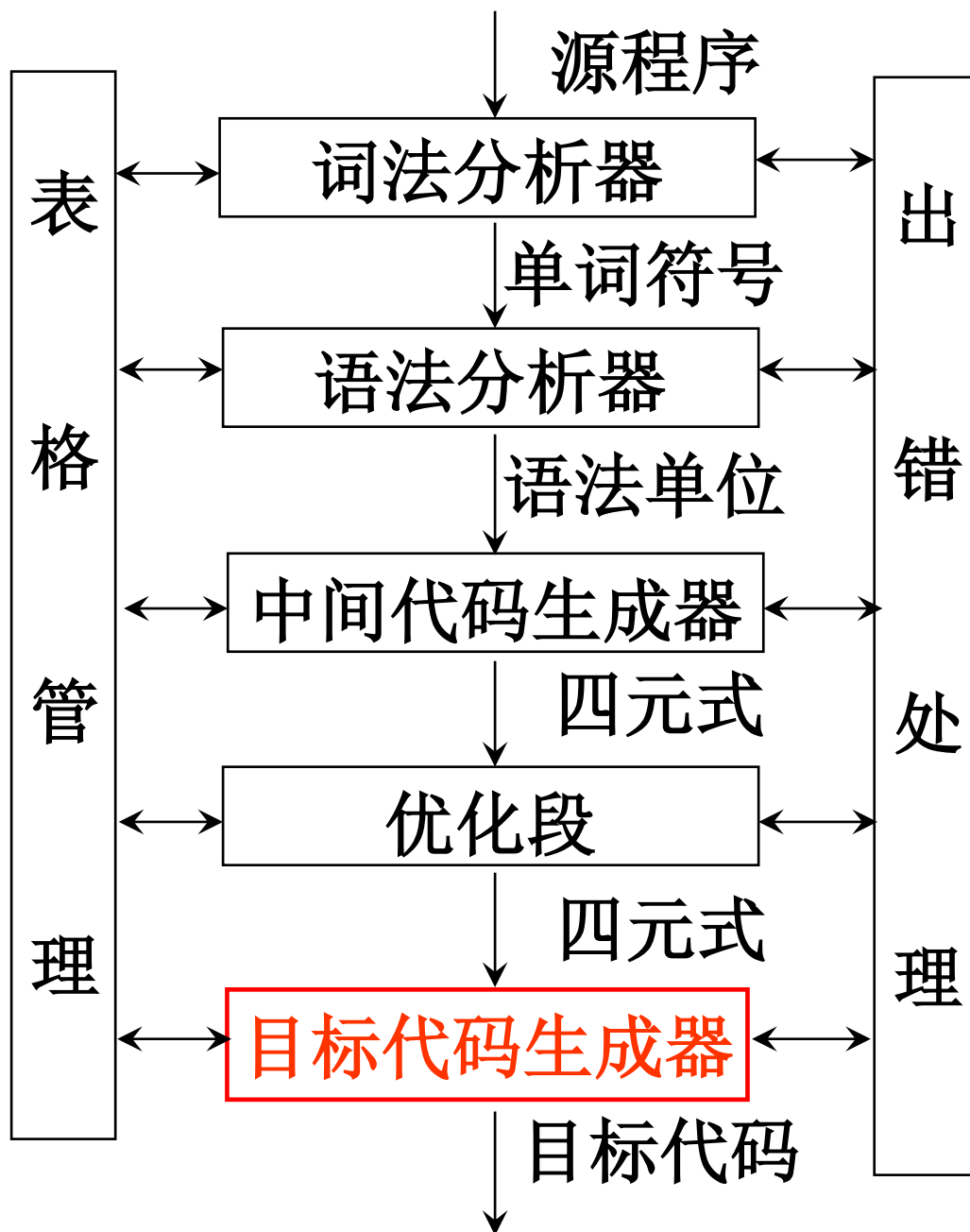




第十一章 目标代码生成

同济大学计算机系



内容线索

- **基本问题**
- **目标机器模型**
- **一个简单的代码生成器**

目标代码生成

- **代码生成**是把语法/语义分析后或优化后的中间代码变换成目标代码。
- 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。

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
 - 利用符号表信息决定数据对象运行时地址
 - 类型检查

基本问题

■ 目标代码一般有以下三种形式：

- **能够立即执行的机器语言代码**，所有地址已经定位；
- **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；
- **汇编语言代码**。尚须经过汇编程序汇编，转换成可执行的机器语言代码。

基本问题

■ 指令选择

- 一致性和完整性
- 指令速度和机器用语
- $a := a + 1$

■ INC a

■ LD R0, a /*将a放入寄存器R0*/

ADD R0, #1 /*1与R0相加*/

ST R0, a /*R0的值存入a*/

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。
- 最优寄存器指派是NP完全问题

■ 计算顺序选择

内容线索

✓ 基本问题

■ 目标机器模型

■ 一个简单的代码生成器

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	$\text{op } R_i, M$	$(R_i) \text{ op } (M) \Rightarrow R_i$
寄存器型	$\text{op } R_i, R_j$	$(R_i) \text{ op } (R_j) \Rightarrow R_i$
变址型	$\text{op } R_i, c(R_j)$	$(R_i) \text{ op } ((R_i)+c) \Rightarrow R_i$
间接型	$\text{op } R_i, *M$	$(R_i) \text{ op } ((M)) \Rightarrow R_i$
	$\text{op } R_i, *R_j$	$(R_i) \text{ op } ((R_j)) \Rightarrow R_i$
	$\text{op } R_i, *c(R_j)$	$(R_i) \text{ op } (((R_j)+c)) \Rightarrow R_i$

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



op 包括一般计算机上常见的一些运算符，如

ADD

加

SUB

减

MUL

乘

DIV

除

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	把 A 单元和 B 单元的值进行比较, 并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≤ X	如 CT=0 或 CT=1 转 X 单元
J= X	如 CT=1 转 X 单元
J≠ X	如 CT≠1 转 X 单元
J> X	如 CT=2 转 X 单元
J≥ X	如 CT=2 或 CT=1 转 X 单元

内容线索

- ✓ 基本问题
- ✓ 目标机器模型
- 一个简单的代码生成器

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

一个简单代码生成器

- 四元式的中间代码变换成目标代码，并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。

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 op C
...

j: D:=A 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

- 1 (x, x) 表示变量的待用信息和活跃信息。其中 i 表示待用信息， y 表示活跃， $^{\wedge}$ 表示非待用和非活跃；
- 2 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；
- 3 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。

■ 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”；

2. 从基本块出口到基本块入口**由后向前**依次处理各个四元式。对每一个四元式 $i: A := B \text{ op } C$ ，依次执行下面的步骤：

- 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) → ([^] , [^])

寄存器描述和地址描述

■ 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
- $AVALUE[A] = \{R1, R2, A\}$

■ 补充说明:

- 因为寄存器的分配是局限于基本块范围内的，一旦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，如果它在基本块之后是活跃的，则要把它存在寄存器中的值存放到它的主存单元中。
- 要特别强调的是，对形如： $A:=B$ 的四元式，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 。

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AVALUE[B]和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'

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

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。

■ 寄存器分配: 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 抢占用非空闲寄存器

- 1 如果B的现行值在某个寄存器 R_i 中, $RVALUE[R_i]$ 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 抢占用非空闲寄存器

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

- (1) 如果 $AVALUE[M]$ 不包含 M ，则生成 目标代码 $ST\ R_i, M$
- (2) 如果 M 是 B ，或者 M 是 C 但同时 B 也在
 $RVALUE[R_i]$ 中，则令 $AVALUE[M]=\{M, R_i\}$ ，
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:

中间代码	目标代码	RVALUE	AVALUE
$T := A - B$	LD R_0, A SUB R_0, B	R_0 含有 T	T 在 R_0 中
$U := A - C$	LD R_1, A SUB R_1, C	R_0 含有 T R_1 含有 U	T 在 R_0 中 U 在 R_1 中
$V := T + U$	ADD R_0, R_1	R_0 含有 V R_1 含有 U	V 在 R_0 中 U 在 R_1 中
$W := V + U$	ADD R_0, R_1 ST R_0, W	R_0 含有 W	W 在 R_0 中