第 9 章　目标代码及其生成

什么是目标代码？目标代码在编译器中起到什么作用呢？

目标代码是指在机器上可以运行的代码，在本章中，可以视作汇编指令。目标代码生成，是编译的最后一个阶段，其功能可表示如下图9.1。

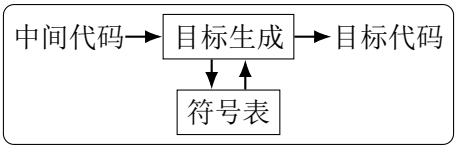

图 9.1: 目标代码功能

目标代码的来源是中间代码，也有从源程序直接生成目标代码的编译器（解释器），本书中介绍的是从中间代码进行转化的目标代码。为了简化教学内容，在生成目标代码时忽略了值单元地址，因此符号表甚少出现，但在实际生成编译器时，符号表对于生成目标代码十分重要。

其中，**中间代码**的种类是多样的，包括第 7 章中介绍的逆波兰式、三元式、四元式、语义树等；**目标代码**包括机器语言，汇编语言等，高级语言也可以作为目标代码，如：代码移植。为了便于讲解，本章中的目标代码是汇编语言的一个虚拟指令集，并不能直接运行，但原理是一致的；**符号表**包括变量的语义词典等，在生成目标代码时，会访问符号表来获取变量地址等信息。

9.1　目标代码生成的基本问题

9.1.1　目标代码选择

目标代码生成任务的目的是生成在机器上可以执行的指令。首先要考虑选择什么作为目标代码？

大多数编译程序不产生绝对地址的机器代码，而是以**汇编语言程序**作为输出，因为汇编代码与机器指令是相互对应的，以汇编语言程序作为输出可以使代码生成阶段变得容易。此外，**指令集的选择**以及**指令的执行速度问题**都是重要因素。为了使算法具有通用性，这里采用的是类似于 8086 的虚拟机及其指令系统，选择原因如下：

(1) 直接生成机器代码，难度较大，且对于不同的架构，机器代码也有所区别；

(2) 希望选择一种相对抽象但又能够理解的指令，汇编语言就具备这一属性；

(3) 不希望过于复杂，复杂可能会影响程序执行效率，即使有较好的可读性，细节上也不便于进行操作。

★ 虚拟机及其指令系统：

1. **虚拟机寄存器：** $R_0, R_1, \ldots, R_{n-1}$

   现代 CPU 的计算，严格意义上来说，都是在寄存器中完成，CPU 不能直接操作内存或外存上的数据，只能操作寄存器中的内容，因此内存或外存上的数据，只有转到寄存器中，CPU 才能对其进行操作。

2. **虚拟机指令系统：**

   (1) 指令的基本形式：op R_i, R_k/M。

   其中 op 表示操作码，可以理解为指令的一种方式，如加减乘除；M 表示变量的内存地址；R_i/R_k 表示寄存器地址。

   “op R_i, R_k”的含义是“$R_i := (R_i)$ op (R_k)”，表示 Ri 寄存器里的内容，和 R_k 寄存器里的内容，通过 op 运算，结果存在 R_i 寄存器中；

   注意：

   ①R_i 寄存器里的内容参与运算，且结果保存在 R_i 中，即 R_i 中原来的值被结果覆盖了。

   ②如果 op 为单目运算，含义是“$R_i := op\ (R_k/M)$”, 表示通过 op 运算，操作 R_k 寄存器里的内容或 M 地址指向的内容，结果存在 R_i 寄存器中。

   (2) 常用的指令：

   ①取数据、存数据：8086 中存、取指令均为 move，这里为了做区分，取数据用 LD，存数据用 ST。

   LD R_i, R_k/M ——表示从 R_k/M 取到的数据存入 R_i 中，写作 $R_i := (R_k/M)$

   ST R_i, R_k/M ——表示将 R_i 中的内容存入 R_k/M 中，写作 $R_k/M := (R_i)$

   ②转向操作：FJ 表示假跳，TJ 表示真跳，JMP 表示无条件跳转。

   FJ R_i, M ——表示判断 R_i 中内容是否为假，为假，则跳转到 M 地址指向的代码

   TJ R_i, M ——表示判断 R_i 中内容是否为真，为真，则跳转到 M 地址指向的代码

   JMP __, M ——表示无条件跳转到 M 地址指向的代码

   ③算术运算：与 8086 指令类似，包含 ADD(加)、SUB(减)、MUL(乘)、DIV(除)

   ADD $R_i, R_k/M$ ——$R_i := (R_i) + (R_k/M)$

   SUB $R_i, R_k/M$ ——$R_i := (R_i) - (R_k/M)$

   MUL $R_i, R_k/M$ ——$R_i := (R_i) * (R_k/M)$

   DIV $R_i, R_k/M$ ——$R_i := (R_i)/(R_k/M)$

   此外，还有很多操作码 op，在具体实现时，可以自行定义。

   ④逻辑运算

   LT (<), GT (>), EQ (==), LE (<=), GE (>=), NE (!=)

   AND (& &), OR (||), NO (!)

介绍完指令集，接下来介绍目标代码生成。四元式序列是不需要进一步解析的单操作，目标指令集确定时，从中间代码到目标代码，实际上是一个**模板**翻译的过程。

高级程序设计语言，即使程序非常复杂，但指令的种类是有限的，四元式也是如此，包括运算四元式、赋值四元式、跳转四元式等。

(1) **运算四元式**。若有四元式 (ω, a, b, t)，表示 a 和 b 进行 ω 操作，结果单元放到 t 中。依据前面介绍的指令集，替换成目标代码时，运算对象放到寄存器中再进行运算。生成三条目标代码如下：

① LD R, a
② $\widetilde{\omega}$ R, b
③ ST R, t

第一条表示将 a 取到寄存器 R 中；

第二条表示寄存器中内容与 b 发生 ω 运算，结果在 R 中;

第三条表示将 R 中内容保存到 t 中。对于加法，只需将 ω 替换为 ADD。显然，如果不考虑代码优化，这是一个模板替换的过程。

(2) **赋值四元式**。若有四元式 (=, a, __, b)，对应的目标代码如下。

① LD R, a
② ST R, b

因为指令集中没有内存单元存到内存单元的操作，所以①第一条先将 a 取到寄存器 R 中；②第二条将寄存器 R 的内容保存到 b 中。

例 9.1 据下列四元式翻译目标代码：

(1) $(+ \quad a \quad b \quad t_1)$
(2) $(- \quad t_1 \quad d \quad t_2)$

解法 1：若完全依据**模板**，将生成如下 6 条目标指令。

① LD R, a
② ADD R, b
③ ST R, t_1
④ LD R, t_1
⑤ SUB R, d
⑥ ST R, t_2

下一步考虑能否进行优化。这 6 条中，有两条做了无用功，第 3 条将寄存器 R 中内容保存到 t_1，紧接着第 4 条将 t_1 中的内容取到寄存器 R 中，这两条可以省去。相当于第 1 条四元式生成的目标代码中省去了模板的最后一条，第 2 条四元式生成的目标代码省去了模板中的第一条，不难发现运算四元式对应模板中第二条必不能省去，另两条可根据情况进行删减，如当一条四元式的结果单元，是下一条四元式的第一运算对象时。但不能简单地省去，否则若后续还有用到 t_1 的操作，将无法进行。如果不省 ST R, t_1，$a+b$ 的结果存入了 t_1，如果后面用到 t_1，也可以取到相应值。t_1 后续是否会被使用，决定了是否要将 t_1 存入内存单元。应用该模板，最好的

优化结果是只保留第二条，最差的结果是三条都保留。

为了精简代码，四元式结果变量值不急于存储。生成目标代码时，可以先生成前两条，第三条目标代码滞后生成。例如在生成 $(+\ a\ b\ t_1)$ 对应的目标代码时，先生成前两条，再根据后一条四元式决定是否生成最后一条四元式。

解法 2：第 (1) 个四元式表示将 a 和 b 相加，结果放在 t_1。a 和 b 是变量，符号表中记录的变量地址是内存地址，CPU 运算需要在寄存器上操作，因此需要先将运算对象 a 取到寄存器 R_0 中，从而通过 ADD 指令对 R_0 和 b 变量对应的内容进行计算，结果放在 R_0 中。

第 (2) 个四元式表示 t_1 减去 d，结果放在 t_2。经过上一步计算，t_1 存在 R_0 中，可以直接使 R_0 中的内容减去 d 变量对应内容，即 SUB R_0, d, R_0 寄存器中内容为 t_2。

① LD R_0, a
② ADD R_0, b
③ SUB R_0, d

例 9.2 根据下列四元式翻译目标代码：

(1) $(+\ \ a\ \ b\ \ t_1)$
(2) $(-\ \ c\ \ d\ \ t_2)$
(3) $(*\ \ t_1\ \ t_2\ \ t_3)$

解：依照例 9.1 解法 1 的分析，先生成 $(+\ a\ b\ t_1)$ 对应的前两条目标代码，第三条 ST 滞后生成。处理下一条四元式，两个操作数与 t_1 均不相关，一种处理方式是将 R_0 寄存器中的内容保存到 t_1 中，生成 ST 指令，然后用 R_0 寄存器做 $c-d$ 的运算，这种方式应用于单寄存器；若有多个寄存器可以选择，可以选 R_1 寄存器进行第二条四元式的运算，同样的，模板中的第三条指令滞后生成。再看第三条四元式，来决定前两条四元式的目标代码中的 ST 指令是否生成，发现第三条四元式用到了前两条四元式的结果单元，前两条四元式的 ST 指令均可省去，直接生成乘法操作的目标代码。生成目标代码如下：

① LD R_0, a
② ADD R_0, b
③ LD R_1, c
④ SUB R_1, d
⑤ MUL R_0, R_1

【讨论】在真正算法实现时，还需要解决一些问题：

(1) 为了精简代码，四元式结果变量值并不急于存储。上例中没有将 t_1、t_2、t_3 的值存入内存中，而是放在寄存器中。

(2) 例 9.1 中的 t_1 的值，系统如何知道是在寄存器 R_0 中？

(3) 例 9.2 存在寄存器分配问题，显然，若 t_2 仍然占用寄存器 R_0，则 t_1 值将被覆盖。例子中假设有多个寄存器，如果是单寄存器，需要先保存 R_0 中的内容，再进行下一步操作。

9.1.2 变量的活跃信息

为了解决上一节的问题，介绍一个重要概念——**活跃信息**，定义的是一个变量在一个基本块或一段程序内被使用的情况。具体来说需要引入变量的定义点和应用点，来判断一个变量是否活跃。

例 9.2 中的 t_1，如果在接下来还会被用到，那它就是活跃的，可能需要执行 ST 指令；如果 t_1 在接下来都不会再被用到，那它就是不活跃的，它所在的寄存器 R_0 就可以让给其他变量。

1. **变量的定义点和应用点**设有四元式：q (ω B C A)

   应用点和定义点是相对一个四元式而言的，变量 B 和 C 是操作数，在四元式 q 中被使用，则称 B 和 C 在四元式 q 处有应用点 (q)；A 是结果单元，在四元式 q 中对 A 进行赋值，即给 A 一个新的定义，则称 A 在四元式 q 处有定义点 (q)。

   利用定义点和应用点的概念，确定一个变量是否为活跃变量。

2. **活跃变量与非活跃变量 活跃变量：**一个变量从某时刻 (q) 起，到下一个定义点止，期间若有应用点，则称该变量在 q 是活跃的 (y)，否则称该变量在 q 是非活跃的 (n)。活跃和非活跃是相对某一时刻而言，脱离时刻概念，讨论活跃和非活跃也就没有意义。

   注意，我们是在一个基本块内讨论变量的活跃信息的，基本块既是优化的基本单位，也是目标代码生成的基本单位，还是求取活跃信息的基本单位。为了方便处理，假定：

   (1) 临时变量在基本块出口后是非活跃的 (n)；在基本块结束后，临时变量不会再被使用。

   (2) 非临时变量在基本块出口后是活跃的 (y)；在基本块结束后，非临时变量会再被使用。

   这样的约定，可能不是最高效的，但却是最安全的。不会将活跃变量误认为非活跃变量，而没有及时更新信息，产生错误。

   以一个赋值语句为例，中间进行算术运算时，产生的临时变量到赋值完成后都不会再被使用。下面通过具体例子说明变量的活跃信息求解过程。

   例 9.3 求下述基本块内变量的活跃信息：$x = (a + b)/(a * (a + b)); i = a + b;$ 解：令 A(I) 中的 I 为变量 A 在某点的活跃信息 (y/n)。

   第一步，写出四元式序列:

   ① (+ a b t_1)
   ② (+ a b t_2)
   ③ (* a t_2 t_3)
   ④ (/ t_1 t_3 t_4)
   ⑤ (= t_4 __ x)
   ⑥ (+ a b t_5)
   ⑦ (= t_5 __ i)

   第二步，进行 DAG 优化，得到优化后的四元式:

① $(+ \quad a \quad b \quad t_1)$
② $(* \quad a \quad t_1 \quad t_3)$
③ $(/ \quad t_1 \quad t_3 \quad x)$
④ $(= \quad t_1 \quad \_\_ \quad i)$

第三步，填写活跃信息。

对于第一个四元式中变量 a，根据定义，看 a 到下一个定义点之间是否有应用点。发现没有下一个定义点，又因为 a 是用户定义变量，在基本块出口，第四个四元式执行完后是活跃的，认为第四个四元式后还有变量 a 的定义点，且到这个定义点之间还有 a 的应用点。在第四个四元式之后使用变量 a，变量 a 是活跃的，在第一个到第四个四元式之间的变量 a 当然也是活跃的，活跃信息填 (y)。

对于第一个四元式中变量 b，虽然在第二个四元式到第四个四元式之间没有使用，但是根据前面对于变量 a 的分析，变量 b 是用户定义变量，在基本块出口时是活跃的，因此活跃信息填 (y)。

对于第一个四元式中变量 t_1，在第一个四元式中被定义，在第二个四元式中被使用，t_1 是活跃的，活跃信息填 (y)。

同理，根据变量在下一定义点前是否被使用，并结合基本块出口处的活跃信息，附有活跃信息的四元式如下。其中对于第三个四元式中变量 t_3，在第四个四元式中没有 t_3 出现，t_3 在这个基本块内不能再被使用，在第四个四元式之后，因为 t_3 是临时变量，基本块出口后是非活跃的，即 t_3 在第四个四元式之后不会再被使用，因此第三个四元式中变量 t_3 是非活跃的，活跃信息填 (n)。同理第四个四元式中变量 t_1 也是非活跃的。

(1) $(+ \quad a(y) \quad b(y) \quad t_1(y))$
(2) $(* \quad a(y) \quad t_1(y) \quad t_3(y))$
(3) $(/ \quad t_1(y) \quad t_3(n) \quad x(y))$
(4) $(= \quad t_1(n) \quad \_\_ \quad i(y))$

3. **基本块内活跃信息求解的算法** ★ 数据结构支持:

(1) 在符号表上增设一个信息项 (L) 用以记录活跃信息，结构如下。活跃信息是动态的，与时刻相关，信息项辅助填写活跃信息。

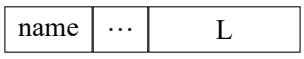

图 9.2: 记录活跃信息的信息项

(2) 四元式中变量 X 的附加信息项 X(L)，取值 L=n/y，表示不活跃/活跃；

★ 算法：

(1) 初值：基本块内各变量 SYMBL[X(L)] 分别填写：若 X 为非临时变量，则置 X(y)，否则置 X(n)。y/n 表示活跃/不活跃，目的是初始化为基本块出口的状态。

(2) **逆序**扫描基本块内各四元式（设为 q: (ω B C A)）：

执行：

① QT[q: A(L)]:= SYMBL[A(L)];，对于结果单元 A，读取符号表中变量 A 的附加信息项，作为四元式中结果单元 A 的活跃信息。

② SYMBL[A(L)]:= (n);，将符号表中变量 A 的附加信息项置为 n（非活跃）。

③ QT[q: B, C(L)]:= SYMBL[B, C(L)];，对于运算对象 B、C，同样读取符号表中对应变量的附加信息项，作为四元式中 B、C 的活跃信息。

④ SYMBL[B, C(L)]:= (y);，将符号表中 B、C 的附加信息项置为 y（活跃）。

以此类推，逆序扫描基本块内各个四元式，直到基本块的第一个四元式。虽然活跃信息的定义是一个正序的定义，但算法填写活跃信息，是逆序填写，从基本块出口状态入手，反向进行。如果出现定义点，则定义点之前活跃信息为 n，如果出现应用点，则应用点前不到定义点的时刻，活跃信息为 y。

下面通过活跃信息生成过程示例进一步说明上述算法。

例 9.4 根据基本块内四元式序列，填写活跃信息:

解：

第一步，初始化符号表中的附加信息项如图9.3。对非临时变量，初始化为 y，对临时变量初始化为 n。

基本块内下述四元式序列如下：
QT[q:]

	q:(ω B(L) C(L) A(L))
	(1)(+ a( ) b( ) t_1( ))
	(2)(- c( ) d( ) t_2( ))
	(3)(* t_1() t_2() t_3())
	(4)(- a() t_3() t_4())
	(5)(/ t_1() 2 t_5())
	(6)(+ t_4() t_5() x())

SYMBL[X(L)]

	⋯	L
a		y
b		y
c		y
d		y
t_1		n
t_2		n
t_3		n
t_4		n
t_5		n
x		y

图 9.3: 初始化附加信息项

第二步，逆序扫描四元式，填写活跃信息。

第 (6) 个四元式，对于结果单元 x，从符号表中取得 x 的附加信息项内容，作为 x 的活跃信息，并将符号表中 x 的附加信息项置为 n，这是 x 的定义点，在定义点之前且没有应用点，x 是非活跃的。类似地，填写 t_4、t_5 的活跃信息，先从符号表中取得附加信息项内容作为对应变量的活跃信息，然后将符号表中的附加信息项置为 y，t_4、t_5 被使用，则在之前也是活跃的。

第 (5) 个四元式，对于结果单元 t_5，从符号表中取得 t_5 的附加信息项内容，作为的它活跃信息，并将符号表中 t_5 的附加信息项置为 n。对于 t_1，活跃信息填从符号表中取得的附加信息项内容，并将符号表中修改为 y。

第 (4) 个四元式，对于结果单元 t_4，取符号表中的附加信息项，作为活跃信息，并将符号表中的附加信息项置为 n。对于运算对象 a 和 t_3，取符号表中的附加信息项，作为活跃信息，并将符号表中的附加信息项置为 y。

类似地，得到活跃信息结果如下图9.4。

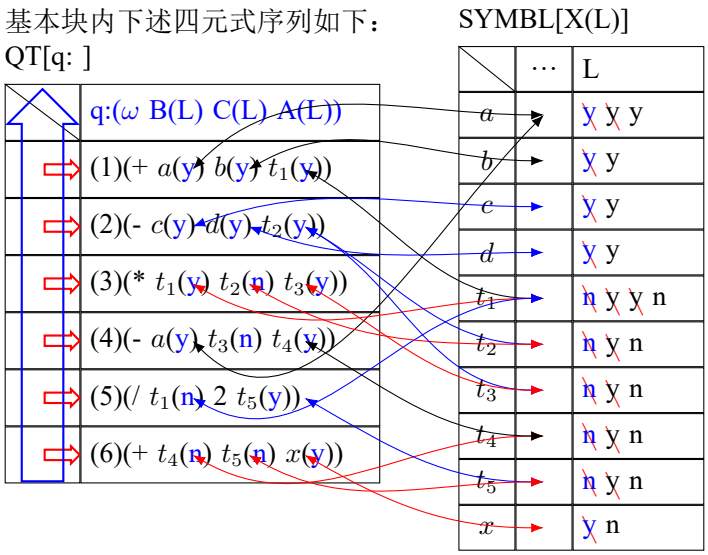

图 9.4: 活跃信息填表结果

用户定义变量大多数情况下是活跃的，但如果对同一变量两次赋值之间，没有进行使用，用户定义变量就会是非活跃的。

9.1.3 寄存器的分配问题

目标代码生成部分，目标指令的生成是根据模板进行，不是重难点，重点和难点在于优化。优化中一个很重要的内容，就是对寄存器使用的优化，即寄存器分配问题。寄存器在整个计算机体系，在编译器设计中十分重要，因为寄存器操作快且指令短，从寄存器中取数据远快于从内存或外存中取数据。

系统需要知道当前寄存器中存储的变量，或者说一个变量是否在寄存器中，在哪一个寄存器中。

1. **设置描述表 RDL(R_0, R_1, …, R_n)** 用以记录寄存器的当前状态，如 RDL.$R_1 = x$，即指明当前变量 x 值在寄存器 R_1 中。
2. **寄存器分配三原则：** 设当前四元式：q: A = B ω C，为 A 分配寄存器有以下三种情况：

   (1) **主动释放**。如果 B 已经在寄存器 R_i 中，则选择 R_i；若 ω 可交换，也可考虑 C 所在寄存器。

   ① 若 B 活跃，即在下个定义点前存在应用点，则要保存 B 的值，方法是：若有空闲寄存器 R_j，则要将 Ri 中的 B 保存到 R_j 中，生成指令 ST R_i, R_j；否则暂时存到内存中，生成指令 ST R_i, B。若 B 不活跃，就不用保存 B 的值，不生成相关指令。

   ② 修改描述表：删除 B，填写 A。

(2) **选空闲者**。从空闲寄存器中选一 R_i；并把 A 填入描述表。

(3) **强迫释放**。剥夺一 R_i：处理办法同规则 (1)。

以 (ω B C A) 为例说明上述三条分配原则。进行 ω 运算，需要先指定寄存器 R，运算结束后，寄存器 R 中的值就是结果单元 A 的值。先确定一个寄存器作为运算寄存器，即给 A 分配一个寄存器。寄存器的选择有三种情况：

(1) B 在 R_i 中，根据运算四元式生成的三条目标代码，第一条是将 B 取到寄存器中，此时 B 已经在寄存器中，这三条中的第一条即可省去，这种情况称为主动释放，B 主动将寄存器 R_i 释放给 A。如果 B 是活跃的，要先保存 B，然后再进行 ω 运算，若有其他空闲寄存器 R_j，就生成指令 ST R_i, R_j，否则生成 ST R_i, B；如果 B 不活跃，直接进行 ω 运算。计算完成后，R_i 寄存器中保存的是 A 的值，因此修改描述表 R_i = A。若对于乘法和加法，第一运算对象和第二运算对象可交换时，还可以考虑主动释放寄存器 C 给 A。

(2) 如果 B 不在寄存器中，乘法加法运算中 C 也不在寄存器中，则在空闲寄存器中选一个分配给 A，并修改描述表。

(3) 如果 B、C 都不在寄存器中，也没有空闲寄存器，就“抢”一个寄存器，处理办法同第一种情况。怎么“抢”也有多种方法：随机抢、优先抢最不活跃的等。

例 9.5 设有三个寄存器：R_0，R_1，R_2。根据给定的四元式及活跃信息，生成目标代码：

解：顺序处理四元式，根据寄存器分配三原则、活跃变量概念，写出目标代码。

设有三个寄存器：R0,R1,R2

支持：寄存器分配原则；活跃变量概念。

QT[q]	OBJ[p]	RDL R0	R1	R2
(1)(+ a(y) b(y) t_1(y))				
(2)(- c(y) d(y) t_2(y))				
(3)(* t_1(y) t_2(n) x(y))				
(4)(= a(y) _ y(n))				
(5)(/ a(n) x(n) x(y))				
(6)(- a(n) x(n) x(y))				
(7)(= x(y) _ a(y))				
(8) …				

图 9.5: 寄存器分配表

第 (1) 个四元式，给结果单元 t_1 分配寄存器，初始状态下三个寄存器均为空，选择寄存器 R_0，生成前两条目标代码为①LD R_0, a ②ADD R_0, b，模板中第三条指令滞后生成。修改描述表 RDL，此时 R_0 **保存** t_1，R_1、R_2 空闲。

第 (2) 个四元式，给结果单元 t_2 分配寄存器，运算对象 c 和 d 都不在寄存器中，根据分配三原则“选空闲者”，将空闲寄存器 R_1 分配给 t_2，生成目标代码③LD R_1, c ④SUB R_1, d，模板第

三条指令滞后生成。修改描述表 RDL，此时 R_0 保存 t_1，R_1 **保存** t_2，R_2 空闲。

第 (3) 个四元式，t_1 和 t_2 相乘，由于 t_1 在寄存器 R_0 中，根据分配三原则“主动释放”，给 x 分配寄存器 R_0，由于 t_1 是活跃的，需要保存，且有空闲寄存器 R_2，将 t_1 保存到 R_2，之后完成计算，生成目标代码⑤ST R_0, R_2 ⑥MUL R_0, R_1，模板第三条仍然滞后生成。修改描述表 RDL，此时 R_0 **保存 x**，R_1 保存 t_2，R_2 **保存** t_1。

第 (4) 个四元式，给 y 分配寄存器，a 不在寄存器中，此时也没有空闲寄存器，根据分配三原则“强迫释放”，因 t_2 不活跃，则剥夺 t_2 所在寄存器 R_1 分配给 y，因 t_2 不活跃，不必生成 ST 指令，生成目标代码⑦LD R_1, a，模板第二条滞后生成。修改描述表 RDL，此时 R_0 保存 x，R_1 **保存 y**，R_2 保存 t_1。

第 (5) 个四元式，a 除以 x 放在 x 中，尽管前一句将 a 放在 R_1，但根据寄存器描述表，R_1 寄存器的标志为 y，认为 R_1 中没有 a。又因除法运算是不可交换的，尽管 x 在寄存器中，但不能使用 x 所在寄存器。根据分配三原则“强迫释放”，因 y 不活跃，则剥夺 y 所在寄存器 R_1 分配给 x，因 y 不活跃，不生成 ST 指令，生成目标代码⑧LD R_1, a ⑨DIV R_1, R_0，模板第三条仍然滞后生成。⑦和⑧相同，后续可以进一步优化，但根据算法逻辑，会生成这样的结果。修改描述表 RDL，此时 R_0 **空闲**，R_1 **保存** x，R_2 保存 t_1。

第 (6) 个四元式，给 y 分配寄存器，运算对象 x 在寄存器 R_1 中，根据分配三原则“主动释放”，将 R_1 分配给 y，由于 x 是活跃的，需要保存，且有空闲寄存器 R_0，将 x 保存到 R_0，然后进行运算，生成目标代码⑩ST R_1, R_0 ⑪SUB R_1, t_1，模板第三条仍然滞后生成。修改描述表 RDL，此时 R_0 **保存** x，R_1 **保存** y，R_2 保存 t_1。

第 (7) 个四元式，给 a 分配寄存器，运算对象 x 在寄存器 R_0 中，根据分配三原则“主动释放”，将 R_0 分配给 a，由于 x 是活跃的，需要保存，且 R_2 寄存器中的 t_1 不活跃，将 x 保存到 R_2 中，生成目标代码⑫ST R_0, R_2，模板第二条滞后生成。修改描述表 RDL，此时 R_0 **保存** a，R_1 保存 y，R_2 **保存** x。

填表结果如下图9.6：

QT[q]	OBJ[p]	R_0	R_1	R_2
(1)(+ a(y) b(y) t_1(y))	①LD R_0,a ②ADD R_0, b	~~t_1~~		
(2)(− c(y) d(y) t_2(y))	③LD R_1, c ④SUB R_1, d		~~t_2~~	
(3)(∗ t_1(y) t_2(n) x(y))	⑤ST R_0, R_2 ⑥MUL R_0, R_1	~~x~~		t_1
(4)(= a(y) _ y(n))	⑦LD R_1, a		~~y~~	
(5)(/ a(n) x(n) x(y))	⑧LD R_1, a ⑨DIV R_1, R_0		~~x~~	
(6)(− a(n) x(n) x(y))	⑩ST R_1, R_0 ⑪SUB R_1, t_1	~~x~~	y	
(7)(= x(y) _ a(y))	⑫ST R_0, R_2	a		x
(8) …				

图 9.6: 寄存器分配结果

在寄存器分配过程中，需要注意：

(1) 一个变量在同一时刻只能占有一个寄存器；

(2) 在基本块出口时，寄存器中的活跃变量应保存其值。

另外，寄存器分配时，是先主动释放，还是先分配空闲寄存器，这是一种算法设计，主要考虑目标代码的效率，与设计的四元式有关，可以自行调整。

如果各个寄存器中的变量都活跃，则选择一个寄存器强迫释放，并将寄存器中的变量保存到内存中，后面使用该变量时，再将其读到寄存器中。这样的访存操作，也是导致程序执行慢的主要原因。

9.1.4 目标代码生成问题

目标代码生成是以**基本块**为单位的，在生成目标代码时要注意如下三个问题：

(1) 基本块开始时所有寄存器应是空闲的；结束基本块时应释放所占用的寄存器。

(2) 一个变量被定值（被赋值）时，要分配一个寄存器 R_i 保留其值，并且要填写相应的描述表 $RDL.R_i$。

(3) 为了生成高效的目标代码，生成算法中要引用寄存器分配三原则和变量的活跃信息。

定义**数据结构**如下：

- QT[q]——四元式区：存放四元式。
- OBJ[p]——目标区：存放目标代码。
- RDL(R_0, R_1, …, R_n)——寄存器状态描述表
- SEM(m)——语义栈（用于信息暂存）：主要用于跳转指令生成目标代码时，与中间代码生成时的语义栈截然不同。

注意，这里没有考虑符号表。

具体讲解单寄存器下，一些常用四元式目标代码生成过程，包括表达式、条件语句和循环语句。

例 9.6 单寄存器 (R) 下**表达式**目标代码生成：

设：R 表示寄存器；RDL 表示描述表；SEM 表示语义栈。

解：对于表达式生成目标代码，都是顺序生成，用不到 SEM 结构。根据题意，仅有一个寄存器 R，起始状态为空。主动释放、选空闲者、强迫释放三原则对于单寄存器的情况，没有很大意义，但仍沿用这一说法。

第 (1) 个四元式，给结果单元 t_1 分配寄存器 R，生成目标代码①LD R, a ②ADD R, b。修改描述表 RDL，R 保存 $a+b$ 的结果 t_1。

B	QT[q]	OBJ[p]	RDL	SEM
	(1)(+ a(y) b(y) t_1(y))			
	(2)(- c(y) d(y) t_2(y))			
	(3)(* t_1(y) t_2(n) t_3(y))			
	(4)(- a(y) t_3(n) t_4(y))			
	(5)(/ t_1(n) 2 t_5(y))			
	(6)(+ t_4(n) t_5(n) x(y))			
	…			

图 9.7: 寄存器分配表

第 (2) 个四元式，给结果单元 t_2 分配寄存器，运算对象 c 不在寄存器 R 中，由于是单寄存器，因此“强迫释放”，剥夺 t_1 所在寄存器 R 分配给 t_2，且 t_1 活跃，需要保存到内存，生成 ST 指令。然后将 c 读到 R，进行 $c-d$ 运算，生成目标代码③ST R, t_1 ④LD R, c ⑤SUB R, d。修改描述表 RDL，R 保存 $c-d$ 的结果 t_2。

第 (3) 个四元式，给结果单元 t_3 分配寄存器，运算对象 t_1 不在寄存器 R 中，乘法运算可交换，运算对象 t_2 在寄存器 R 中，进行“主动释放”，且 t_2 不活跃，不需要保存到内存，生成目标代码⑥MUL R, t_1。修改描述表 RDL，R 保存 t_1*t_2 的结果 t_3。

第 (4) 个四元式，虽然 t_3 在寄存器中，但由于减法运算不可交换，因此“强迫释放”，先读入 a 再减 t_3：将 t_3 保存到内存（减法中使用），生成 ST 指令。然后将 a 读到 R，进行 $a-t_3$ 运算，生成目标代码⑦ST R, t_3 ⑧LD R, a ⑨SUB, R, t_3。修改描述表 RDL，R 保存 $a-t_3$ 的结果 t_4。（此处 t_3 标注非活跃的原因，是在读 a 之前保护现场，而非运算完成后释放 t_3）

第 (5) 个四元式，给结果单元 t_5 分配寄存器，运算对象 t_1 不在寄存器 R 中，“强迫释放”，剥夺 t_4 所在寄存器 R 分配给 t_5，且 t_4 活跃，需要保存到内存，生成 ST 指令。然后将 t_1 读到 R，进行 t_1/2 运算，生成目标代码⑩ST R, t_4 ⑪LD R, t_1 ⑫DIV R, 2。修改描述表 RDL，R 保存 t_1/2 的结果 t_5。

第 (6) 个四元式，与 (3) 类似，给 x 分配寄存器，加法运算可交换，运算对象 t_5 在寄存器 R 中，“主动释放”，生成目标代码⑬ADD R, t_4。

填表结果如下图9.8：

表达式的目标代码是顺序生成的，比较容易理解，接下来介绍带跳转语句的目标代码生成。

例 9.7 单寄存器 (R) 下，**条件语句**目标代码生成：给定程序段如下，生成目标代码。

if$(a>b)x=(a+b)*c$; else $x=5-a*b$

解：SEM 栈用于保存待返填地址，具体实现可采用栈或队列的结构，本书中采用栈的形式进行说明。

B	QT[q]	OBJ[p]	RDL	SEM
⇨	(1)(+ a(y) b(y) t_1(y))	①LD R, a ②ADD R, b	~~t_1~~	
⇨	(2)(- c(y) d(y) t_2(y))	③ST R, t_1 ④LD R, c ⑤SUB R, d	~~t_2~~	
⇨	(3)(* t_1(y) t_2(n) t_3(y))	⑥MUL R, t_1	~~t_3~~	
⇨	(4)(- a(y) t_3(n) t_4(y))	⑦ST R, t_3 ⑧LD R, a ⑨SUB R, t_3	~~t_4~~	
⇨	(5)(/ t_1(n) 2 t_5(y))	⑩ST R, t_4 ⑪LD R, t_1 ⑫DIV R, 2	~~t_5~~	
⇨	(6)(+ t_4(n) t_5(n) x(y))	⑬ADD R, t_4	x	
	…			

图 9.8: 寄存器分配结果

第一步，生成四元式序列，划分基本块并标注活跃信息。基本块划分时，跳转到的语句，及跳转的下一条语句都是基本块的开始，并要求程序入口也是基本块的开始。在基本块入口和出口时，要保证寄存器是空闲的。

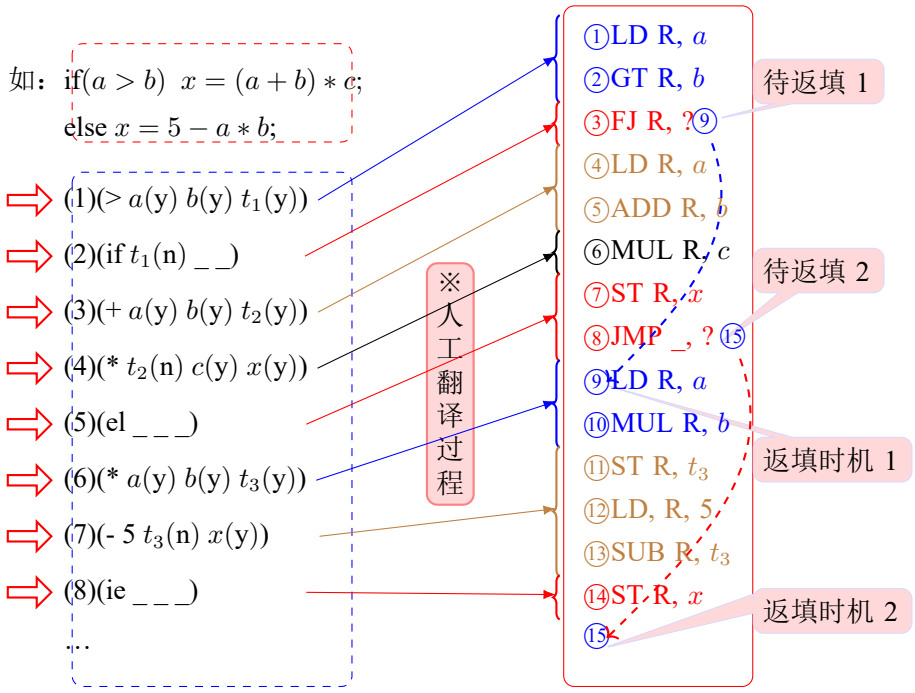

图 9.9: 例 9.7 人工翻译过程

第二步，人工翻译目标代码。根据题意，只有一个寄存器 R，初始状态为空。

第 (1) 个四元式，给 t_1 分配寄存器，生成目标代码①LD R, a ②GT R, b，GT 为 > 的操作码，表示比较 R 中内容是否大于 b。修改描述表 RDL，R 保存 t_1。

第 (2) 个四元式，if 四元式当 t_1 为假时，跳转到第 (6) 个四元式的第一条目标代码，但在生成 if 四元式的目标代码时，跳转位置未知，生成假跳目标代码③FJ R, ?，? 表示待返填。跳转位置要等第 (6) 个四元式目标代码生成后，才能填写，因此将带问号的目标代码编号③保存到 SEM

中。第 (2) 个四元式是一个基本块的出口，需要将所有寄存器内容释放，t_1 不活跃，不需要保存到内存。修改描述表 RDL，R 空闲。

第 (3) 个四元式，给 t_2 分配寄存器，将 a 读入寄存器 R，进行 $a+b$ 运算，生成目标代码④LD R, a ⑤ADD R, b。修改描述表 RDL，R 保存 t_2。

第 (4) 个四元式，运算对象 t_2 在寄存器中，乘法运算可交换，进行“主动释放”，且 t_2 不活跃，不需要保存到内存，直接进行 t_2*c 运算，生成目标代码⑥MUL R, c。修改描述表 RDL，R 保存 x。

第 (5) 个四元式，else 四元式是基本块出口，需要清空寄存器。根据约定，用户定义变量在基本块出口是活跃的，需要保存到内存，生成目标代码⑦ST R, x。同时该四元式表示无条件跳转，跳转到 ie 四元式的下一条，跳转位置未知，生成目标代码⑧JMP , ?，将带问号的目标代码编号⑧保存到 SEM 中，? 待返填。

第 (6) 个四元式，生成的第一条目标代码编号为⑨，注意第 (2) 个四元式的跳转位置为第 (6) 个四元式的第一条目标代码，所以将⑨填入第一个返填位置：从 SEM 队列中找到编码③，然后回填到编码为③的目标代码的？处。之后生成第 (6) 个四元式的目标代码⑨LD R, a ⑩MUL R, b。修改描述表 RDL，R 保存 t_3。

第 (7) 个四元式，与前一例类似，减法运算不可交换，第二运算对象 t_3 需要保存到内存供减法中使用，生成目标代码⑪ST R, t_3 ⑫LD R, 5 ⑬SUB R, t_3。修改描述表 RDL，R 保存 x。

第 (8) 个四元式，作为基本块出口，要释放寄存器，用户定义变量 x 保存到内存，生成目标代码⑭ST R, x。下一条目标代码编号为 15，填入第二个返填位置：从 SEM 队列中找到编码⑧，然后回填到编码为⑧的目标代码的? 处。

注意，要及时处理跳转地址返填。在实践中，先返填，后编写跳转指令。如生成第五个四元式对应的目标代码⑦、⑧时，先不生成⑧的指令，先将下一条目标代码的编号⑨返填，再编写跳转指令⑧。因为编写跳转指令后，要将编号压栈，栈顶不是③，是⑧，返填时就会出现问题。

填表结果如下图9.10：

上述例题中，都是通过人工的方式翻译目标代码，接下来系统介绍生成目标代码。

9.2 目标代码生成算法设计

通过两道例题总结算法。

例 9.8 给定如下基本块，写出四元式序列，进行 DAG 优化，完成单寄存器目标代码生成：

$$a = 10 + 5$$
$$x = 10 + 5 - y$$
$$y = a * x$$

解：第一步，生成四元式序列如下:

如：if($a > b$) $x = (a + b) * c$; else $x = 5 - a * b$;

※ 计算机目标代码生成过程示例：

B	QUAT[q]	OBJ[p]	RDL	SEM
→	(1)(> a(y) b(y) t_1(y))	①LD R, a ②GT R, b	~~$t1$~~	
←	(2)(if t_1(n) __ __)	③FJ R, ?⑨		~~③~~
→	(3)(+ a(y) b(y) t_2(y))	④LD R, a ⑤ADD R, b	~~t_2~~	
	(4)(* t_2(n) c(y) x(y))	⑥MUL R, c	~~x~~	
→	(5)(el __ __ __)	⑦ST R, x ⑧JMP __, ?⑮		~~⑧~~
←	(6)(* a(y) b(y) t_3(y))	⑨LD R, a ⑩MUL R, b	~~t_3~~	
	(7)(− 5 t_3(y) x(y))	⑪ST R, t_3 ⑫LD R, 5 ⑬SUB R, t_3	~~x~~	
←	(8)(ie __ __ __)	⑭ST R, x		
	…	⑮		

图 9.10: 例 9.7 填表结果

① (+, 10, 5, t_1)
② (=, t_1, __, a)
③ (+, 10, 5, t_2)
④ (−, t_2, y, t_3)
⑤ (=, t_3, __, x)
⑥ (*, a, x , t_4)
⑦ (=, t_4, __, y)

第二步，画 DAG 图如图9.11:

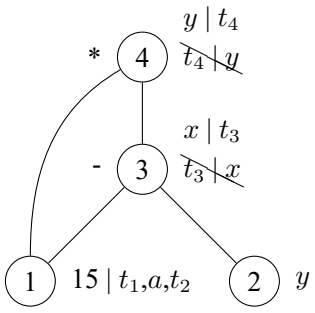

图 9.11: 例 9.8 DAG 图

第三步，得到优化后的四元式序列如下。括号中待填的是活跃信息。

① (=, 15, __, a( ))
② (__, 15, y( ), x( ))
③ (*, 15, x( ), y( ))

第四步，生成目标代码。(1) 标活跃信息：

给变量 a, x, y 标活跃信息，由于均为用户定义变量，初值均为 y，逆序填活跃信息。对最后一个四元式，填结果单元 y 的活跃信息，先从符号表主表的信息项读取当前状态 y，在符号表中返填 n；填运算对象 x 的活跃信息，先从符号表主表的信息项读取当前状态 y，在符号表中返填 y。类似地，完成活跃信息的填写，结果如下：

① (=, 15, __, a(y))
② (__, 15, y(n), x(y))
③ ($*$, 15, x(y), y(y))

a	n
x	n
y	y

(2) 目标指令生成：

第一个四元式，是赋值四元式，生成模板为：LD R, …; ST R, …给 a 分配寄存器，生成目标指令：LD R, 15 模板第二条滞后生成。

第二个四元式，是运算四元式，生成模板为 LD R, …; $\widetilde{\omega}$ R, …; ST R, …。给 x 分配寄存器，运算对象不在寄存器中，由于 a 是活跃变量，需要保存到内存，生成 ST 指令，生成目标指令 ST R, a; LD R_m 15; SUB R, y，模板第三条滞后生成。

第三个四元式，给 y 分配寄存器，寄存器中的值为第二运算对象，乘法运算可交换，模板第一条可省去，x 主动释放寄存器，由于 x 是活跃变量，需要保存到内存，生成 ST 指令，生成目标指令为 ST R, x; MUL R, 15。一个基本块结束，释放寄存器，由于 y 是活跃变量，需要保存到内存，生成目标指令 ST R, y。结果如下：

目标指令生成

① LD R, 15
② ST R, a
③ LD R, 15
④ SUB R, y
⑤ ST R, x
⑥ MUL R, 15
⑦ ST R, y

例 9.9 给定如下基本块，写出四元式序列，进行 DAG 优化，完成单寄存器目标代码生成：

$y = 1$
$x = 10 + 5 - y$
$y = a * x$
$b = a + b$

第一步，生成四元式序列如下:

① (=, 1, __, y)
② (+, 10, 5, t_1)
③ (−, t_1, y, t_2)
④ (=, t_2, __, x)
⑤ (∗, a, x, t_3)
⑥ (=, t_3, __, y)
⑦ (+, a, b, t_4)
⑧ (=, t_4, __, b)

第二步，画 DAG 图如下图9.12:

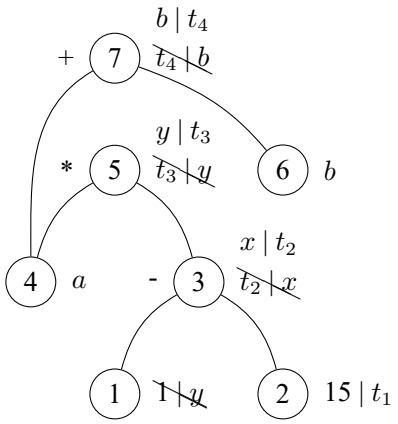

图 9.12: 例 9.9 DAG 图

第三步，得到优化后的四元式序列如下，括号中待填的是活跃信息。

① (−, 15, 1, x( ))
② (∗, a( ), x( ), y( ))
③ (+, a( ), b( ), b( ))

第四步，生成目标代码。（标活跃信息，目标指令生成）

① (−, 15, 1, x(y))
② (∗, a(y), x(y), y(y))
③ (+, a(y), b(n), b(y))

x | n
y | n
a | y
b | y

目标指令生成

① LD R, 15
② SUB R, 1
③ ST R, x
④ MUL R, a
⑤ ST R, y
⑥ LD R, a
⑦ ADD R, b
⑧ ST R, b

总结算法之前，先要明确目标代码的生成要点和生成环境。

9.2.1 目标代码生成要点和生成环境

★ 生成要点

1. **目标代码生成是在一个基本块上进行的：**

(1) 入口：寄存器空闲；

(2) 出口：释放寄存器。如9.7中编号 7 和 14 的目标代码。

2. **目标代码生成是以一个一个四元式为单位进行的：**

(1) 表达式、赋值四元式：首先对结果变量申请寄存器；然后编写目标指令；并修改描述表。

(2) 转向四元式：首先保存占用寄存器的活跃变量值（转向四元式是基本块出口）；然后再编写跳转指令；同时记住待返填地址。先保存活跃变量，还是先编写跳转指令，可以自己设计。

注意：释放寄存器时，编写存储指令，保存占有寄存器的活跃变量值，通常发生在如下两个时刻：

(1) 为结果变量申请寄存器时。强迫释放时，要先保存寄存器内活跃变量。

(2) 基本块出口时。出口需要释放寄存器。

★ 生成环境

1. **虚拟机：**单一寄存器 R；指令形式 p: (op R, M)，M 表示变量内存地址

含义：R :=(R) op (M) 或 R := op (M)

2. **表、区和栈：** QY[q]——四元式区（附有变量的活跃信息）；

OBJ[p]——目标代码区；

SEM[m]——语义栈（登记待返填的目标地址）；

SYMBL[i]——符号表；

RDL[R]——寄存器描述表；

当 RDL=0 时，表示寄存器 R 空闲；当 RDL=X 时，表示寄存器 R 被变量 X 占用。

3. **变量和函数：** CODE (op, R, M; …) ——（送代码函数）把目标代码送目标区；

BACK (p_i, p_k) ——（返填函数）把地址 p_k 返填到地址 p_i 中，p_k 表示跳转到的地址，p_i 表示待填地址。

9.2.2 表达式四元式目标代码生成算法

设当前扫描的四元式 q: (ω B C A)，是操作四元式，对应包含三条目标指令的模板。

(1) 为结果单元 A 预分配寄存器，编写目标指令:

① 当 RDL==0，则 CODE(LD R, B ; $\widetilde{\omega}$ R, C)。$\widetilde{\omega}$ 表示算符 ω 对应的操作码。

② 当 RDL==B，则

若 B(y)，则 CODE(ST R, B ; $\widetilde{\omega}$ R, C)；

否则 CODE($\widetilde{\omega}$ R, C)。

③ 当 RDL==C，且 ω 可交换，则

若 C(y)，则 CODE(ST R, C ; $\widetilde{\omega}$ R, B)；

否则 CODE($\widetilde{\omega}$ R, B)。

④ 当 RDL==D（上述三种情况之外）则

若 D(y)，则 CODE(ST R, D ; LD R, B ; $\widetilde{\omega}$ R, C)；

否则 CODE(LD R, B ; $\widetilde{\omega}$ R, C)

(2) 变量 A 登录到描述表中：RDL := A；

9.2.3 赋值四元式目标代码生成算法

设当前四元式 q: (= B __ A)

(1) 为 A 预分配寄存器，编写目标指令：

① 当 RDL==0，则 CODE(LD R, B)；此时 R 寄存器的标志是 A。

②当 RDL==B，则

若 B(y)，则 CODE(ST R, B)。

此时 B 在寄存器里，但不一定在内存里，所以需要判断 B 是否活跃。若 B 活跃，寄存器被清空时要保证 B 写回内存；若 B 不活跃，则不需要保存，直接修改描述表即可。

③ 当 RDL==D (D != B, D != A) 则

若 D(y)，则 CODE(ST R, D ; LD R, B)；

否则 CODE(LD R, B)。

(2) 变量 A 登录到描述表中：RDL := A;

9.2.4 条件语句四元式目标代码生成算法

有了表达式四元式和赋值四元式目标代码生成算法，接下来介绍条件语句和循环语句目标代码生成的算法。if 语句、while 语句等从语义上实现的是一种逻辑，对于不同条件，有不同的处理，这类逻辑在 if 语句中，主要关注 if 四元式、el 四元式和 ie 四元式。为了简化说明，假设不存在 if 嵌套，其他四元式均顺序执行，为表达式四元式或赋值四元式，前面已经介绍过目标代码生成的算法。接下来主要要解决的问题是，对于 if 四元式、el 四元式和 ie 四元式，如何生成目标指令。

回顾例9.7，if 四元式在中间代码级别语义为假跳，即当 t_1 为假时，跳转到 el 四元式的下一句。目标代码的语义与中间代码等价，if 语句的目标指令语义同样对应假跳。再看 el 四元式，中间代码语义为无条件跳，对应目标指令的无条件跳。对 ie 四元式，在中间代码级别是作为一个跳转标志，没有语义，因此没有对应的目标指令生成。

如：if$(a > b)$ $x = (a + b) * c$; else $x = 5 - a * b$;

※ 计算机目标代码生成过程示例：

B	QUAT[q]	OBJ[p]	RDL	SEM
→	(1)(> a(y) b(y) t_1(y))	①LD R, a ②GT R, b	~~t_1~~	
←	(2)(if $t_1(n)$ __ __)	③FJ R, ?⑨		③
→	(3)(+ a(y) b(y) t_2(y))	④LD R, a ⑤ADD R, b	~~t_2~~	
	(4)(∗ $t_2(n)$ c(y) x(y))	⑥MUL R, c	~~x~~	
→	(5)(el __ __ __)	⑦ST R, x ⑧JMP __, ?⑮		⑧
←	(6)(∗ a(y) b(y) t_3(y))	⑨LD R, a ⑩MUL R, b	~~t_3~~	
	(7)(− 5 t_3(y) x(y))	⑪ST R, t_3 ⑫LD R, 5 ⑬SUB R, t_3	~~x~~	
←	(8)(ie __ __ __)	⑭ST R, x		
	…	⑮		

实践中，先返填，后编跳转指令！

图 9.13: 目标代码生成示例

根据上例，总结得到条件语句四元式目标代码生成算法，SEM 采用栈结构，先返填，后编写跳转指令。

★ 条件语句的四元式结构：

设条件语句：if(E) S1; else S2;

则四元式结构如图9.14：

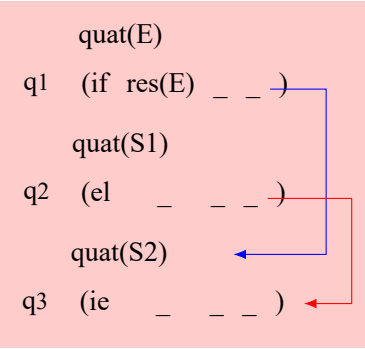

图 9.14: 四元式结构

注意：q2(el __ __ __) 的下面就是 S2 语句的入口。

★ 算法设计：

1. 设当前四元式 q: (if B __ __)

   (1) 若 RDL==0（寄存器为空），则 CODE(LD R, B ; FJ R, __); PUSH(p);

   p 为待返填的目标代码 (FJ R, __) 的编号（地址）。

   (2) 若 RDL==B，

   ① 若 B(y)，则 CODE(ST R, B ; FJ R, __)；否则 CODE(FJ R, __);

   if 语句为基本块出口，要释放寄存器，B 在寄存器 R 中，若 B 活跃，要保存到内存。

   ② PUSH(p); RDL:=0;

   (3) 若 RDL==D (D != B)

   ① 若 D(y)，则 CODE(ST R, D ; LD R, B ; FJ R, __)；否则 CODE(LD R, B ; FJ R, __);

   ② PUSH(p); RDL:=0;

2. 设当前四元式 q: (el __ __ __)

   else 语句有两个功能：①它的下一条语句是 if 语句跳转到的位置；②它本身是跳转语句，跳转到条件语句的出口位置。且 else 语句是一个基本块的出口，需要释放寄存器。

   (1) 当 RDL==X 且 X(y)，则 CODE(ST R, X ;); RDL:=0

   (2) 返填转向地址：POP(p'); BACK(p', p+2)；POP 函数表示 SEM 栈顶地址弹到 p'，BACK 函数表示把地址 p+2 填入 p' 中。

   9.7中 SEM 若使用栈结构，对于第 (5) 个四元式，未生成对应目标代码 □ 时，执行 POP(p') 函数，根据 SEM 栈顶元素 p'（编号③），找到对应的目标指令。下一步执行 BACK(p', p+2)，p' 是待返填的目标指令编号，对于条件语句，在编号为 p 的目标指令之后，还有编号为 p+1 的无条件跳转目标指令 JMP ?，跳转的目标位置在 JMP 指令之后，即编号为 p+2，因此将 p+2 返填回 p'。此处，p 为当前指令编号⑦，p+2 为⑨，即将⑨返填回③对应的目标代码。⑧目标代码还未生成，先将⑨返填回③。

   (3) 编写转向指令，CODE(JMP __, __); PUSH(p)

3. 设当前四元式 q: (ie __ __ __)

if end 语句为基本块出口，需要释放寄存器。

(1) 当 RDL==X 且 X(y)，则 CODE(ST R, X ;); RDL:=0

(2) 返填转向地址：POP(p'); BACK(p', p+1)。

9.7中，对于第 (8) 个四元式，已经生成编号 14 的目标代码，p 为编号 14，执行 POP(p')，当前 SEM 栈顶元素 p' 为⑧。下一步执行 BACK(p', p+1)，因为 (ie __ __ __) 生成目标指令中没有 JMP 指令，因此跳转目标位置的编码为 p+1 为编号 15，将编号 15 返填回⑧对应的目标代码。

9.2.5 循环语句四元式目标代码生成算法

有了条件语句四元式目标代码生成的基础，循环语句就容易理解了。

例 9.10 写出给定语句的目标代码：

while $(a > b)\ x = (a + b) * c$;

解：

第一步，写出对应四元式序列，划分基本块，并标注活跃信息。

第二步，生成目标代码。

第 (1) 个四元式，每次循环结束后，都要跳转到该位置，再进行条件判断，这个语句本身不生成相应的目标代码，但是它的下一条语句是循环的开始位置，所以要将这个语句的编号①压入 SEM 栈中，为了后面跳转到下一次循环时，找到跳转位置。

第 (2) 个四元式，与前面例题类似，生成目标代码①LD R, a ②GT R, b。

第 (3) 个四元式，do 语句判断 t_1 为假时，进行跳转，又因该四元式为基本块出口，需要释放寄存器，t_1 非活跃，不需要进行保存，生成目标指令③ FJ R, ?，跳转到第 (6) 个四元式生成的第一条目标代码，现在目标代码编号未知，将③压入 SEM 栈，? 待返填。

第 (4)、(5) 个四元式，与前面例题类似，分别生成目标代码④LD R, ? ⑤ADD R, b 和⑥MUL R, c。

第 (6) 个四元式，是基本块出口，需要释放寄存器，保存寄存器 R 中的活跃变量 x 到内存中，生成目标代码⑦ST R, x；同时也是跳转语句，跳转到循环体的开始。下一条目标代码编号为⑨，返填回 SEM 栈顶③对应的目标代码。之后生成目标代码⑧JMP __, ?，跳转到循环的开始，? 待返填。此时 SEM 栈顶为①，填⑧对应的目标代码中。

注意，在实践中，先返填，后编写跳转指令，即在生成⑧时，先将⑨返填回 SEM 栈顶元素③中，再将①填入⑧中。

如：if$(a > b)\ x = (a + b) * c;$ else$x = 5 - a * b$;

★ 算法设计

※ 计算机目标代码生成过程示例：

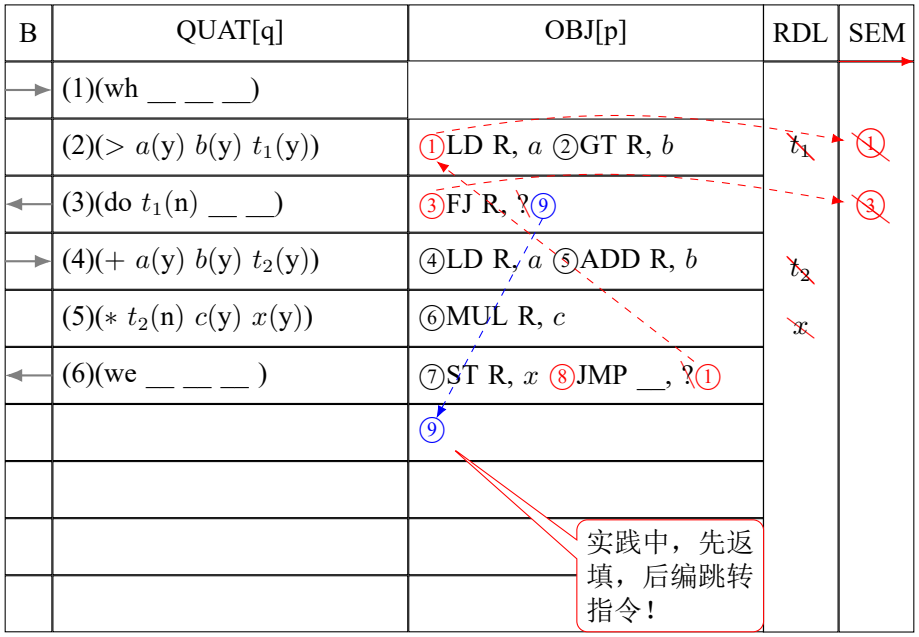

图 9.15: 目标代码生成示例

1. 设当前四元式 q: (wh __ __ __)，这是一个基本块的入口，也是一个循环的入口。

   PUSH(p); 将当前要生成的目标指令编号压栈。

2. 设当前四元式 q: (do B __ __)，do 语句判断 B 是否为假，为假则跳转。

   (1) 当 RDL==0，则 CODE(LD R, B ; FJ R, __); PUSH(p) 将指令编号或地址压入 SEM 栈，等待返填。

   (2) 当 RDL==B，则

   ① 若 B(y)，则 CODE(ST R, B ; FJ R, __)；否则 CODE(FJ R, __);

   ② PUSH(p); RDL:=0;

   在 (1) 中未改变寄存器状态，始终为空，(2) 中寄存器状态原为 B，需要释放寄存器。(3) 当 RDL==D (D != B)，则

   ① 若 D(y)，则 CODE(ST R, D ; LD R, B ; FJ R, __)；否则 CODE(LD R, B ; FJ R, __);

   ② PUSH(p); RDL:=0;

3. 设当前四元式 q: (we __ __ __)，作用 1：标记结束；作用 2：跳转到 while 开始。

   (1) 若 RDL==X 且 X(y)，则 CODE(ST R, X ;);

   (2) RDL:=0;

   (3) 返填转向地址：POP(p'); BACK(p', p+2)。

   9.10，在第 (6) 个四元式中，已经生成目标代码⑦，当前 p 为编号⑦，执行 POP(p')，根据 SEM 栈顶元素 p'（编号③），找到对应的目标指令。下一步执行 BACK(p', p+2)，p' 是待返填的目标指令编号③，在编号为 p 的目标指令之后，还有编号为 p+1 的无条件跳转目标指令 JMP ?，跳转的目标位置在 JMP 指令之后，即编号为 p+2，因此将 p+2 返填回 p'。p+2 为⑨，将⑨返填回③对应的目标代码。

(4) POP(p');

9.10中，将 SEM 栈顶地址①弹入 p'。

(5) CODE(JMP __,p');

9.10中，跳转到 p'，将①填入⑧对应的目标代码中。

9.3 一个简单代码生成器的实现

下面给出的简单代码生成器的主控程序流程图9.16。

首先进行预处理，划分基本块。接下来，对一个基本块进行处理，生成变量活跃信息，顺序遍历四元式，编写目标指令，直到到达基本块出口，释放寄存器，并跳转到取下一基本块，继续执行上述操作，直到取不到下一基本块，表示处理结束。主控程序不唯一，可以自行设计。

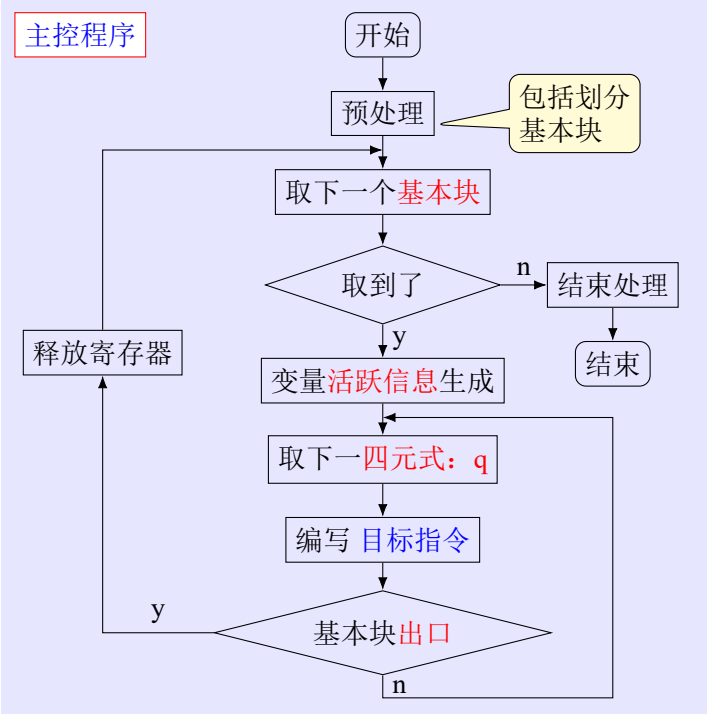

图 9.16: 主控程序流程图

下面介绍一个表达式与赋值四元式生成目标代码的例子，这些语句都是顺序执行，比较简单。

例 9.11 设有赋值语句（四元式序列）：$x = a * (a + b - d);\ \ a = (a + b)/2;\ \ y = 5;$

解：

第一步，生成四元式序列，这里只有一个基本块，标注活跃信息。

第二步，顺序遍历四元式，编写目标指令。

第 (1) 个四元式，给 t_1 分配寄存器，生成目标代码①LD R, a ②ADD R, b。

第 (2) 个四元式，给 t_2 分配寄存器，运算对象 t_1 在寄存器中，且 t_1 活跃，需要保存到内存，

生成目标代码③ST R, t_1 ④SUB R, d。

第 (3) 个四元式，乘法运算可交换，运算对象 t_2 在寄存器中，且 t_2 非活跃，直接进行计算，生成目标代码⑤MUL R, a。

第 (4) 个四元式，给 a 分配寄存器，运算对象 t_1 不在寄存器中，原本寄存器中的 x 活跃，需要保存到内存，生成目标代码⑥ST R, x ⑦LD R, t_1 ⑧DIV R, 2。

第 (5) 个四元式，给 y 分配寄存器，寄存器中的 a 活跃，需要保存到内存，生成目标代码⑨ST R, a ⑩LD R, 5。到达基本块出口，释放寄存器，且 y 活跃，需要保存到内存，生成目标代码⑪ST R, y。

第 (3) 个四元式中的 a 是非活跃的，活跃的定义是从当前到下一个定义点之间有应用点，而变量 a，第 (4) 个四元式是它的定义点，当前在第 (3) 个四元式，中间没有其他应用点，因此它是非活跃的。

填表结果如下图9.17：

B	QT[q]	OBJ[p]	RDL	SEM
	(1)(+ a(y) b(y) t_1(y))	①LD R, a ②ADD R, b	~~t_1~~	
	(2)(− t_1(y) d(y) t_2(y))	③ST R, t_1 ④SUB R, d	~~t_2~~	
	(3)(∗ a(n) t_2(n) x(y))	⑤MUL R, a	~~x~~	
	(4)(/ t_1(n) 2 a(y))	⑥ST R, x ⑦LD R, t_1	~~a~~	
		⑧DIV R, 2		
	(5)(:= 5 _ y(y))	⑨ST R, a ⑩LD R, 5	~~y~~	
	基本块出口	⑪ST R, y 释放寄存器！		

图 9.17: 例 9.11 寄存器分配填表结果

例 9.12 给定如下语句，写出四元式序列，进行 DAG 优化，完成单寄存器目标代码生成。

if $(a > b)$ $x = (a + b)/(c - d) + (a + b)$;

第一步，生成四元式序列如下：

① (>, a, b, t_1)
② (if, t_1, __, __)
③ (+, a, b, t_2)
④ (−, c, d, t_3)
⑤ (/, t_2, t_3, t_4)
⑥ (+, a, b, t_5)
⑦ (+, t_4, t_5, t_6)
⑧ (=, t_6, __, x)
⑨ (end, __, __, __)

第二步，划分基本块，对第二个基本块，画 DAG 图如下9.18:

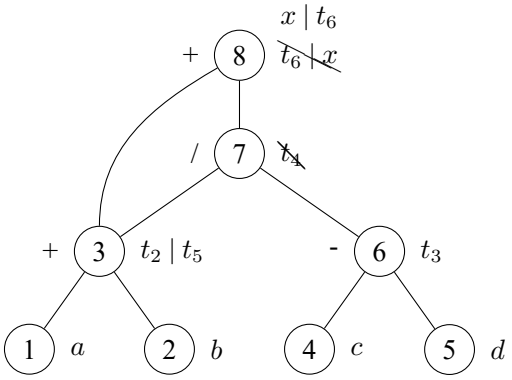

图 9.18: 例 9.12 DAG 图

第三步，写出优化后四元式如下图。括号中是活跃信息。

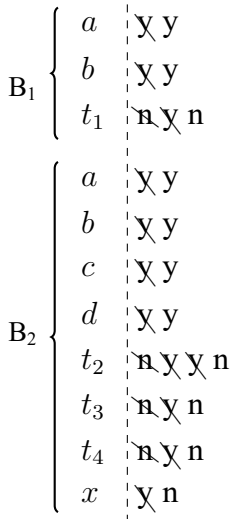

图 9.19: 例 9.12 活跃信息

① (>, a(y), b(y), t_1(y))
② (if, t_1(n), __, __)
③ (+, a(y), b(y), t_2(y))
④ (−, c(y), d(y), t_3(y))
⑤ (/, t_2(y), t_3(n), t_4(y))
⑥ (+, a(n), b(n), t_5(y))
⑦ (end, __, __, __)

第四步，生成目标指令。

目标指令

① LD R, a
② GT R, b
③ FJ R, ?
④ LD R, a
⑤ ADD R, b
⑥ ST R, t_2
⑦ LD R, c
⑧ SUB R, d
⑨ ST R, t_3
⑩ LD R, t_2
⑪ IV R, t_3
⑫ ADD R, t_2
⑬ ST R, x