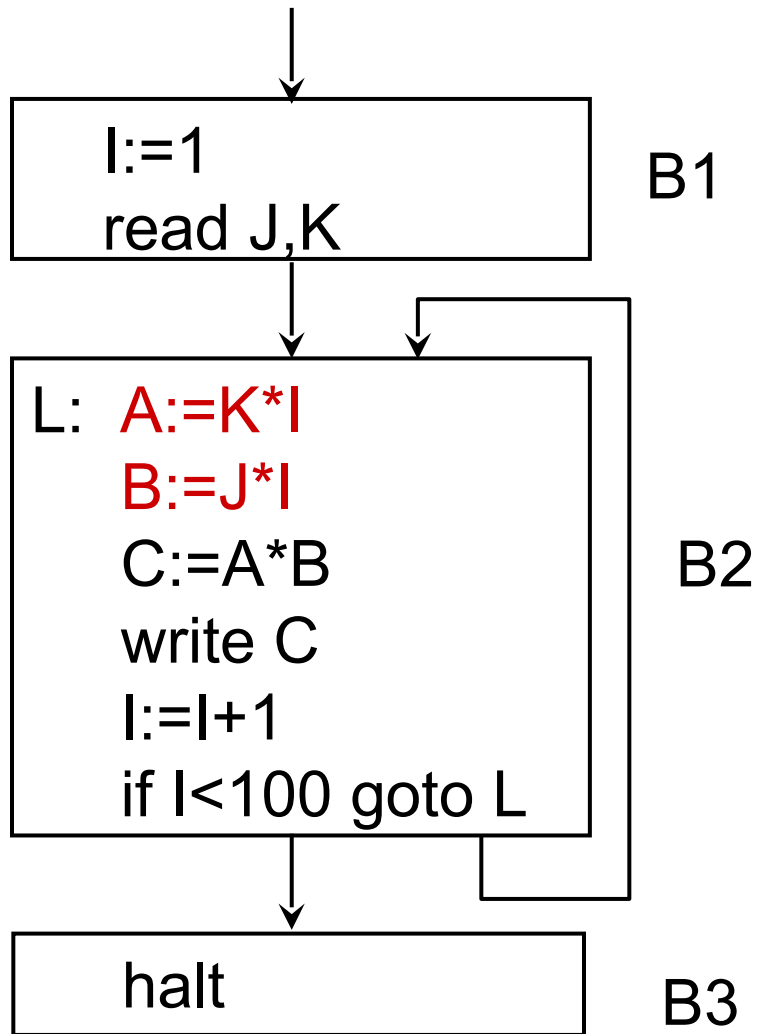
只有 L 在基本块后面还要被引用:

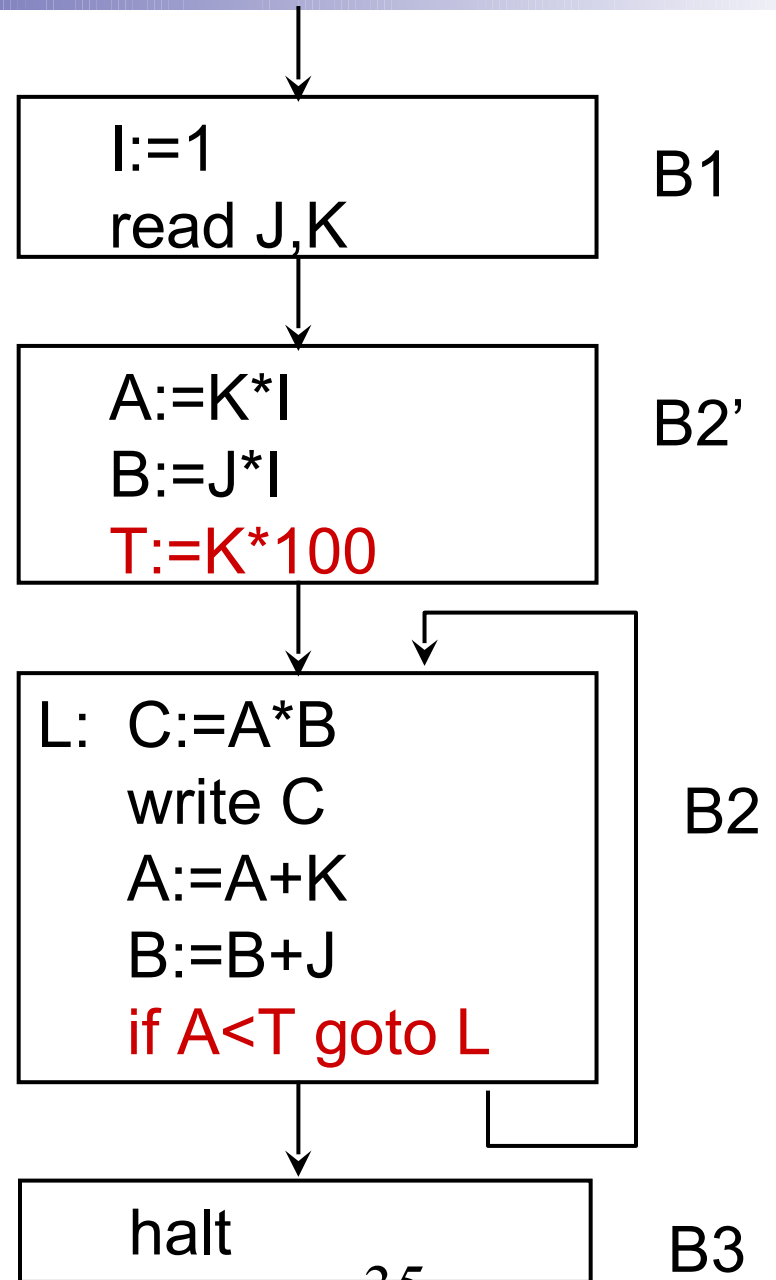
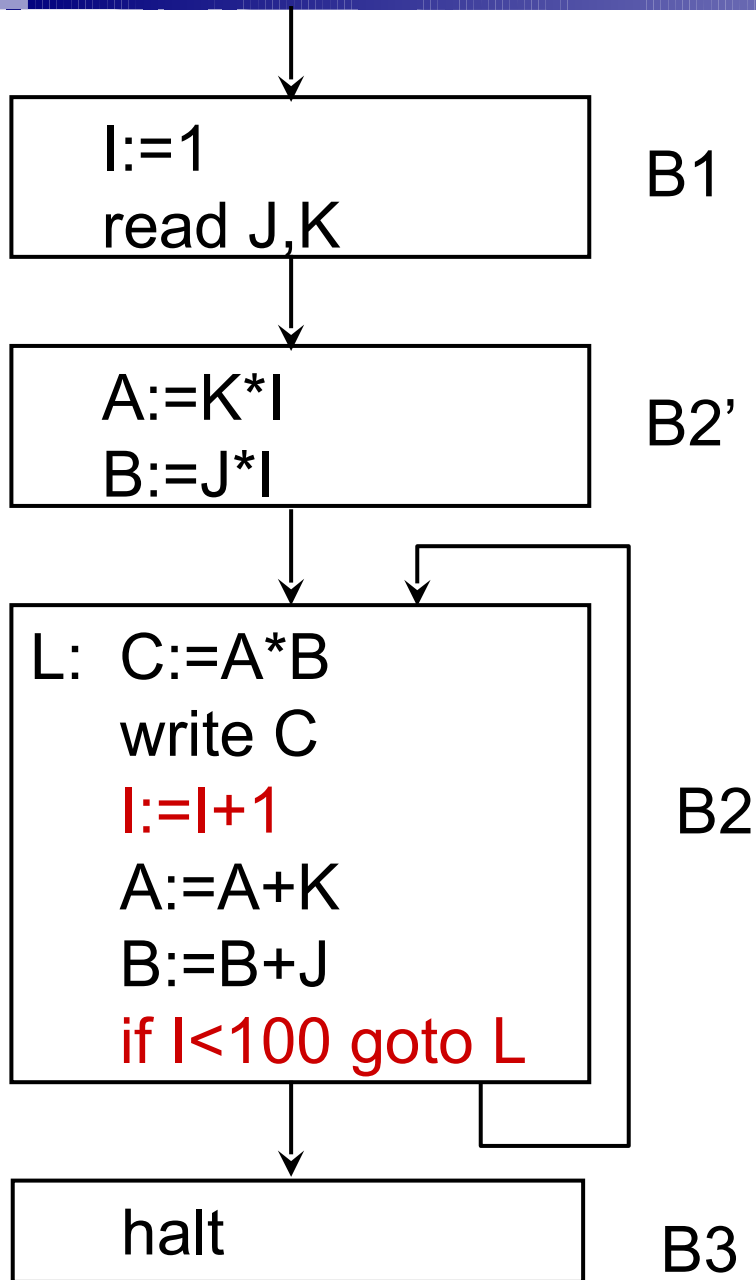
D:=A+C
 E:=A*C
 J:=D+E
 L:=15+J

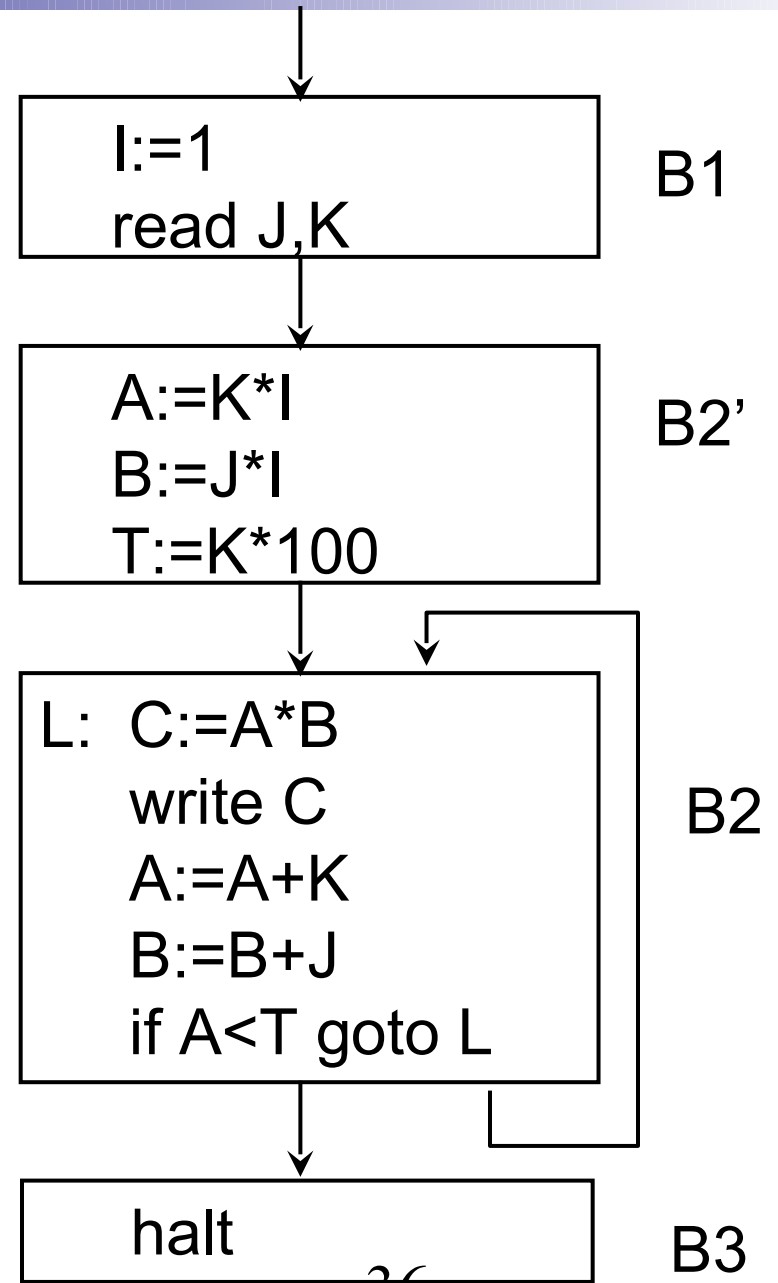
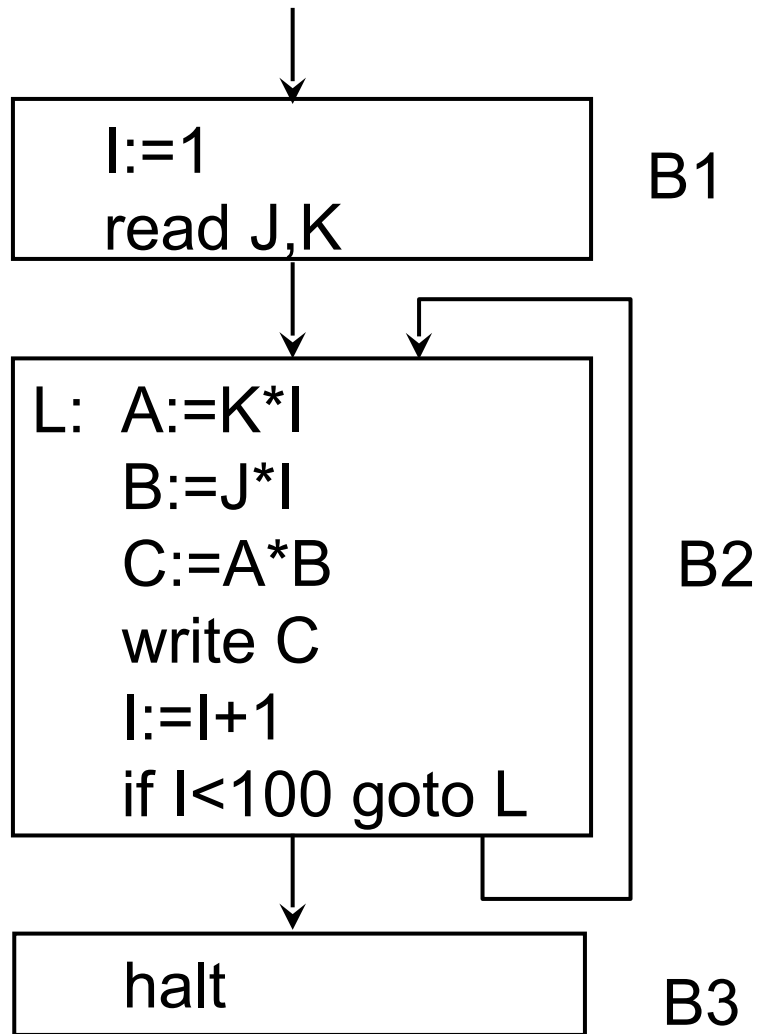
P306-4. 对以下四元式程序，对其中循环进行循环优化。

```
      I:=1
      read J, K
L:    A:=K*I
      B:=J*I
      C:=A*B
      write C
      I:=I+1
      if I<100 goto L
      halt
```



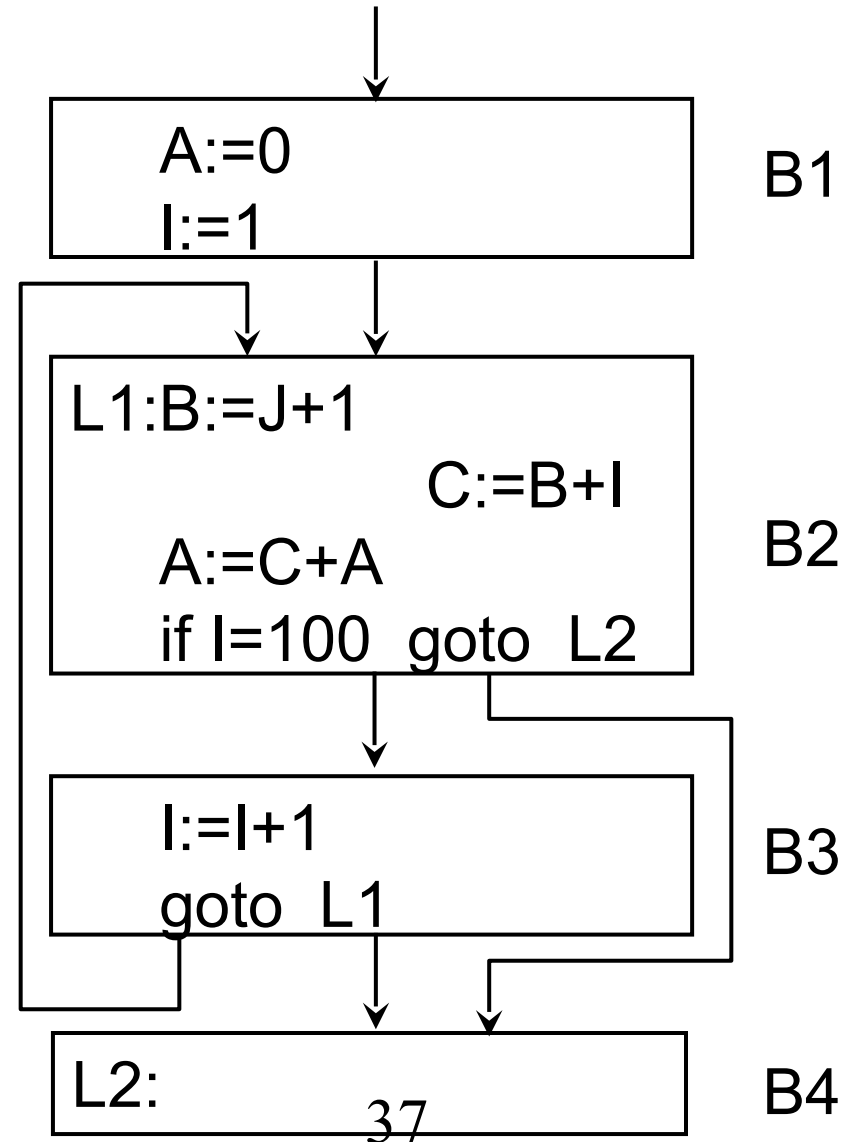


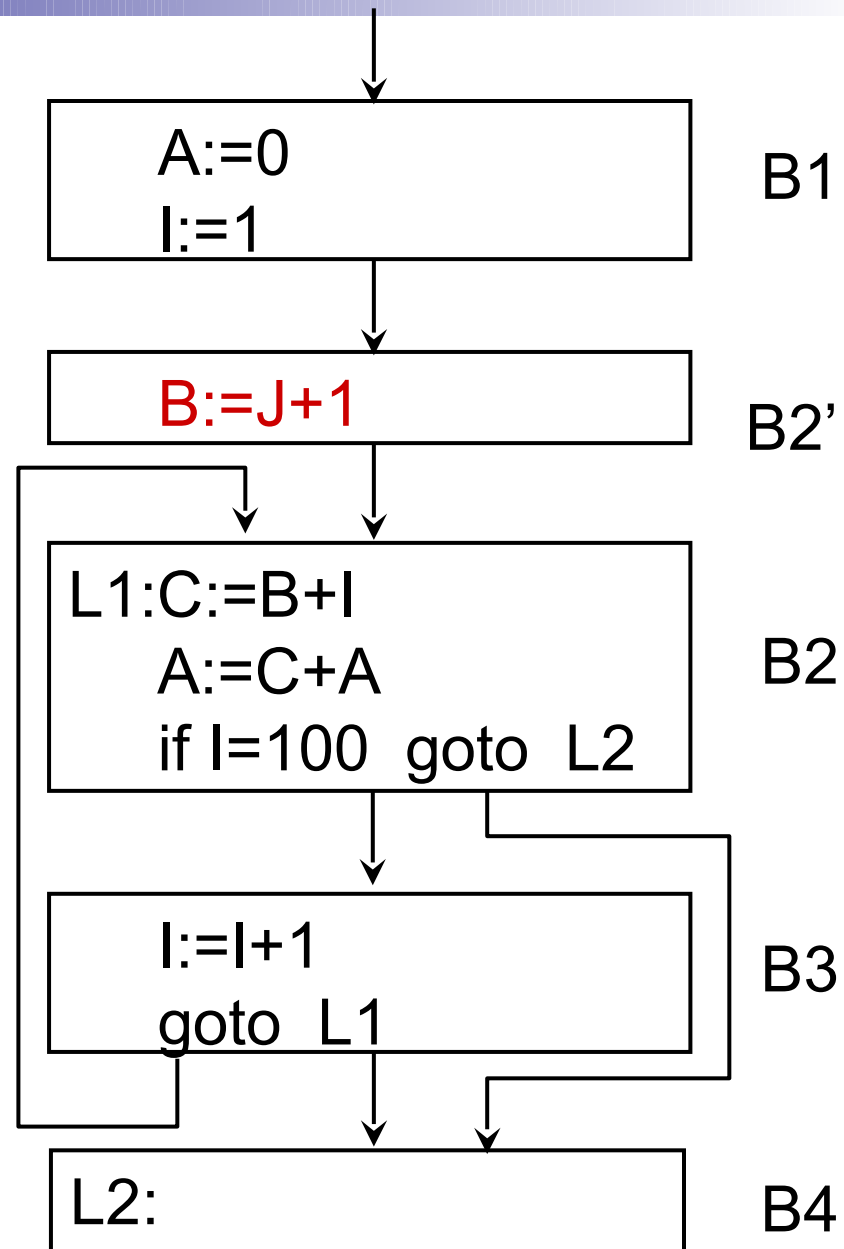
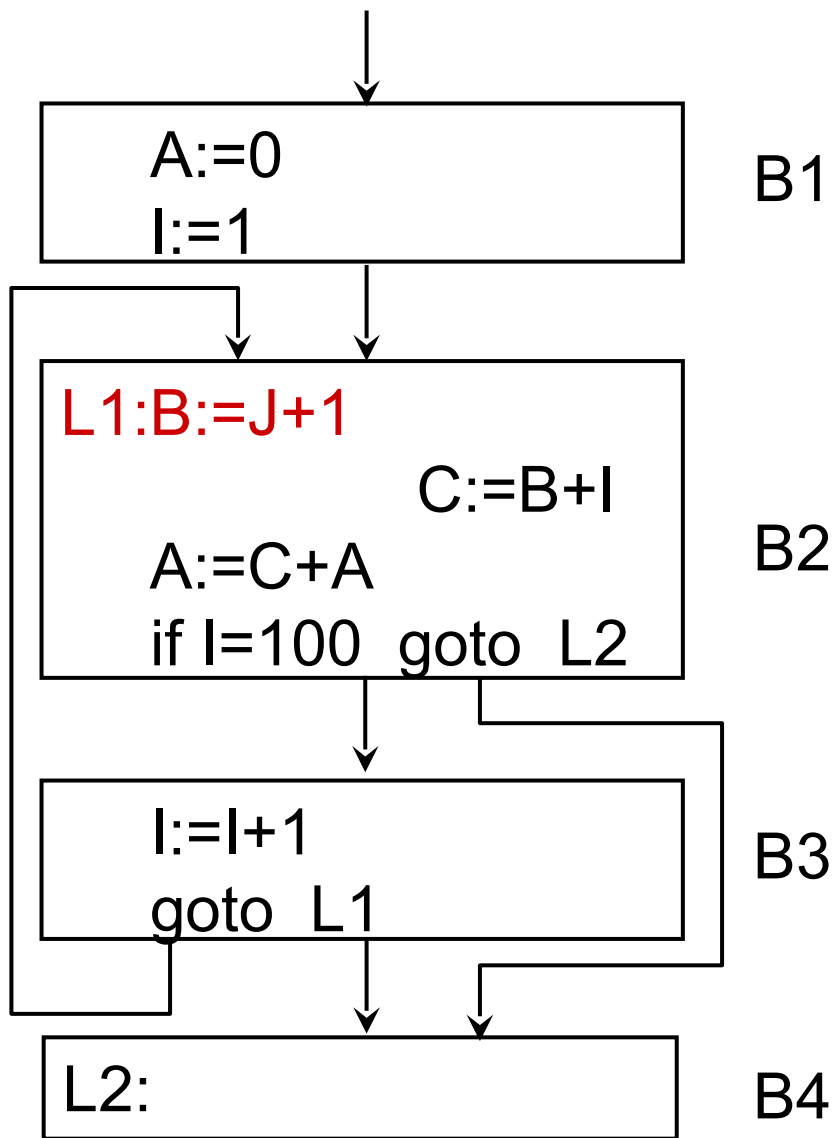


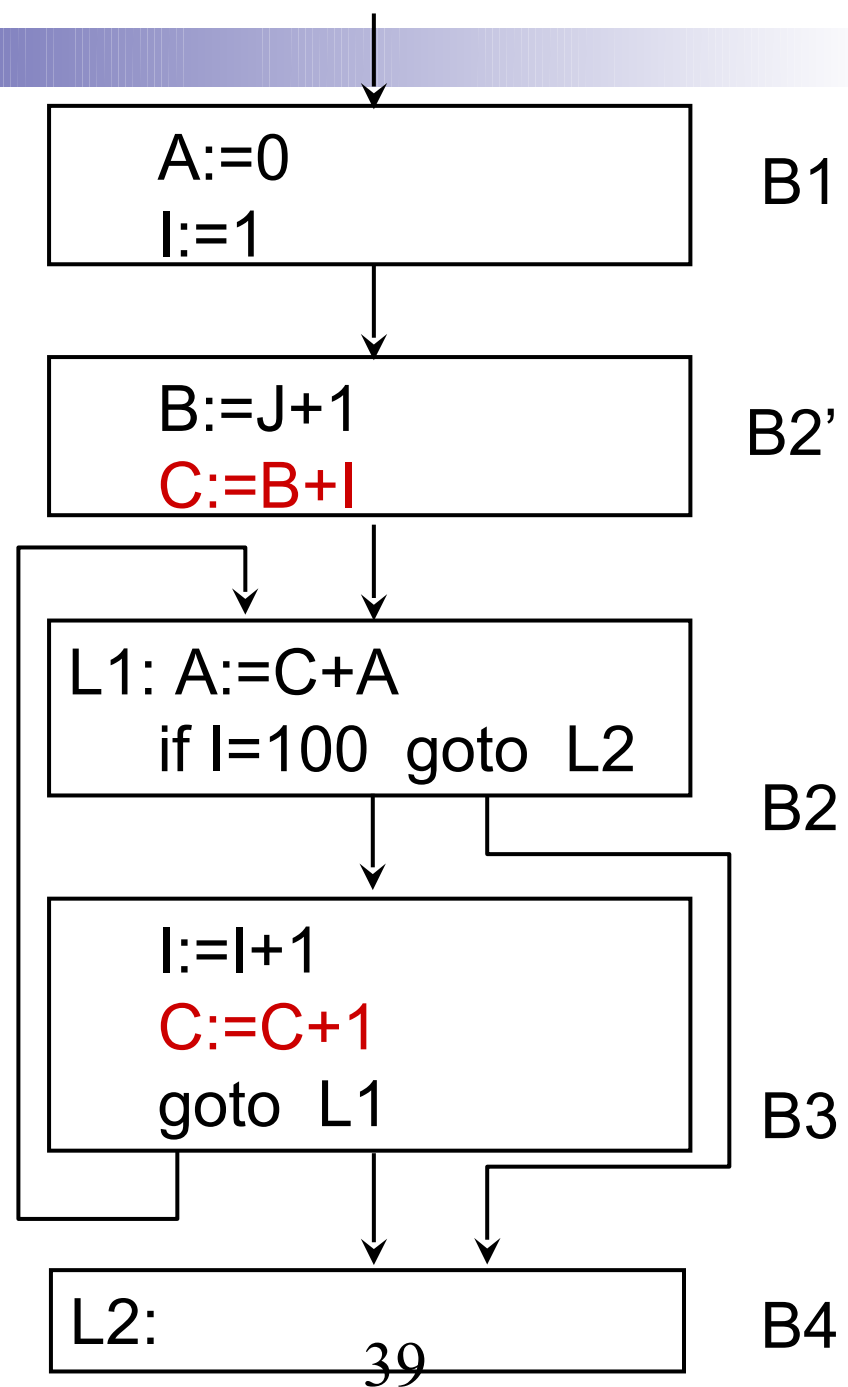
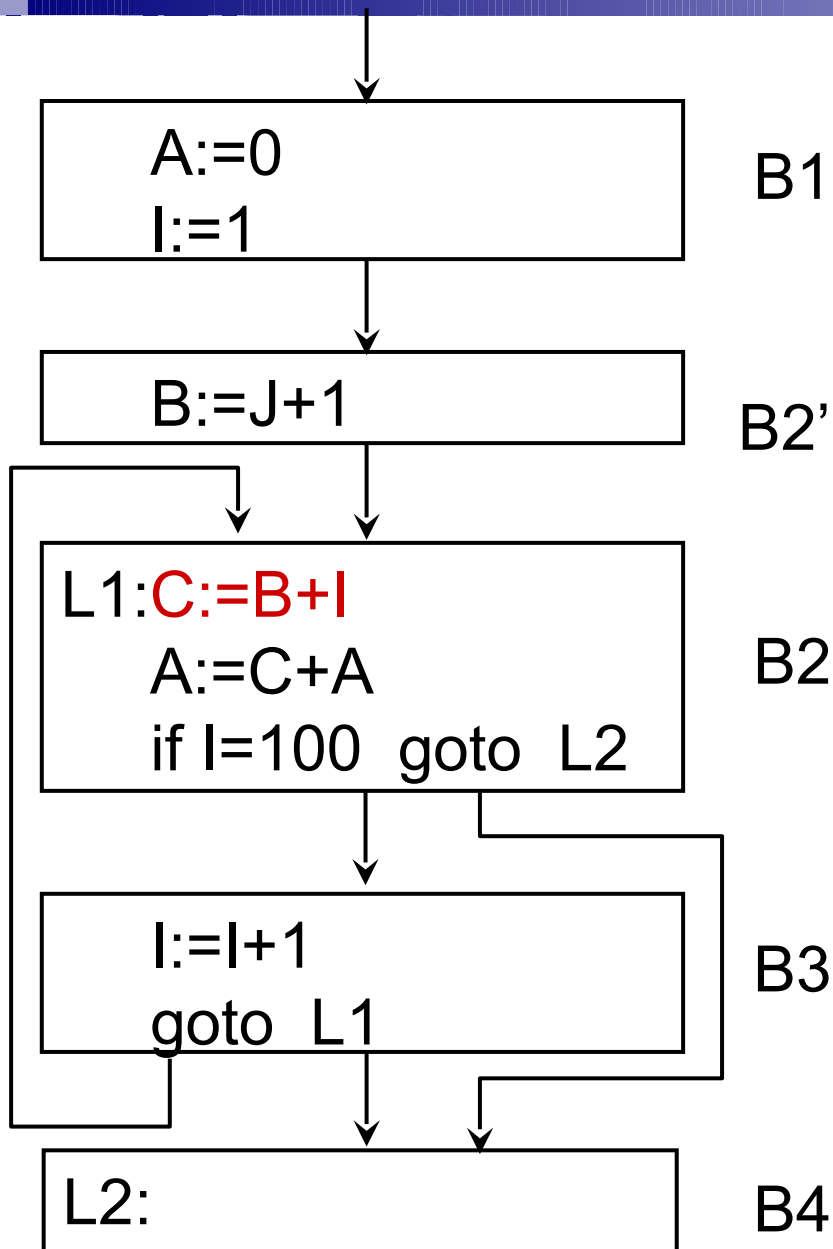


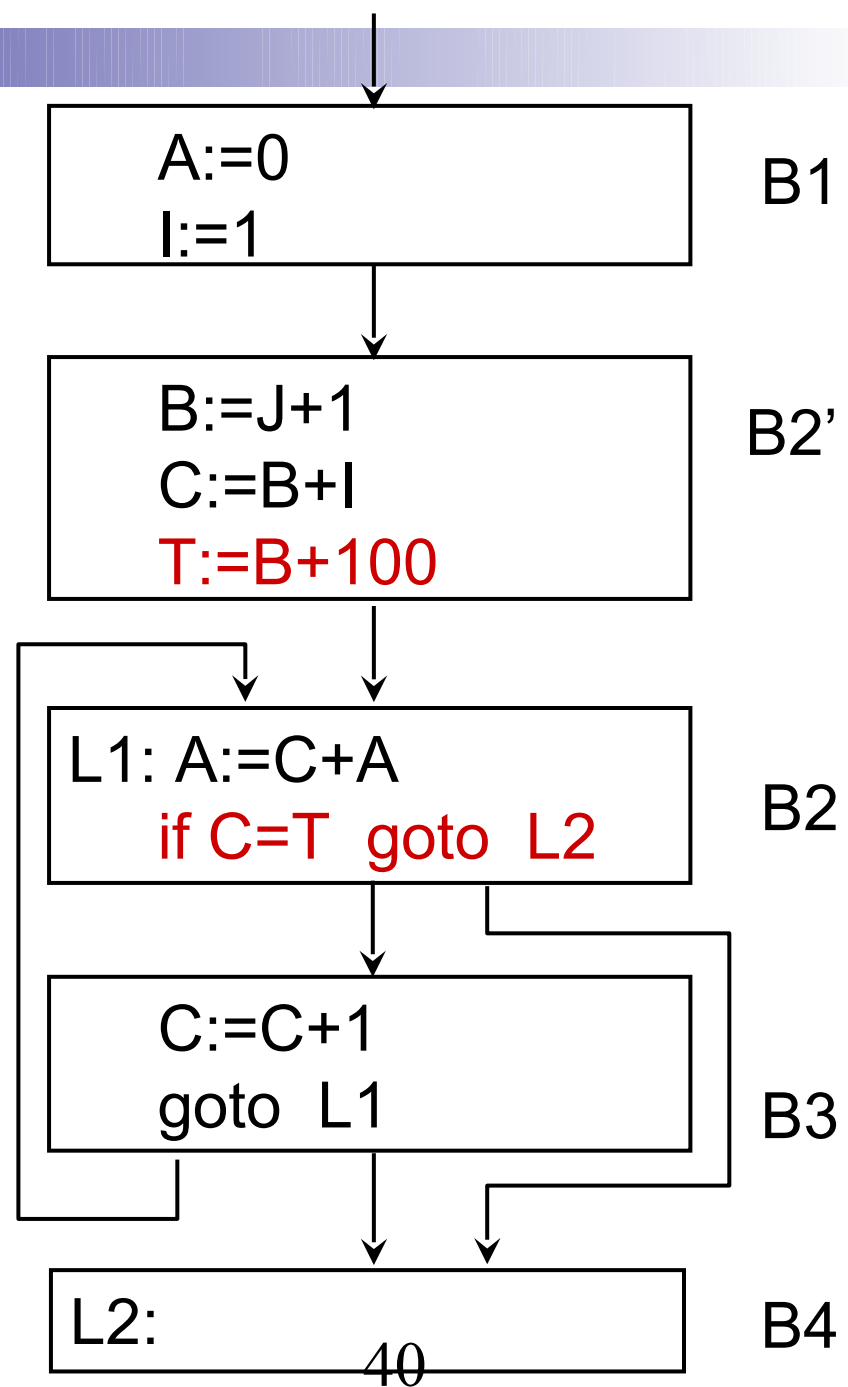
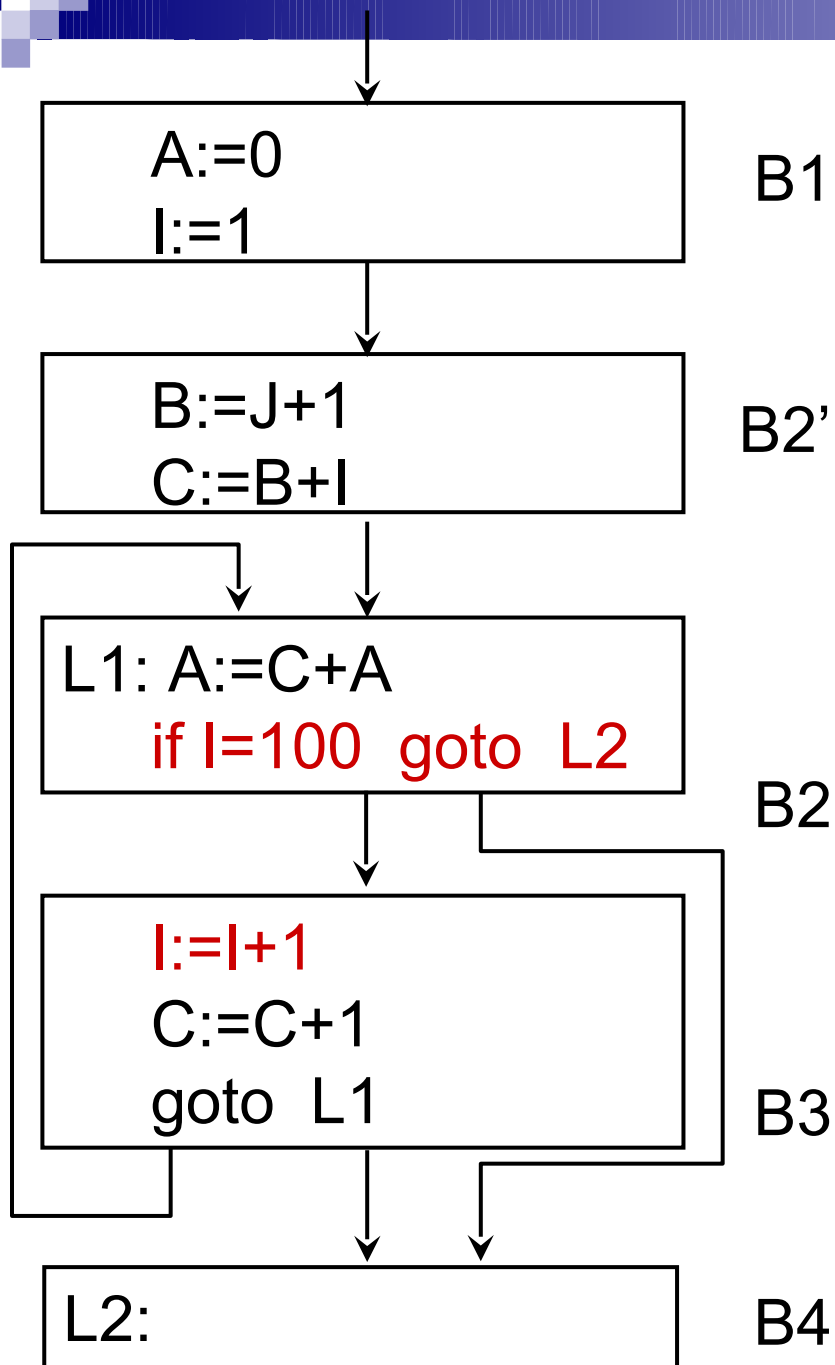
P306-5. 以下程序是某程序的最内循环，是
对它进行循环优化。

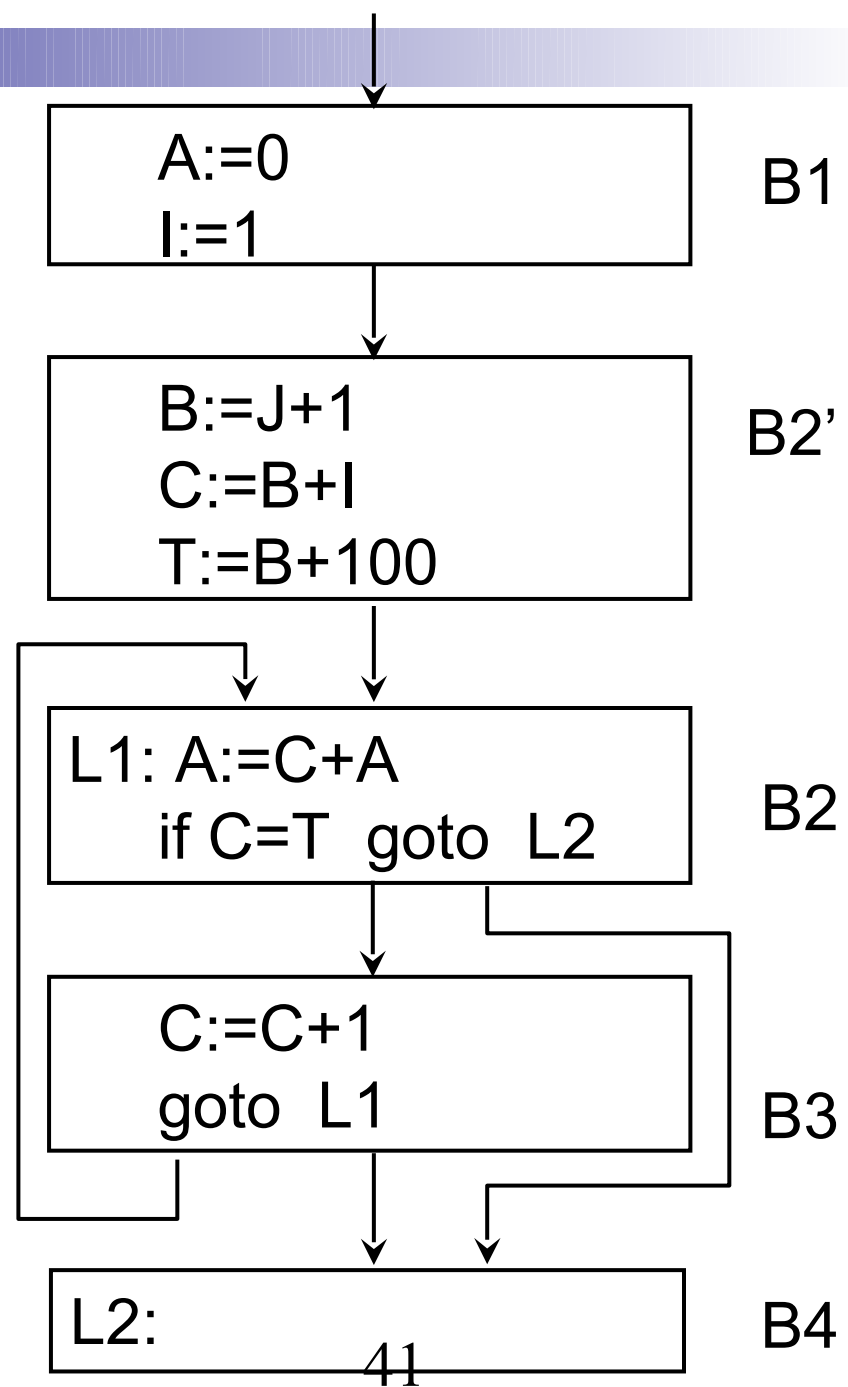
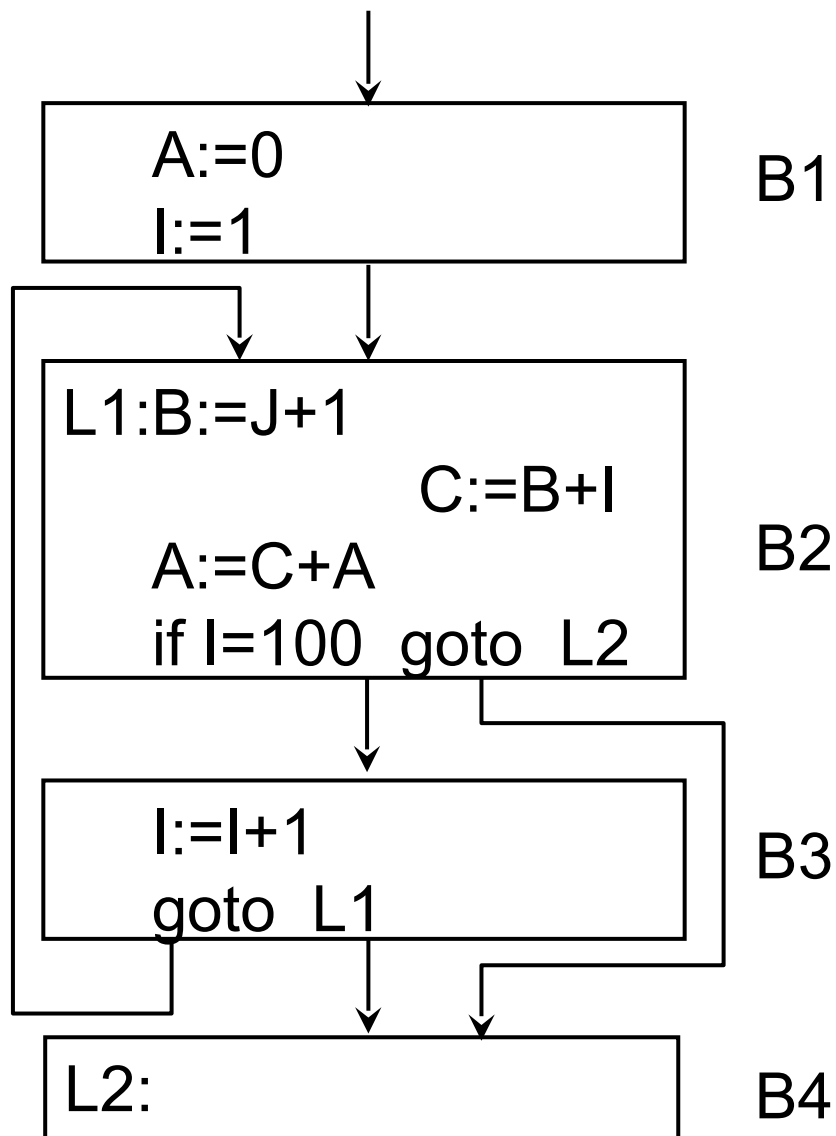
```
A:=0
I:=1
L1:  B:=J+1
      C:=B+I
      A:=C+A
      if I=100 goto L2
      I:=I+1
      goto L1
L2:
```











第十一章 代码生成

- 基本问题
- 目标机器模型
- 一个简单代码生成器

P327-1. 对以下中间代码序列 G :

$T1 := B - C$

$T2 := A * T1$

$T3 := D + 1$

$T4 := E - F$

$T5 := T3 * T4$

$W := T2 / T5$

假设可用寄存器为 R0 和 R1，W 是基本块出口的活跃变量，用简单代码生成算法生成其目标代码，同时列出代码生成过程中的寄存器描述和地址描述。

假设只有 R0 和 R1 是可用寄存器，生成的目标代码和相应的 RVALUE 和 AVALUE
UE:

中间代码	目标代码	RVALUE	AValue
T1:=B - C	LD R ₀ , B SUB R ₀ , C	R ₀ 含有 T1	T1 在 R ₀ 中
T2:=A*T1	LD R ₁ , A MUL R ₁ , R ₀	R ₀ 含有 T1 R ₁ 含有 T2	T1 在 R ₀ 中 T2 在 R ₁ 中
T3:=D + 1	LD R ₀ , D ADD R ₀ , 1	R ₀ 含有 T3 R ₁ 含有 T2	T3 在 R ₀ 中 T2 在 R ₁ 中
T4:=E-F	ST R ₁ , T2 LD R ₁ , E SUB R ₁ , F	R ₀ 含有 T3 R ₁ 含有 T4	T2 在 T2 中 T3 在 R ₀ 中 44 T4 在 R ₁ 中

假设只有 R0 和 R1 是可用寄存器，生成的目标代码和相应的 RVALUE 和 AVALUE:

中间代码	目标代码	RVALUE	AValue
T5:=T3*T4	MUL R ₀ , R ₁	R ₀ 含有 T5 R ₁ 含有 T4	T2 在 T2 中 T5 在 R ₀ 中 T4 在 R ₁ 中
W:=T2/T5	LD R ₁ , T2 DIV R ₁ , R ₀	R ₀ 含有 T5 R ₁ 含有 W	T2 在 T2 中 T5 在 R ₀ 中 W 在 R ₁ 中
	ST R ₁ , W		

小结

- 语义分析和中间代码产生
 - 翻译成四元式、构造翻译模式
 - 布尔表达式
 - 赋值语句
 - 控制语句

小结

- 运行时存储空间组织
 - 参数传递
- 优化
 - 划分基本块
 - 局部优化: DAG
 - 循环优化
- 代码生成
 - 寄存器分配

1. 假设可用的寄存器为 R0 和 R1，且所有临时单元都是有是非活跃的，试对以下四元式基本块：

```
T1:=B-C
T2:=A*T1
T3:=D+1
T4:=E-F
T5:=T3*T4
W=T2/T5
```

用简单代码生成算法生成其目标代码。

解题思路：

简单代码生成算法是依次对四元式进行翻译。不妨先考虑下面单一的四元式的翻译过程：

$T:=a+b$

由于汇编语言的加法指令代码形式为：

ADD R, X

其中 ADD 表示为加法指令，R 为第一个操作数，且第一个操作数必须为寄存器类型，X 为第二个操作数，它可以是寄存器类型，也可以是内存型的变量类型。该指令的意义为，将第一个操作数 R 与第二个操作数 X 进行相加，再将累加结果存回到第一个操作数的寄存器 R 中。所以，要完整地翻译出四元式 $T:=a+b$ ，则可能要如下的三条汇编指令：

```
LD    R, a
ADD   R, b
ST    R, T
```

上面三条指令的含义是：第一条指令是第一个操作数 b 从内存装入一个寄存器 R 中，第二条指令进行加法运算；第三条指令再将第二条指令的累加结果写回到内存中的变量 T 中。

那么，是不是一个四元翻译成目标代码时都要生成三条汇编指令呢？如果考虑到目标代码生成过程中的两优化问题，则答案是否定的。即，为了使生成的目标代码更短和充分利用计算机的寄存器，上面的三条指令中，第一条指令和第三条指令在某些情形下可能不是必需的。这是因为，如果在翻译此四元式之前，已经有指令将操作数 a 放在某个寄存器中，则第一条指令可以省略；同样，如果下一个四元式紧接着要引用操作数 T，则第三指令不必急于执行，可延迟到以后适当的时机再执行。

更进一步，如果必须要用到第一条指令（第一个操作数只在内存中而不在寄存器中）则此时所有寄存器全部分配完毕，该如何处理。此时要根据寄存器中所有变量的待用信息（也就是引用点）来决定淘汰一个寄存器给当前的四元式使用。寄存器的淘汰策略如下：

- 如果有一个寄存器中的变量已经没有后续的引用点，且该变量是非活跃的，则可直接将该寄存器作为空闲寄存器使用；
- 如果所有寄存器中变量在基本块内都还有引用点，且都是活跃的，则将引用点最远的变量所占用寄存器中的结果存回到内存相应的变量中，再将该寄存器分配给当前的指令使用。

解答：

该基本块的目标代码如下（指令后面为相应注释）：

```
LD R0, B      //取第一个空闲寄存器 R0
```

SUB R0, C //运算结束后 R0 中为 T1 结果，内存中无该结果

LD R1, A //取一个空闲寄存器 R1

MUL R1, R0 //运算结束后 R1 中为 T2 结果，内存中无该结果

LD R0, D //此时 R0 中结果 T1 已经没有引用点，且临时单元 T1 是非活跃的，所以，寄存器 R0 可作为空闲寄存器使用。

ADD R0, "1" //运算结束后 R0 中为 T3 结果，内存中无该结果

ST R1, T2 //翻译四元式 $T4:=E-F$ 时，所有寄存器已经分配完毕，寄存器 R0 中存的 T3 和寄存器 R1 存的 T2 都是有用的。由于 T2 的下一个引用点较 T3 的下一个引用点更远，所以暂时可将寄存器 R1 中的结果存回到内存的变量 T2 中，从而将寄存器 R1 空闲以备使用。

LD R1, E

SUB R1, F //运算结束后 R1 中为 T4 结果，内存中无该结果

MUL R0, R1 //运算结束后 R0 中为 T5 结果，内存中无该结果。注意，该指令将寄存器 R0 中原来的结果 T3 冲掉了。可以这么做的原因是，T3 在该指令后不再有引用点，且是非活跃变量。

LD R1, T2 //此时 R1 中结果 T4 已经没有引用点，且临时单元 T4 是非活跃的，所以，寄存器 R1 可作为空闲寄存器使用。

DIV R1, R0 //运算结束后 R1 中为 W 结果，内存中无该结果。此时所有指令部分已经翻译完毕。

ST R1, W //指令翻译完毕时，寄存器中存有最新的计算结果，必须将它们存回到内存相应的单元中去，否则，在翻译下一个基本块时，所有的寄存器被当成空闲的寄存器使用，从而造成计算结果的丢失。考虑到寄存器 R0 中的 T5 的寄存器 R1 中的 W，临时单元 T5 是非活跃的，所以，只要将结果 W 存回对应单元即可。

2. 假设可用的寄存器为 R0 和 R1，其中 T4 是活跃变量，试对以下四元式基本块：

$T1:=A+B$
 $T2:=C+D$
 $T3:=E-T2$
 $T4:=T1-T3$

用简单代码生成算法生成其目标代码。

解答：

该基本块的目标代码如下：

LD R0, A
 ADD R0, B
 LD R1, C
 ADD R1, D
 ST R0, T1
 LD R0, E
 SUB R0, R1

```
LD R1, T1
SUB    R1, R0
ST R1, T4
```


例题 11.1 假设可用的寄存器为 R0 和 R1，且所有临时单元都有是非活跃的，试对以下四元式基本块：

T1:=B-C

T2:=A*T1

T3:=D+1

T4:=E-F

T5:=T3*T4

W=T2/T5

用简单代码生成算法生成其目标代码。

解题思路：

简单代码生成算法是依次对四元式进行翻译。不妨先考虑下面单一的四元式的翻译过程：

T:=a+b

由于汇编语言的加法指令代码形式为：

ADD R, X

其中 ADD 表示为加法指令，R 为第一个操作数，且第一个操作数必须为寄存器类型，X 为第二个操作数，它可以是寄存器类型，也可以是内存型的变量类型。该指令的意义为，将第一个操作数 R 与第二个操作数 X 进行相加，再将累加结果存回到第一个操作数的寄存器 R 中。所以，要完整地翻译出四元式 T:=a+b，则可能要如下的三条汇编指令：

LD R, a

ADD R, b

ST R, T

上面三条指令的含义是：第一条指令是第一个操作数 b 从内存装入一个寄存器 R 中，第二条指令进行加法运算；第三条指令再将第二条指令的累加结果写回到内存中的变量 T 中。

那么，是不是一个四元翻译成目标代码时都要生成三条汇指令呢？如果考虑到目标代码生成过程中的两优化问题，则答案是否定的。即，为了使生成的目标代码更短和充分利用计算机的寄存器，上面的三条指令中，第一条指令和第三条指令在某些情形下可能不是必需的。这是因为，如果在翻译此四元式之前，已经有指令将操作数 a 放在某个寄存器中，则第一条指令可以省略；同样，如果下一个四元式紧接着要引用操作数 T，则第三指令不必急于执行，可延迟到以后适当的时机再执行。

更进一步，如果必须要用到第一条指令（第一个操作数只在内存中而不在寄存器中），则此时所有寄存器全部分配完毕，该如何处理。此时要根据寄存器中所有变量的待用信息（也就是引用点）来决定淘汰一个寄存器给当前的四元式使用。寄存器的淘汰策略如下：

- ① 如果有一个寄存器中的变量已经没有后续的引用点，且该变量是非活跃的，则可直接将该寄存器作为空闲寄存器使用；
- ② 如果所有寄存器中变量在基本块内都还有引用点，且都是活跃的，则将引用点最远的变量所占用寄存器中的结果存回到内存相应的变量中，再将该寄存器分配给当前的指令使用。

解答：

该基本块的目标代码如下（指令后面为相应注释）：

LD R0, B //取第一个空闲寄存器 R0

SUB R0, C //运算结束后 R0 中为 T1 结果，内存中无该结果

LD R1, A //取一个空闲寄存器 R1

MUL R1, R0 //运算结束后 R1 中为 T2 结果，内存中无该结果

LD R0, D //此时 R0 中结果 T1 已经没有引用点，且临时单元 T1 是非活跃的，所以，寄存器 R0 可作为空闲寄存器使用。

ADD R0, "1" //运算结束后 R0 中为 T3 结果，内存中无该结果

ST R1, T2 //翻译四元式 $T4 := E - F$ 时，所有寄存器已经分配完毕，寄存器 R0 中存的 T3 和寄存器 R1 存的 T2 都是有用的。由于 T2 的下一个引用点较 T3 的下一个引用点更远，所以暂时可将寄存器 R1 中的结果存回到内存的变量 T2 中，从而将寄存器 R1 空闲以备使用。

LD R1, E

SUB R1, F //运算结束后 R1 中为 T4 结果，内存中无该结果

MUL R0, R1 //运算结束后 R0 中为 T5 结果，内存中无该结果。注意，该指令将寄存器 R0 中原来的结果 T3 冲掉了。可以这么做的原因是，T3 在该指令后不再有引用点，且是非活跃变量。

LD R1, T2 //此时 R1 中结果 T4 已经没有引用点，且临时单元 T4 是非活跃的，所以，寄存器 R1 可作为空闲寄存器使用。

DIV R1, R0 //运算结束后 R1 中为 W 结果，内存中无该结果。此时所有指令部分已经翻译完毕。

ST R1, W //指令翻译完毕时，寄存器中存有最新的计算结果，必须将它们存回到内存相应的单元中去，否则，在翻译下一个基本块时，所有的寄存器被当成空闲的寄存器使用，从

而造成计算结果的丢失。考虑到寄存器 R0 中的 T5 的寄存器 R1 中的 W，临时单元 T5 是非活跃的，所以，只要将结果 W 存回对应单元即可。

代码生成是指把语法分析后或优化后的中间代码变换成目标代码。

绝对指令代码是指能够立即执行的机器语言代码，所有地址已经定位。

可重新定位指令代码是指待装配的机器语言模块，执行时，由连接装配程序把它们和某些运行程序连接起来，转换成能执行的机器语言代码。

汇编指令代码是指尚须经过汇编程序汇编，转换成可执行的机器语言代码。

如果在一个基本块内，四元式 i 对 A 定值，四元式 j 要引用 A 值，而从 i 到 j 之间没有 A 的其他定值，那么，我们称 j 是四元式 i 的变量 A 的**待用信息**。

1. 对以下中间代码序列 C:

T1:=B-C

T2:=A*T1

T3:=D+1

T4:=E-F

T5:=T3*T4

W:=T2/T5

假设可用寄存器为 R0 和 R1，W 是基本块出口的活跃变量，用简单代码生成算法生成其目标代码，同时列出代码生成过程中的寄存器描述和地址描述。

解答：

生成的目标代码、寄存器描述和地址描述如下：

中间代码	目标代码	寄存器描述	地址描述
T1:=B-C	LD R0,B SUB R0,C	R0 含 T1	T1 在 R0
T2:=A*T1	LD R1,A MUL R1,R0	R0 含 T1 R1 含 T2	T1 在 R0 T2 在 R1
T3:=D+1	LD R0,D ADD R0,#1	R1 含 T2 R0 含 T3	T2 在 R1 T3 在 R0
T4:=E-F	ST R1,T2 LD R1,E SUB R1,F	R0 含 T3 R1 含 T4	T3 在 R0 T4 在 R1 T2 在内存
T5:=T3*T4	MUL R0,R1	R1 含 T4 R0 含 T5	T4 在 R1 T2 在内存 T5 在 R0
W:=T2/T5	LD R1,T2 DIV R1,R0 ST R1, W	R0 含 T5 R1 含 W	T5 在 R0 W 在内存和 R1

2. 对以下中间代码序列：

T1:=A+B

T2:=T1-C

$T3 := D + E$

$T3 := T2 * T3$

$T4 := T1 + T3$

$T5 := T3 - E$

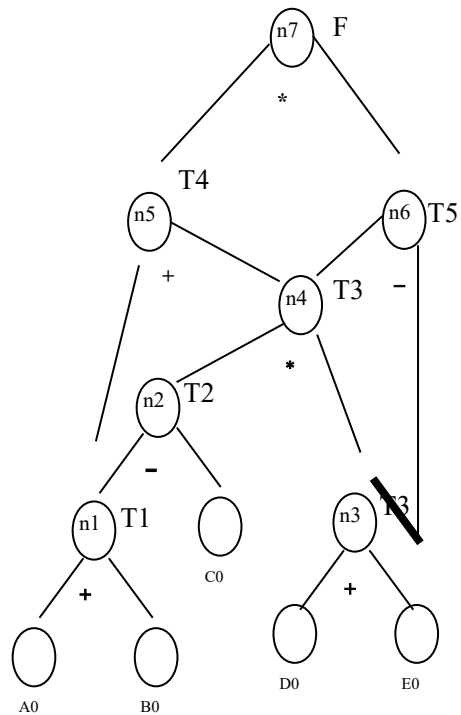
$F := T4 * T5$

(1) 应用 DAG 结点排序算法重新排序；

(2) 假设可用寄存器为 R0, F 是基本块出口处的活跃变量，应用简单代码生成算法分别生成排序前后的中间代码序列的目标代码，并比较其优劣。

解答：

(1) 设原中间代码序列为 G，则 G 的 DAG 图为：



由上可知 DAG 图有 7 个内部结点，根据结点重新排序算法，将内部结点进行排序，为：

$n3, n1, n2, n4, n6, n5, n7$

按照该顺序将中间代码进行改写得到 G'：

$T3 := D + E$

$T1 := A + B$

$T2 := T1 - C$

$T3 := T2 * T3$

T5:=T3-E

T4:=T1+T3

F:=T4*T5

(2)G 和 G'的目标代码分别为：

- | | |
|----------------|----------------|
| (1) LD R0,A | (1) LD R0,D |
| (2) ADD R0,B | (2) ADD R0,E |
| (3) ST R0,T1 | (3) ST R0,T3 |
| (4) SUB R0,C | (4) LD R0,A |
| (5) ST R0,T2 | (5) ADD R0,B |
| (6) LD R0,D | (6) ST R0,T1 |
| (7) ADD R0,E | (7) ADD R0,C |
| (8) ST R0,T3 | (8) MUL R0,T3 |
| (9) LD R0,T2 | (9) ST R0,T3 |
| (10) MUL R0,T3 | (10) SUB R0,E |
| (11) ST R0,T3 | (11) ST R0 T5 |
| (12) LD R0,T1 | (12) LD R0,T1 |
| (13) ADD R0,T3 | (13) ADD R0,T3 |
| (14) ST R0,T4 | (14) MUL R0,T5 |
| (15) LD R0,T3 | (15) ST R0,F |
| (16) SUB R0,E | |
| (17) ST R0,T5 | |
| (18) LD R0,T4 | |
| (19) MUL R0,T5 | |
| (20) ST R0,F | |

G'的目标代码比 G 的目标代码短，少 5 条指令，说明结点重排后得到代码更优化，更高效。