



# 第十一章 目标代码生成

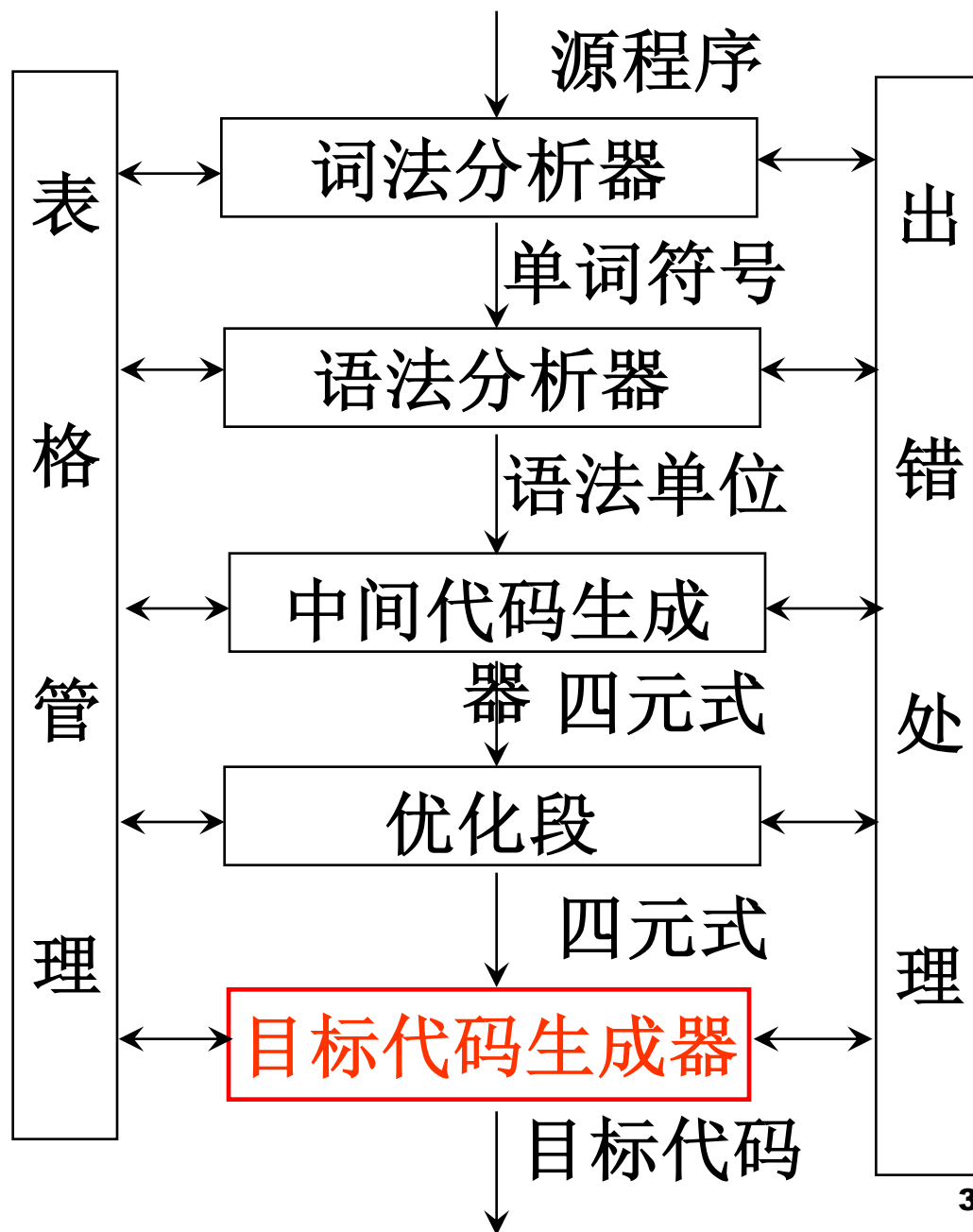
主讲人：高珍

# 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**是把语法分析后或优化后的中间代码变换成目标代码。



# 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代码，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

## ■ 代码生成器的输出

### □ 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

## ■ 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（快）。
- 如何充分利用计算机的指令系统的特点。

# 基本问题

## ■ 方法考虑一：指令选择（短）

- 一致性和完整性

- 指令速度等

- $a := a + 1$

- INC a

- LD  $R_0, a$       /\*将a放入寄存器 $R_0$ \*/

- ADD  $R_0, \#1$     /\*1与 $R_0$ 相加\*/

- ST  $R_0, a$         /\* $R_0$ 的值存入a\*/

# 基本问题

## ■ 方法考虑二：寄存器分配（快）

- 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。

## ■ 方法考虑三：计算顺序选择

- 计算顺序影响目标代码的有效性，有些计算顺序要求寄存器数量少，能够提高效率



# 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

# 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 <sub>i</sub> , M	$(R_i) \text{ op } (M) \Rightarrow R_i$
寄存器型	op R <sub>i</sub> , R <sub>j</sub>	$(R_i) \text{ op } (R_j) \Rightarrow R_i$
变址型	op R <sub>i</sub> , c(R <sub>j</sub> )	$(R_i) \text{ op } ((R_j)+c) \Rightarrow R_i$
间接型	op R <sub>i</sub> , *M	$(R_i) \text{ op } ((M)) \Rightarrow R_i$
	op R <sub>i</sub> , *R <sub>j</sub>	$(R_i) \text{ op } ((R_j)) \Rightarrow R_i$
	op R <sub>i</sub> , *c(R <sub>j</sub> )	$(R_i) \text{ op } (((R_j)+c)) \Rightarrow R_i$

如果op是一目运行符，则“op R<sub>i</sub>, M”的意义为：op (M) ⇒ R<sub>i</sub>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
<b>LD <math>R_i, B</math></b>	把 B 单元的内容取到寄存器 R, 即 $(B) \Rightarrow R_i$
<b>ST <math>R_i, B</math></b>	把寄存器 $R_i$ 的内容存到 B 单元, 即 $(R_i) \Rightarrow B$
<b>J X</b>	无条件转向 X 单元
<b>CMP A, B</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。
<b>J&lt; X</b>	如 CT=0 转 X 单元
<b>J≤ X</b>	如 CT=0 或 CT=1 转 X 单元
<b>J= X</b>	如 CT=1 转 X 单元
<b>J≠ X</b>	如 CT≠1 转 X 单元
<b>J&gt; X</b>	如 CT=2 转 X 单元
<b>J≥ X</b>	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。

- 考虑效率和充分利用寄存器，进行代码优化:

LD  $R_0, B$

ADD  $R_0, C$

MUL  $R_0, D$

ADD  $R_0, E$

ST  $R_0, A$

# 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
- $AVALUE[A] = \{R_1, R_2, A\}$



# 待用和活跃信息

## 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。

# 举例

(1)

100:  $A := B + C$   
101:  $D := B - C$

(2)

100:  $A := B + C$   
101:  $B := 1$

(3)

100:  $A := B + C$   
101:  $B := B + 1$

案例	A	B	C
(1)	( $\wedge$ , y)	(101, y)	(101, y)
(2)	( $\wedge$ , y)	( $\wedge$ , $\wedge$ )	( $\wedge$ , $\wedge$ )
(3)	( $\wedge$ , y)	(101, y)	( $\wedge$ , $\wedge$ )

## ■ 解决方法

### □ 倒着扫描

## ■ 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”。

举例

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

(i+1):  $D := A \text{ op } E$



## 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)	<b>W:=V+U</b>	( <sup>^</sup> ,y)	( <sup>^</sup> , <sup>^</sup> )	( <sup>^</sup> , <sup>^</sup> )
(3)	<b>V:=T+U</b>	(4,y)	( <sup>^</sup> , <sup>^</sup> )	(4,y)
(2)	<b>U:=A-C</b>	(3,y)	( <sup>^</sup> , <sup>^</sup> )	( <sup>^</sup> , <sup>^</sup> )
(1)	<b>T:=A-B</b>	(3,y)	(2,y)	( <sup>^</sup> , <sup>^</sup> )

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

**T:=A-B**  
**U:=A-C**  
**V:=T+U**  
**W:=V+U**

- 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”。



# 课堂测试

## ■ CANVAS

# 寄存器分配算法

# 寄存器使用原则

- 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。

# ■ 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(And), {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(此处要求B在后面是无用的变量或B是A, 目的都是避免生成Store指令以及后续还可能的Load指令。只要这个B在后续还要用到, 就不选择 $R_i$ 寄存器。)}, 则选取 $R_i$ 为所需要的寄存器R(返回);

规则一:

If (B独占 $R_i$  & B不需要刷到内存) Then return( $R_i$ )

其中, B不会再被用到的情况:

- (1) B就是A, 马上会有新的值, 不会再被用到
- (2) B的待用信息显示其不会再被用到

## ■ 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 抢占用非空闲寄存器

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 (返回) ;

规则二:

If  $\text{RVALUE}(R_i) = \{ \}$

Then return( $R_i$ )

## ■ 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

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。

规则三：

抢占 $R_i$ ，挑选标准：

- (1)  $R_i$ 的变量不需要被刷到内存，可以直接抢  
(可能有两种情况：已经在内存了，或者不会被用到了)
- (2)  $R_i$ 的变量要在最远的将来才会引用到  
(把需要的变量刷到内存)

寄存器 $R_i$ 被抢  
其上的变量如何处理？

## 为被抢的 $R_i$ 中的变量生成存数指令

假设1 语句  $A:=B \text{ op } C$

$RVALUE(R_i)=\{A\}$ , A是否需要刷到内存

答案: **不需要**。因为A马上会有新值, 且在此之前, A值不会再用到了。

假设3 语句  $A:=B \text{ op } A$

$RVALUE(R_i)=\{A,B\}$ , A是否需要刷到内存

答案: **不需要**。虽然要用A, 但是此时 $R_i$ 不会被覆盖, A仍然在 $R_i$ 中。

假设2 语句  $A:=B \text{ op } A$

$RVALUE(R_i)=\{A\}$ , A是否需要刷到内存

答案: **需要**。因为这个运算需要用到A值, 且 $R_i$ 会被B覆盖, A需要事先存储下来。

假设4 语句  $A:=B \text{ op } C$

$RVALUE(R_i)=\{B,C,D\}$ , B,C,D是否需要刷到内存

答案: **需要**。

**对 $RVALUE(R_i)$ 中每一变量M, 如果M不是A(假设4), 或者如果M是A又是C, 但不是B并且B也不在 $RVALUE(R_i)$ 中(假设2), 则**

**(1) 生成目标代码  $ST\ R_i, M$  (刷到内存)**

**例外:** 如果 $AVALUE(M)=\{M\}$ , 则不需要Store (不用刷内存)

**(2) 更改变量地址描述数组  $AVALUE(M)=\{M\}$  (脱离 $R_i$ )**

**例外:** 如果M是B,  $AVALUE(M)=\{M, R_i\}$  (如假设2, 暂时不用脱离 $R_i$ )

**例外:** 如果M是C &  $RVALUE(R_i)=\{B,C\}$ , 则 $AVALUE(M)=\{M, R_i\}$  (如假设4, 暂时不用脱离 $R_i$ )

**(3) 删除 $RVALUE(R_i)$ 中的M (更新寄存器描述数组) 返回**





# 代码生成算法

# 代码生成算法

一、四元式:  $A := B \text{ op } C$

二、四元式:  $A := B$

三、基本块结束

# 代码生成算法

## 一、对每个四元式: $i: A := B \text{ op } C$ , 依次执行(5步):

1. 申请寄存器: 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  $R_i$ , 用作存放操作数  $B$  及操作结果  $A$  的寄存器。
2. 获取  $B$  和  $C$  的存放位置: 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. 生成目标码

如果B已经在寄存器中( $B' = Ri$ )，则生成目标码：

op Ri, C'

如果B不在寄存器中( $B' \neq Ri$ )，则生成目标码：

LD Ri, B'

op Ri, C'

如果B' 或C' 为Ri，则删除AVALUE[B/C] 中的Ri。(脱离关联)

## 4. 更新变量地址描述数组和寄存器描述数组

AVALUE[A]={Ri}, RVALUE[R]={A}

## 5. 及时释放B，C的寄存器

如果: (1)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 (2)B或C占用了某寄存器 $R_k$

则：修改RVALUE( $R_k$ )及AVALUE(B/C)，释放寄存器

# 代码生成算法

## 二、对四元式 $A:=B$ 的处理

要特别强调的是，对该四元式，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 $R_i$

前提	$A:=B$
$RVALUE(R_i)=\{B\}$	$RVALUE(R_i)=\{B, A\}$
$AVALUE(B)=\{R_i\}$	$AVALUE(B)=\{R_i\}$ $AVALUE(A)=\{R_i\}$

**思考：**如果 $B$ 不在现存寄存器中呢？

**LD  $R_i, B$**

- (1) 修改寄存器描述数组 $RVALUE(R_i)$ :  $\{B, A\}$  (替换)
- (2) 变量地址描述数组 $AVALUE(B)$  :  $\{xxx, R_i\}$  (追加)

# 代码生成算法

## 三、基本块结束处的代码

### 基本块结束时把活跃变量刷到内存中

因为寄存器的分配是局限于基本块范围之内的，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M，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

**ST R<sub>i</sub>, M**

修改M的变量地址描述数组  $AVALUE(M)=\{xxx, M\}$  (追加)

## 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。

- 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

中间代码	目标代码	RVALUE	AVALUE
$T := A - B$	获取 $R_0$ LD $R_0, A$ SUB $R_0, B$	$R_0: T$	$T: R_0$
$U := A - C$	获取 $R_1$ LD $R_1, A$ SUB $R_1, C$	$R_0: T$ $R_1: U$	$T: R_0$ (暂不脱离)  $U: R_1$
$V := T + U$	获取 $R_0$ ADD $R_0, R_1$	$R_0: V$ $R_1: U$	$V: R_0$ (暂不脱离) $U: R_1$ $T: (脱离R_0关系)$
$W := V + U$	获取 $R_0$ ADD $R_0, R_1$  ST $R_0, W$	$R_0: W$ $R_1: (及时释放)$	$W: R_0, W$ $U: (及时释放)$ $V: (脱离R_0关系)$



# 课堂测试

## ■ CANVAS

Dank u

Dutch

Merci

French

Спасибо

Russian

Gracias

Spanish

شكراً

Arabic

감사합니다

Korean

תודה רבה

Hebrew

Tack så mycket

Swedish

धन्यवाद

Hindi

Obrigado

Brazilian  
Portuguese

谢谢

Chinese

***Thank You !***

Dankon

Esperanto

ありがとうございます

Japanese

Trugarez

Breton

Danke

German

Tak

Danish

Grazie

Italian

நன்றி

Tamil

děkuji

Czech

ขอบพระคุณ

Thai

go raibh maith agat

Gaelic