第8章 代 码 生 成

本章要点

- 中间代码和用于代码生成的数据结构
- 基本的代码生成技术
- 数据结构引用的代码生成
- 控制语句和逻辑表达式的代码生成
- 过程和函数调用的代码生成
- 商用编译器中的代码生成：两个案例研究
- TM：简单的目标机器
- TINY语言的代码生成器
- 代码优化技术考察
- TINY代码生成器的简单优化

在这一章中，我们着手编译器的最后工作——用来生成目标机器的可执行代码，这个可执行代码是源代码语义的忠实体现。代码生成是编译器最复杂的阶段，因为它不仅依赖于源语言的特征，而且还依赖于目标结构、运行时环境的结构和运行在目标机器的操作系统的细节信息。通过收集源程序进一步的信息，并通过定制生成代码以便利用目标机器，如寄存器、寻址模式、管道和高速缓存的特殊性质，代码生成通常也涉及到了一些优化或改善的尝试。

由于代码生成较复杂，所以编译器一般将这一阶段分成几个涉及不同中间数据结构的步骤，其中包括了某种称做中间代码(itermediate code)的抽象代码。编译器也可能没有生成真正的可执行代码，而是生成了某种形式的汇编代码，这必须由汇编器、链接器和装入器进行进一步处理。汇编器、链接器和装入器可由操作系统提供或由编译器自带。在这一章中，我们仅仅集中关注于中间代码和汇编代码的生成，这两者之间有很多共同特性。我们不考虑汇编代码到可执行代码的更进一步的处理，汇编语言或系统的编程文本可以更充分地处理它。

本章的第1节考虑中间代码的两种普遍形式，三地址码和P-代码，并且讨论它们的一些属性。第2节描述生成中间代码或汇编代码的基本算法。接下来的章节讨论针对不同语言特性的代码生成技术，这包括了表达式、赋值语句、控制语句(如if语句，`while`语句)以及过程和函数调用。

之后的一节将应用在前面章节中学到的技术开发TINY语言的一个汇编代码生成器。由于在这种细节水平上的代码生成需要实际的目标机器，因此首先讨论一个目标结构和机器模拟器TM。附录C提供了源代码清单。然后，我们再描述完整的TINY语言的代码生成器。最后给出一个关于标准代码的改善、优化技术的简介，同时描述了怎样将一些简单的技术融入到TINY代码生成器之中。

8.1 中间代码和用于代码生成的数据结构

在翻译期间，中间表示(intermediate representation)或IR代表了源程序的数据结构。迄今为止，本文使用了抽象语法树作为主要的IR。除IR外，翻译期间的主要数据结构是符号表，这在第6章中已学过了。

虽然抽象语法树是源代码完美充分的表述，即使对于代码生成也不过这样(这一点我们将在后面的章节中看到)，但是它与目标代码极不相像，在控制流构造的描述上尤为如此。在控制流构造上，目标代码(如机器代码或汇编代码)使用转移语句而不是if和`while`语句。因此，

编译器编写者可能希望从语法树生成一个更接近目标代码的中间表示形式，或者用这样一个中间表示代替语法树，然后再从这个新的中间表示生成目标代码。这种类似目标代码的中间表示称为中间代码(intermediate code)。

中间代码能采用很多形式，几乎有多少种编译器就有多少种中间代码形式。然而所有中间代码都代表了语法树的某种线性化(linearization)形式，也就是说，语法树用顺序形式表示。中间代码可以是高水平的，它几乎和语法树一样可以抽象地表示各种操作。它或者还可以非常接近目标代码。它可以使用或不使用目标机器和运行时环境的细节信息，如数据类型的尺寸、变量的地址和寄存器。它可以混合或不混合符号表中包括的信息，如作用域、嵌套层数和变量的偏移量。假如它混合了符号表中包括的信息，目标代码的生成基于中间代码就足够了；否则，编译器必须保留符号表。

当编译器的目标是产生非常高效的代码时，中间代码是极其有用的。如要产生高效的代码就需要相当数量的目标代码属性分析，使用中间代码能使这变得容易。特别地，虽然从语法树中直接得到混合细节分析信息的附加数据结构不是不可能的，但它能更容易地从中间代码中得到。

中间代码在使编译器更容易重定向上也是有用的：假如中间代码与目标机器相对独立，那么要为不同目标机器生成代码就仅需重写从中间代码到目标代码的翻译器。这比重写整个编译器要容易。

在本节中我们将学习两个中间代码的普遍形式：三地址码 (three-address code)和P-代码(P-code)。这两种中间代码以许多不同的形式出现，我们的研究将仅集中在普遍特性上，而不是代表了某一个版本的细节化描述。这种描述能在本章最后“注意与参考”节所描述的文献中找到。

8.1.1 三地址码

三地址码最基本的用法说明被设计成表示算术表达式的求值，形式如下：

```
x = y op z
```

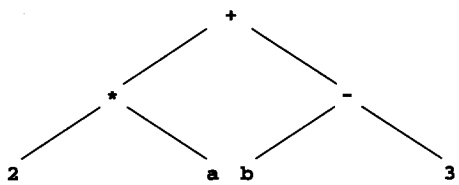
这个用法说明表示了对**y**和**z**的值的应用操作符*op*，并将值赋给**x**。这里的*op*可以是算术运算符，如+或-，也可以是其他能操作于**y**、**z**值的操作符。

三地址码这个名字来自于这个用法说明的形式，因为 **x**、**y**、**z**通常代表了内存中的3个地址。但是要注意，**x**的地址的使用不同于**y**、**z**的地址的使用。**y**、**z**(**x**不能)可以代表常量或没有运行时地址的字面常量。

为了看清这种形式的三地址码如何能表示表达式的计算，考虑下边的算术表达式

```
2*a + ( b-3 )
```

语法树如下：



相应的三地址码如下

```
t1 = 2 * a
```

```
t2 = b - 3
t3 = t1 + t2
```

三地址码要求编译器产生临时变量名，在这个例子中的是 **t1**、**t2**和**t3**。这些临时变量对应于语法树的内部节点而表示计算值，在这个例子中用临时变量 **t3**表示根节点值⊖。这里并没有说明如何在内存中分配这些临时变量；它们通常将被分到寄存器中，但也有可能保存在活动记录里面(参见第7章“临时栈”的讨论)。

三地址码仅代表了从左至右的语法树线性化，因为首先列出了相对于根的左子树的求值的代码，编译器在某种情况下希望用另一种顺序也是有可能的。我们注意到对于三地址码来说，是有可能使用另一顺序，即(临时变量有不同的意思)：

```
t1 = b-3
t2 = 2*a
t3 = t2+t1
```

很明显，上面所示的这种三地址码形式对于表示所有语言，即使是最小的程序语言的特性也是不够的。例如一元操作符(如负号)就需要一个三地址码的变种(包含两个地址)如：

```
t2 = -t1
```

为适应标准程序语言的使用结构，必须为每个结构改变三地址码的形式。如果语言中含有不常见到的特性，那么就必须为表达的这种特性发明另一种三地址码形式。这就是三地址码之所以没有标准形式的原因(正如语法树没有标准形式一样)。

在这一章余下的章节中我们将逐个处理一些程序语言共有的结构，并显示怎样将这些结构翻译成三地址码。为了知道其结果，我们给出一个用 TINY语言编写的完整示例。

请考虑来自第1章1.7节的TINY例子，该例计算了一个整数的阶乘，我们把它重新放在程序清单8-1中。

程序清单8-1 TINY程序示例

```
{ Sample program
  in TINY language--
  computes factorial
}
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

程序清单8-2是这个例子的三地址码。这个代码包含了许多三地址码的不同形式。首先，内置的输入和输出操作符 **read**和**write**已被直接翻译成一地址指令。其次，这里有一个条件转移指令**if_false**，它通常被用来翻译 if语句和循环语句，它包含两个地址：被检测的条

⊖ 诸如**t1**、**t2**的名字仅仅意味着是这种代码通常类型的代表。实际上，如果像这里一样使用源代码名字的话，三地址码中的临时变量名必须区别于在实际的源代码中所用的名字。

件值和转移的代码地址。一地址的 **label**指令指示了这个转移地址的位置。有了用以实现三地址码的数据结构，这些 **label**指令可能并不是必需的。 **halt**指令(无地址)用来标志代码的结束。

程序清单8-2　程序清单8-1中TINY程序的三地址码

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

最后，我们注意到原代码中的赋值语句导致了如下形式的 **copy**指令

```
x=y
```

例如，例程中语句

```
fact:=fact*x;
```

翻译成2个三地址码

```
t2=tact *x
fact=t2
```

即使三地址码指令也是足够了，这种情况的技术原因将在8.2节中讲述。

8.1.2　用于实现三地址码的数据结构

三地址码通常不被实现成我们所写的文本形式 (虽然这是可能的)，相反是将其实现为包含几个域的记录结构。并将整个三地址指令序列实现成链表或数组，它能被保存在内存中并在需要时可以从临时文件中读写。

最通常的实现是将三地址码按其所显示的内容实现。这意味着有 4个域是必需的：1个操作符和3个地址。对于那些少于3个地址的指令，将一个或更多的地域置成 null或“ empty ”，具体选择哪个域取决于实现。必须有 4个域的三地址码表示叫做四元式 (quadruple)。程序清单8-2的三地址码的四元式在程序清单8-3中给出。这里我们用数学中的元组概念书写四元式。

程序清单8-3　程序清单8-2中的三地址码的四元式实现

```
(rd,x,_,_)
(gt,x,0,t1)
(if_f,t1,L1,_)
(asn,1,fact,_)
(lab,L2,_,_)
(mul,fact,x,t2)
```

```
(asn,t2,fact,_)
(sub,x,1,t3)
(asn,t3,x,_)
(eq,x,0,t4)
(if_f,t4,L2,_)
(wri,fact,_,_)
(lab,L1,_,_)
(halt,_,_,_)
```

程序清单8-4 程序清单8-3中四元式的数据结构的C定义

```
typedef enum {rd,gt,if_f,asn,lab,mul,
                sub,eq,wri,halt,. . .} OpKind;
typedef enum {Empty,IntConst,String} AddrKind;
typedef struct
            { AddrKind kind;
              union
              { int val;
                char * name;
              } contents;
            } Address;
typedef struct
            { OpKind op;
              Address addr1,addr2,addr3;
            } Quad;
```

程序清单8-4所示的是程序清单8-3中的四元式的C类型定义，在这些定义中，允许地址为整数、常量或字符串(代表临时变量或一般变量的名字)。由于使用了名字，就必须将其加入到符号表中，以供进一步处理时查询。另一种替代方法是在四元式中使用指向符号表入口的指针，这将避免额外的查询。这对可嵌套的语言来说有特别的好处，因为这时候的名字查询还需要更多的嵌套层信息，如果常量也输入符号表，那么将不再需要地址数据类型中的union。

三地址码另一个不同的实现是用自己的指令来代表临时变量，这样地址域从3个减少到了两个。因此在三地址指令中包含3个地址而目标地址总是一个临时变量⊖。如此的三地址码实现称为三元式(triple)。它要求：或是通过数组的索引号或是通过链表指针，每个三地址指令都是可引用的，如程序清单8-5是程序清单8-2的三地址码作为三元式实现的抽象表达。在那幅图中，我们使用了一个数字系统，它对应于代表三元式的数组索引。在三元式内，把三元式引用用圆括号括起，以同常量相区别。程序清单8-5取消了`label`指令，代之以三元式引用。

程序清单8-5 程序清单8-2中三地址码的三元式表示

```
(0)    (rd,x,_)
(1)    (gt,x,0)
(2)    (if_f,(1),(11))
(3)    (asn,1,fact)
(4)    (mul,fact,x)
(5)    (asn,(4),fact)
(6)    (sub,x,1)
(7)    (asn,(6),x)
```

⊖ 这不是三地址码的固有属性，但能通过实现来保证。例如，程序清单8-2的代码就是这样的(程序清单8-3也是)。

```
(8)     (eq,x,0)
(9)     (if_f,(8),(4))
(10)    (wri,fact,_)
(11)    (halt,_,_)
```

三元式是代表三地址码的有效方法，空间数量减少了且编译器不需要产生临时变量名；然而，三元式也有一个不利因素：用数组索引代表三元式使得三元式位置的移动变得很困难，而如用链表的话就不存在这个问题。三元式和对三元式的C代码定义的问题仍处于实践价段。

8.1.3 P-代码

在70年代和80年代早期，P-代码作为由许多Pascal编译器产生的标准目标汇编代码被设计成称作P-机器(P-machine)的假想栈机器的实际代码。P-机器在不同的平台上由不同的解释器实现。这个思想使得pascal编译器变得容易移植，只需对新平台重写 P-机器解释器即可。P-代码已被证明是一个非常有用的中间代码，它的各种扩展和修改版在许多自然代码的编译器中得到了使用，其中大多数都是针对类Pascal语言的。

由于将P-代码设计成直接可执行的，所以它包含了对特殊环境的明确描述、数据尺寸，还有P-机器大量的特有信息，如果要理解P-代码程序，就必须提供上述信息。为避开细节而恰当地说明问题，在这里只描述P-代码的一个简化的抽象版本。各种不同版本的实际 P-代码的描述能在本章最后列出的大量参考书中找到。

从我们的目的出发，P-机器包括一个代码存储器、一个未指定的存放命名变量的数据存储器、一个存放临时数据的栈，还有一些保持栈和支持执行的寄存器。作为 P-代码的第1个例子，考虑如下表达式，这个表达式在8.1.1节中已用过，它的语法树在8.1.1节：

```
2*a+(b-3)
```

这个表达式的P-代码版本如下：

```
ldc 2           ; load constant 2
lod a           ; load value of variable a
mpi             ; integer multiplication
lod b           ; load value of variable b
ldc 3           ; load constant 3
sbi             ; integer substraction
adi             ; integer addition
```

这些指令被看作代表如下的P-机器操作：`ldc 2`首先将值2压入临时栈，然后，`lod a`将变量`a`的值压入栈。指令`mpi`将这两个值从栈中弹出，使之相乘 (按弹出的相反顺序)，再将结果压入栈。接下来两个指令(`loa b`和`ldc 3`)将`b`的值和常量3压入栈(现在栈中有3个值)，随后，`sbi`指令弹出栈顶的两个值，用第1个值去减第2个值，再把结果压入栈中，最后`adi`指令弹出余下的两个值并使之相加，再将结果压入栈。代码结束时，栈中只有一个值，它代表了这次运算的结果。

作为第2个例子，考虑赋值语句：

```
x := y + 1
```

对应于如下的P-代码指令：

```
lda x           ; load address of x
lod y           ; load value of y
```

```
ldc 1          ; load constant 1
adi            ; add
sto            ; store top to address
               ; below top & pop both
```

注意，这段代码首先计算x的地址，然后将表达式的值赋给**x**，最后执行**sto**命令，这个命令需要临时栈顶上的两个值：一个是要被存储的值，在它下面的那个是值所要存入的地址。**sto**指令也弹出两个值(在这例子中使栈变空)。在**x:=y+1**中，左边的**x**的用处和右边的**y**的用处不一样，相应的P-代码用装入地址(**lda**)和装入值(**lod**)作出区别。

作为本节中最后的一个P-代码例子，我们对程序清单8-1中的TINY程序给出P-代码的翻译，如程序清单8-6所示，其中每条操作都有注释。

程序清单8-6 程序清单8-1中TINY程序的P-码指令

```
lda x          ; load address of x
rdi            ; read an integer, store to
               ; address on top of stack (& pop it)
lod x          ; load the value of x
ldc 0          ; load constant 0
grt            ; pop and compare top two values
               ; push Boolean result
fjp L1         ; pop Boolean value, jump to L1 if false
lda fact       ; load address of fact
ldc 1          ; load constant 1
sto            ; pop two values, storing first to
               ; address represented by second
lab L2         ; definition of label L2
lda fact       ; load address of fact
lod fact       ; load value of fact
lod x          ; load value of x
mpi            ; multiply
sto            ; store top to address of second & pop
lda x          ; load address of x
lod x          ; load value of x
ldc 1          ; load constant 1
sbi            ; subtract
sto            ; store (as before)
lod x          ; load value of x
ldc 0          ; load constant 0
equ            ; test for equality
fjp L2         ; jump to L2 if false
lod fact       ; load value of fact
wri            ; write top of stack & pop
lab L1         ; definition of label L1
stp
```

程序清单8-6中的P-代码包含了几个新的P-代码指令。首先，无参的**rdi**和**wri**指令实现了TINY中的整型的**read**和**write**语句。**rdi** P-代码指令要求要读的那个变量地址应位于栈顶，这个地址作为指令的一部分被弹出。**wri**指令要求要写的值地址位于栈顶，这个值作为指令的一部分弹出。程序清单8-6中出现的另一些新指令是**lab**指令，它定义了标签名字的位置；**fjp**指令(“false jump”)需要在栈顶有一个布尔值；**sbi**指令(整数减法)的操作类似于其他算术指令；**grt**指令(大于)和**etu**(等于)指令需要两个整型值位于栈顶(要被弹出)然后压入他们的布尔

型结果。**stp**指令对应于前面三地址码的**halt**指令。

1) **P-代码和三地址码的比较** P-代码在许多方面比三地址码更接近于实际的机器码。P-代码指令也需要较少地址；我们已见过的都是一地址或零地址指令，另一方面，P-代码在指令数量方面不如三地址码紧凑，P-代码不是自包含的，指令操作隐含地依赖于栈(隐含的栈定位实际上就是“缺省的”地址)，栈的好处是在代码的每一处都包含了所需的所有临时值，编译器不用如三地址码中那样为它们再分配名字。

2) **P-代码的实现** 历史上，P-代码已经大量地作为文本文件生成，但前面的三元地址码的内部数据结构描述(三元式和四元式)也能作用于P-代码的修改版。

8.2 基本的代码生成技术

本节讨论代码生成的基本方法，在下一节，我们将针对单个的语言结构进行代码生成。

8.2.1 作为合成属性的中间代码或目标代码

中间代码生成(或没有中间代码的直接目标代码生成)能被看作是一个属性计算，这类似于第6章中研究的许多属性问题，实际上假如生成代码被看作一个字符串属性(每条指令用换行符分隔)，这个代码就成了一个合成属性并能用属性文法定义，并且能在分析期间直接生成或者通过语法树的后序遍历生成。

为了看清楚三地址码或P-代码怎样被作为合成属性定义，考虑下边的文法，它代表了C表达式的一个子集。

exp→***id*** **=** *exp* | *aexp*
aexp→*aexp* **+** *factor* | *factor*
factor→**(** *exp* **)** | ***num*** | ***id***

这个文法仅包含了两个操作，赋值(=)和加法(+)[⊖]。记号***id***代表简单标识符，记号***num***代表了表示整数的简单数字序列。这两个记号被假设成有一个预先计算过的*strval*属性，它可以是字符串或词(例如“42”是***num***，“xtemp”是***id***)。

1) **P-代码** 我们首先考虑P-代码的情况，由于不需要产生临时变量名，属性文法会简单些，然而，嵌套赋值的存在是一个复杂因素。在这种情况下，我们希望保留被存的值作为赋值表达式的结果值，然而标准的P-代码指令**sto**是有害的，因为所赋的值会丢掉(P-代码在这里显示出了它pascal源，在pascal源代码中不存在嵌套的赋值语句)。我们通过引入一个无害的存储(nondestructive store)指令**stn**来解决这个问题，**stn**和**sto**一样，都假设栈顶有一个值且下面有一地址。**stn**将值存入那个地址，但在丢弃那个地址时栈顶上仍保留了那个值。表8-1是使用这个新的指令后P-代码属性的属性文法。在那幅图中，已经用属性名pcode表示P-代码串，并已把两个不同的符号用于串的连接：++表示所连的串之间不能在同一行，||表示连接一个串用空格相隔。

我们将跟踪某个例子的pcode属性计算留给读者并写出来，例如：表达式**(x=x+3)+4**有如下的pcode属性：

```
lda x
lod x
```

⊖ 这个例子中的赋值有如下的语义：**x=e**将**e**的值存入**x**，该赋值的结果值是**e**。

```
ldc 3
adi
stn
ldc 4
adi
```

表8-1 P-代码合成字符串属性的属性文法

文法规则	语义规则
$exp_1 \rightarrow$ **id** = exp_2	exp_1.pcode = **"lda"** ‖ **id**.strval ++ exp_2.pcode ++ **"stn"**
$exp \rightarrow aexp$	exp.pcode = aexp.pcode
$aexp_1 \rightarrow aexp_2 + factor$	$aexp_1$.pcode = $aexp_2$.pcode ++ factor.pcode ++ **"adi"**
$aexp \rightarrow factor$	aexp.pcode = factor.pcode
$factor \rightarrow (exp)$	factor.pcode = exp.pcode
$factor \rightarrow$ **num**	factor.pcode = **"ldc"**‖**num**.strval
$factor \rightarrow$ **id**	factor.pcode = **"lod"**‖**id**.strval

2) **三地址码** 前面那个简单表达式的三地址码属性文法在表 8-2中给出。在那张表中，我们称代码属性为tacode。同表8-1，也用++表示其间插有换行符的串连接，‖表示其间有空格的串连接。与P-代码不同，三地码要求为表达式的中间结果生成临时变量名，这就要求属性文法在每个节点中都包括一个新名字属性。这个属性也是合成的，如果没有为一个内部节点分配一个新产生的临时名，就用 *newtemp*( )产生一个临时名字系列 **t1**、**t2**、**t3**，. . .(每次调用*newtemp*( )就返回一个新的)。在这个简单例子中，仅对应于 +的节点需临时名，赋值操作使用右边的表达式的名字。

表8-2 三地址码作为合成串属性的属性文法

文法规则	语义规则
$exp_1 \rightarrow$ **id** = exp_2	exp_1.name = exp_2.name
	exp_1.tacode = exp_2.tacode ++ **id**.strval ‖ **"="**‖exp_2.name
$exp \rightarrow aexp$	exp.name = aexp.name
	exp.tacode = aexp.tacode
$aexp_1 \rightarrow aexp_2$ + factor	$aexp_1$.name = newtemp()
	$aexp_1$.tacode = $aexp_2$.tacode ++ factor.tacode ++ $aexp_1$.name‖ **"="**‖ $aexp_2$.name ‖ **"+"**‖factor.name
$aexp \rightarrow factor$	aexp.name = factor.name
	aexp.tacode = factor.tacode
$factor \rightarrow$ **(** *exp* **)**	factor.name = exp.name
	factor.tacode = exp.tacode
$factor \rightarrow$ **num**	factor.name = **num**.strval
	factor.tacode = **""**
$factor \rightarrow$ **id**	factor.name = **id**.strval
	factor.tacode = **""**

表8-2的产生式*exp*→*aexp*和*aexp*→*factor*中，将名字属性即*tacode*属性从子节点提到父节点，在操作符内部节点中，新名字属性应在联合的 *tacode*代码之前生成。在叶产生式*factor*→**num**和

factor→**id** 中记号的串值记作*factor.name*。与P-代码不同，在叶产生式的节点上没有生成三地址码(用""表示空串)。

再次，我们让读者按表8-2所给出的等式写出表达式 **(x=x+3)+4**的每一步的*tacode*，这个表达式的*tacode*属性如下：

```
t1 = x+3
x = t1
t2 = t1+4
```

(这里假设*newtemp*( )用后序调用并且产生从**t1**开始的临时名)。注意**x=x+3**是怎样用临时名产生两个三地址指令的。这是属性值总是为每一个子表达式产生一个临时名的结果，它包括了赋值号的右边部分。

将代码生成看成一个合成字符串属性计算，对于清楚地显示语法树各部分代码系列的关系以及比较不同的代码生成方法是很有用的，但它作为真实的代码生成技术是不实际的，这有几个原因：首先，串连接的使用造成了过度的串拷贝因而浪费了内存 (除非连接符做得非常复杂)，其次，通常希望产生几片代码作为代码产生的收益并且将这几片代码写入一个文件或将它们插入一个数据结构(如四元式数组)，这就需要语义动作，而语义动作又不与属性的标准后序合成有牵连。最后，即使将代码看作是纯合成是有用的，但通常的代码生成很大程度上依赖于继承属性，这将使得属性文法大大复杂化。由于这个原因，我们就不必麻烦再去写出实现前面例子中的属性文法的代码了(即使是伪码)。相反地，在下一节中，我们将转向更直接的代码生成技术。

8.2.2 实际的代码生成

标准的代码生成或者涉及语法树后序遍历的修改，这棵语法树是由前面例子的属性文法所包含的，或者如没有显式生成的语法树，则在分析中涉及了相等效的动作。基本算法由下面的递归过程描述(用于二叉树，但容易将其推广到节点子树多于2的情况)：

```
procedure genCode ( T:treenode );
begin
  if T is not nil then
    generate code to prepare for code of left child of T;
    genCode (left child of T );
    generate code to prepare for code of right child of T;
    genCode (right child of T );
    generate code to implement the action of T;
end;
```

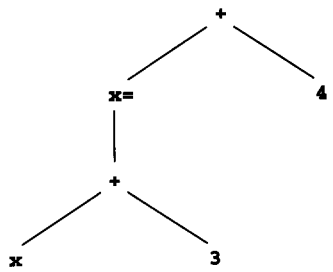
注意，这个递归遍历过程不仅有一个后序部分 (产生实现*T* 动作的代码)，而且还有一个前序和一个中序部分(为*T* 的左右子树产生准备代码)。通常，*T* 表示的每一个动作需要前序和中序准备代码的稍微不同的一个版本。

为了详细地看清在一个特殊的例子中怎样构造产生代码的过程，考虑我们在这一节已用过的简单算术表达式的语法，这个语法的抽象语法树的C语言定义如下所示：

```
typedef enum { Plus, Assign } Optype;
typedef enum { OpKind, ConstKind, IdKind } NodeKind;
```

```
typedef struct streenode
     { NodeKind kind;
       Optype op; /* used with OpKind */
       struct streenode *lchild, *rchild;
       int val; /* used with ConstKind */
       char * strval;
         /* used for identifiers and numbers */
     } STreeNode;
typedef STreeNode *SyntaxTree;
```

用这些定义，表达式**(x=x+3)+4**的语法树如下所示：



注意，赋值节点包含了要被赋于的表达式 (在**strval**域中)，以致一个赋值节点只有一个子树(要被赋于的表达式) ⊖。

基于这棵语法树的结构，我们能写出一个 **genCode**过程来产生程序清单 8-7的P-代码。对图中的代码作如下注释：首先，代码用标准 C函数**sprintf**把字符串连接到本地的临时变量**codestr**。其次，调用过程**emitCode**用以生成一个P-代码行，在数据结构或输出文件中不显示它的细节。最后，两个操作符 (加号和赋值号)需要两个不同的遍历顺序，加号仅需要一些后序处理，而赋值需要一些前序处理和一些后序处理。因此不可能在所有情况下，能将递归调用都写成一个样子。

程序清单8-7　表8-1属性文法定义的P-码的代码生成过程的实现

```
void genCode( SyntaxTree t)
{ char codestr[CODESIZE];
  /* CODESIZE = max length of 1 line of P-code */
  if (t != NULL)
  { switch (t->kind)
    { case OpKind:
        switch (t->op)
        { case Plus:
            genCode(t->lchild);
            genCode(t->rchild);
            emitCode("adi");
            break;
          case Assign:
            sprintf(codestr,"%s %s",
                            "lda",t->strval);
            emitCode(codestr);
            genCode(t->lchild);
```

⊖ 在这个例子中，我们将数字也当作字符串保存在**strval**域中。

```
            emitCode("stn");
            break;
          default:
            emitCode("Error");
            break;
        }
        break;
      case ConstKind:
        sprintf(codestr,"%s %s","ldc",t->strval);
        emitCode(codestr);
        break;
      case IdKind:
        sprintf(codestr,"%s %s","lod",t->strval);
        emitCode(codestr);
        break;
      default:
        emitCode("Error");
        break;
    }
  }
}
```

即使用到了必需的不同顺序的遍历，显示代码生成仍然可以在分析时进行（没有语法树的生成），我们在程序清单8-8中出示了一个Yacc说明文件，它直接对应于程序清单8-7的代码(注意赋值的前序和后序的组合处理是怎样翻译成独立的部分)。

程序清单8-8　根据表8-1的属性文法生成P-代码的Yacc说明

```
%{
#define YYSTYPE char *
    /* make Yacc use strings as values */

/* other inclusion code ... */
%}

%token NUM ID

%%

exp     : ID
          { sprintf(codestr,"%s %s","lda",$1);
            emitCode(codestr); }
          '=' exp
          { emitCode("stn"); }
        | aexp
        ;
aexp    : aexp '+' factor {emitCode("adi");}
        | factor
        ;

factor  : '(' exp ')'
        | NUM       { sprintf(codestr,"%s %s","ldc",$1);
                      emitCode(codestr); }
        | ID        { sprintf(codestr,"%s %s","lod",$1);
```

```
                    emitCode(codestr); }
        ;

%%
/* utility functions ... */
```

请读者按表8-2的属性文法写出一个*genCode*过程和生成三地址码的Yacc说明。

8.2.3 从中间代码生成目标代码

如果编译器或者直接从分析中或者从一棵语法树中产生了中间代码，那么下一步就是产生最后的目标代码(通常在对中间代码的进一步处理之后)，这一步本来就相当复杂，特别是当中间代码为高度象征性的，且只包含了很少或根本没包含目标机器和运行时环境的信息。在这种情况下，最后的代码生成必须支持变量和临时变量的实际定位，并增加支持运行时环境所必需的代码。寄存器的合适定位和寄存器使用信息的维护(如哪个寄存器可用和哪个包含了已知值)是一个特别重要的问题，我们将在本章最后再讨论分配问题细节。现在仅讨论这个处理的通用技术。

通常，来自中间代码的代码生成涉及了两个标准技术：宏扩展(macro expansion)和静态模拟(static simulation)。宏扩展涉及到用一系列等效的目标代码指令代替每一种中间代码指令。这要求编译器保留关于定位和独立的数据结构的代码惯用语的决定，它要求宏过程按照中间代码中涉及的特殊数据的需要改变代码序列。因此，这一步要比在C预处理器或宏汇编器中可利用的宏扩展的简单形式复杂得多。静态模拟包括中间代码效果的直线模拟和生成匹配这些效果的目标代码。这也需要额外的数据结构，它可以是非常简单的跟踪形式，并在与宏扩展的连接中使用，也可以是非常复杂的抽象解释(abstract interpretation) (当计算值时，对它们保持代数学的追踪)。

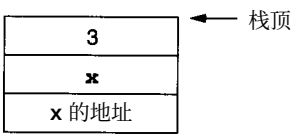
在考虑从P-代码翻译成三元地址码 (反之亦然)时，我们能了解这些技术的细节。想象一个在这节中已作为运行例子用过的小表达式语法，并考虑表达式 **(x=x+3)+4**，它的P-代码和三地址码的翻译在前面已经给出。我们首先考虑将这个表达式的 P-代码：

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

翻译成相应的三地址码：

```
t1 = x + 3
x = t1
t2 = t1 + 4
```

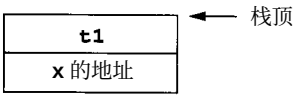
这需要执行一个P-机器栈的静态模拟用以发现代码的三地址码等效式。在翻译期间，用实际的栈数据结构实现它们。在前三条P-代码指令后仍无三地址码指令产生，但是已经将 P-机器栈修改以反映这些装入。栈内容如下所示：



现在当处理**adi**操作时，产生三地址指令

```
t1=x+3
```

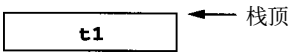
栈改成：



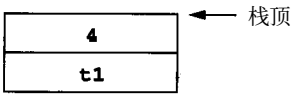
stn指令造成三地址指令

```
x=t1
```

生成，栈变成为：



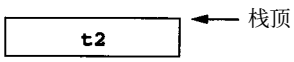
下一个指令压常量4入栈:



最后**adi**指令生成三地址指令

```
t2 = t1 + 4
```

栈变成为



这就完成了静态模拟和翻译。

我们现在考虑将三地址码翻译成P-代码的情况。假如忽略临时变量名增加的复杂性，就能用简单的宏扩展来完成。因此，一个三地址指令

```
a = b + c
```

总能被翻译成如下的P-代码指令序列

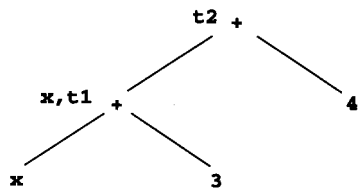
```
lda a
lod b ; or ldc b if b is a const
lod c ; or ldc c if c is a const
adi
sto
```

这导致了如下的三地址码到P-代码的翻译(有点不令人满意)

```
lda t1
lod x
ldc 3
adi
sto
lda x
lod t1
sto
lda t2
```

```
lod t1
ldc 4
adi
sto
```

假如要消除额外的临时变量，就需要用一个比宏扩展复杂得多的方案。一种可能是从三地址码生成一棵新树，这棵树用每个指令的操作符和所分配的名字来标记树节点，从而显示代码的效果。它可看作是静态模似的一种形式。前面的三地址码的结果树如下：



注意三地址指令

```
x = t1
```

在树中不生成节点，而是造成 **t1**节点获得另一个名字 **x**。这棵树同源代码的语法树类似，但又有不同⊖。P-代码能从这个树产生，这非常类似于从语法树中产生 P-代码。通过对内部节点只分配永久名，而消除了临时变量。因此，在这棵样例树中，只分配了 **x**，在P-代码中根本没有使用 **t1**、**t2**。对应于根节点 (**t2**)的值被放在P机器栈中。这导致了同前面完全一样的P-代码生成，只要在存储时用 **stn**代替**sto**即可。我们鼓励读者编写伪码或 C代码执行这个处理。

8.3 数据结构引用的代码生成

8.3.1 地址计算

在前一节中，我们已经看到如何生成一个简单的算术表达式和赋值语句的中间代码。这些例子中，所有的基本值或者是常量或者是简单变量 (程序变量如**x**，临时变量如**t1**)。简单变量仅通过名字识别一一到目标代码的翻译需要这些名字由实际地址代替，这些地址可以是寄存器、绝对地址(针对全局变量)，或是活动记录的偏移量 (针对局部变量，可能包括嵌套层)。这些地址可以在中间代码生成时插入，也可以推迟到实际代码生成时 (用符号保持地址)。

为了定位实际地址，有许多情况需要执行地址计算，即使是在中间代码中，这些计算也必须被直接表达出来。这样的计算发生在数组下标记录域和指针引用中。我们将依次讨论这些情况。但首先将描述一下可以表达这样的地址计算的三地址码和 P-代码扩展。

1) **用于地址计算的三地址码** 在三地址码中，对新操作符的需要不是很多。通常的算术操作符能被用来计算地址，但对于显示地址模式的方法需要几个新符号。在我们的三地址码版本中，使用了与C语言中意义一样的“ & ”和“ * ”来指示地址模式。例如，假设想把常量 2存放在变量**x**加上10个字节的地址处，用三地址码表示如下：

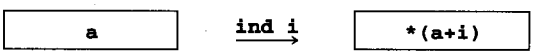
```
t1 = &x + 10
*t1 = 2
```

⊖ 这棵树是称为基本块的DAG(DAG of a basic block)的一般结构的特例，这在8.9.3节中描述。

这些新地址模式的实现要求三地址码的数据结构包含一个或多个新域。例如，图 8-4的四元式数据结构增加了一个枚举型的**AddrMode**域，枚举值为**None**、**Address**和**Indiredct**。

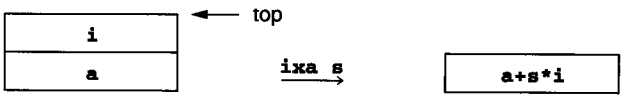
2) **用于地址计算的P-代码** 在P-代码中，通常引入新的指令表示新的地址模式(由于很少有外在的地址需要指定地址模式)。为了这个目的将引入如下两条指令：

• **ind** (间接装入) 用一个整型偏移量作为参数，假设栈顶上有一个地址，就将这个地址与偏移量相加得到新地址，再将新地址中的值压入栈以代替原来栈顶的地址。



执行前的栈　　执行后的栈

• **ixa** (索引地址) 用整型比例因子作为参数，假设一个偏移量已在栈顶并且在其下边有一个基地址，则用比例因子与偏移量相乘，再加上基地址以得到新地址，再将偏移量和基地址从栈中弹出，压入新地址。



执行前的栈　　执行后的栈

这两个P-代码指令，和以前介绍过的**lda** (装入地址)指令,将允许我们执行和三地址码中地址模式一样的地址计算和引用 ⊖。例如，前面的例子(把常量2存入变量**x**加10字节处的地址)现在用P-代码实现如下：

```
lda x
ldc 10
ixa 1
ldc 2
sto
```

我们现在开始讨论数组，记录和指针，随后是目标代码生成和一个扩展的例子。

8.3.2 数组引用

数组引用通过表达式涉及了数组变量下标，得到数组元素的引用或值。正如下面的 C代码所示：

```
int a[SIZE]; int i,j;
...
a[i+1] = a[j*2] + 3;
```

在这个赋值语句中，**a**的下标表达式**i+1**产生了一个地址(赋值的目标地址)，而通过下标表达式**j*2**在已计算的地址中得到**a**元素类型的一个值。由于数值按顺序存放于存储器中，每个地址必须根据**a**的基地址(base address) (即数组**a**在存储器中的起始地址)和线性地依赖于下标的偏移量计算。当需要的是一个值而不是地址时，就必须生成一个额外的间接步骤从已计算的地址中取出值。

⊖ 实际上，**ixa**指令能用算术运算符来模拟，除了在 P-代码中，这些操作被类型化 (**adi** = 整数相乘)，因此不能应用于地址。我们不强调P-代码的类型限制，因为这涉及额外的参数，为了简化，没有使用。

从下标值中计算偏移量如下：首先，假如下标范围不从 0开始(这在Pascal和Ada中是可能的)，就必须对下标值作一调整。其次，调整后的下标值必须与比例因子 (scale factor)相乘，比例因子等于存储器中数组元素类型的尺寸。最后比例作用过的下标值加上基地址，从而得到数组元素的最终地址。

例如，C数组引用**a[i+1]**的地址是：[⊖]

```
a + (i + 1) * sizeof (int)
```

更一般地，任何语言中数组元素**a[***t***]**的地址是：

$$base\_address\ (\mathbf{a}) + (t - lower\_bound\ (\mathbf{a})) * element\_size\ (\mathbf{a})$$

我们现在转向用三地址码和P-代码表示这个地址计算的方法。为了不依赖目标机器，假设数组变量的地址就是它的基地址。因此，如果**a**是数组变量，在三地址码和P-代码中，**&a**就是数组**a**的基地址。P-代码

```
lda a
```

将**a**的基地址压入 P-机器栈。由于数组引用计算依赖于目标机器元素类型的尺寸，所以用**elem-size(a)**代表目标机器上数组**a**的元素尺寸[⊜]。由于这是一个静态量(假设是静态类型)，这个表达式在编译时将用常量代替。

1) **数组引用的三地址码** 在三地址码中表示数组引用的一个可能的方法是引入两个新的操作。一个是获取数组元素的值

```
t2 = a[t1]
```

还有一个是给数组元素地址中赋值。

```
a[t2] = t1
```

(这些就用符号**=[]**和**[]=**表示)，用这个术语表示实际地址的计算不是必要的 (机器依赖性如元素尺寸将从这个概念中消失)。例如，源代码语句

```
a[i+1] = a [j*2] + 3
```

将翻译成下面的三地址指令:

```
t1 = j * 2
t2 = a [t1]
t3 = t2 + 3
t4 = i + 1
a [t5] = t3
```

然而，引入前面所述的地址模式仍是必需的，在处理记录域和指针引用时，用统一方式处理所有地址运算是有意义的，因此在三地址码中也直接写出数组元素地址的计算。例如，赋值语句：

```
t2 = a [t1]
```

能写成：(用更多的临时变量：**t3**和**t4**)

```
t3 = t1 * elem_size (a)
t4 = &a + t3
t2 = *t4
```

⊖ 在C中，一个数组(如此例中的**a**)的名字代表了它的基地址。

⊜ 这实际上是一个由符号表提供的函数。

赋值语句

```
a[t2] = t1
```

写成

```
t3 = t2 * elem_size(a)
t4 = &a + t3
*t4 = t1
```

最后，一个更复杂的例子，源代码语句

```
a[i+1] = a [j*2] + 3;
```

翻译成三地址指令如下：

```
t1 = j * 2
t2 = t1 * elem_size(a)
t3 = &a + t2
t4 = *t3
t5 = t4 + 3
t6 = i + 1
t7 = t6 * elem_size (a)
t8 = &a + t7
*t8 = t5
```

2) **数组引用的P-代码** 正如前面所描述的，我们用新的地址指令**ind**和**ixa**。指令**ixa**实际上由数组地址计算精确地构成。**ind**指令用来装入前面已计算地址的值 (例如实现一个间接装入)，数组引用

```
t2 = a [t1]
```

的P-代码表示如下：

```
lda t2
lda a
lod t1
ixa elem_size(a)
ind 0
sto
```

数组赋值

```
a[t2] = t1
```

的P-代码如下：

```
lda a
lod t2
ixa elem_size(a)
lod t1
sto
```

最后，前面那个较复杂的例子

```
a [i+1] = a [j*2] + 3 ;
```

翻译成P-代码如下：

```
lda a
lod i
```

```
ldc 1
adi
ixa elem_size(a)
lda a
lod j
ldc 2
mpi
ixa elem_size(a)
ind 0
ldc 3
adi
sto
```

3) **数组引用的代码生成过程** 这里将显示数组引用是怎样由一个代码生成过程产生的，我们用前一节中的C表达式子集的例子，但扩充了下标操作。要用的新文法如下：

exp → *subs* **=** *exp* | *aexp*
aexp → *aexp* **+** *factor* | *factor*
factor → **(** *exp* **)** | ***num*** | *subs*
subs → ***id*** | ***id*** **[** *exp* **]**

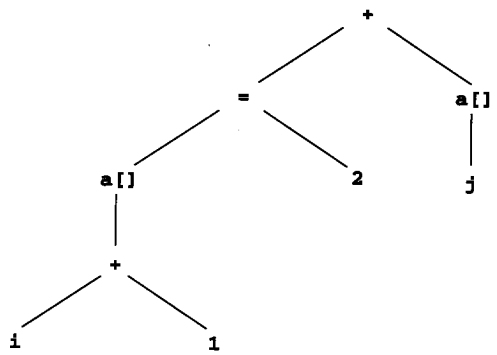
注意，赋值的目标可能是一个简单变量，也可能是一个有下标的变量（都包括在非终结符 *subs* 中)。我们使用和前面的语法树一样的数据结构，另外为下标增加 **Subs**操作

```
typedef enum { Plus, Assign, Subs } Optype;
/* 像前面一样的其他说明 */
```

由于下标表达式可以在一个赋值号的左边，那么是不可能将目标变量名存入赋值节点中（因为可能没有这样的名字)。赋值节点现在有 2个子树，这同加法节点一样。左子树必须是标识符或下标表达式。下标本身只能被用作标识符，因此，存储数组变量名于下标节点。所以，表达式：

```
(a[i+1]=2)+a[j]
```

的语法树如下：



程序清单8-9是一个为这样的语法树产生 P-代码的代码生成过程 (同程序清单8-7比照)。这个代码同程序清单8-7代码的主要不同在于它需要继承属性**isAddr**，用于识别下标表达式标识符在赋值号的左边还是在右边。如果**isAddr**被设置成**TRUE**，那么就必须返回表达式的地址；否则返回值。请读者检验下面的表达式**(a[i+1]=2)+a[j]**的P-代码生成过程：

```
lda a
lod i
ldc 1
adi
ixa elem_size(a)
ldc 2
stn
lda a
lod j
ixa elem_size(a)
ind 0
adi
```

在练习中也请读者检验这个文法的三地址码的代码生成器的构造。

程序清单8-9　上面的表达式文法对应的P-码的生成过程的实现

```
void genCode( SyntaxTree t, int isAddr)
{ char codestr[CODESIZE];
  /* CODESIZE = max length of 1 line of P-code */
  if (t != NULL)
  { switch (t->kind)
    { case OpKind:
        switch (t->op)
        { case Plus:
            if (isAddr) emitCode("Error");
            else { genCode(t->lchild,FALSE);
                   genCode(t->rchild,FALSE);
                   emitCode("adi");}
            break;
          case Assign:
            genCode(t->lchild,TRUE);
            genCode(t->rchild,FALSE);
            emitCode("stn");
            break;
          case Subs:
            sprintf(codestr,"%s %s","lda",t->strval);
            emitCode(codestr);
            genCode(t->lchild,FALSE);
            sprintf(codestr,"%s%s%s",
               "ixa elem_size(",t->strval,")");
            emitCode(codestr);
            if (!isAddr) emitCode("ind 0");
            break;
          default:
            emitCode("Error");
            break;
        }
        break;
      case ConstKind:
        if (isAddr) emitCode("Error");
        else
        { sprintf(codestr,"%s %s","ldc",t->strval);
          emitCode(codestr);
        }
```

```
            break;
          case IdKind:
            if (isAddr)
              sprintf(codestr,"%s %s","lda",t->strval);
            else
              sprintf(codestr,"%s %s","lod",t->strval);
            emitCode(codestr);
            break;
          default:
            emitCode("Error");
            break;
        }
      }
    }
```

4) **多维数组** 在数组地址计算中，在大多数语言中多维数组的存在是其复杂的一个因素。例如，在C语言中，二维数组(具有不同的索引大小)可以声明为：

```
int a[15][10] ;
```

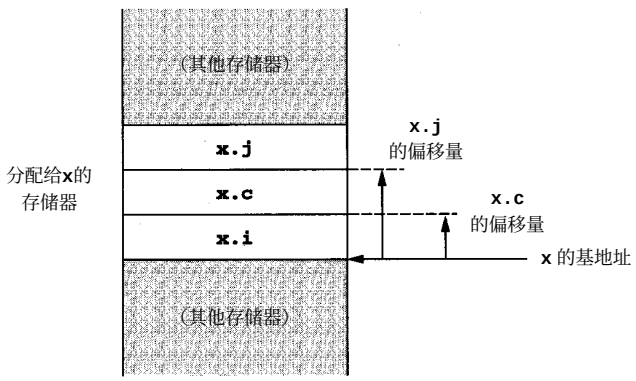
这样的数组可以用部分下标以生成维数更少的数组，或者完全使用下标以产生该数组元素类型的一个值。例如，在C语言中给定上面**a**的声明，表达式**a[i]**使用**a**的部分下标以产生一个一维整型数组，而表达式**a[i][j]**使用**a**的全部下标产生一个整型值。数组变量的部分或全部用下标表示后的地址可通过递归使用一维数组中描述的技术来计算。

8.3.3 栈记录结构和指针引用

计算记录结构的域地址提出了一个同计算下标数组地址相同的问题。首先，计算结构变量的基地址，然后，找到域偏移量(通常是固定的)，两者相加得到结果地址。例如，考虑C语言声明：

```
typedef struct rec
  { int i ;
    char c;
    int j ;
  } Rei;
...
Rec x;
```

一般地，变量**x**如下图所示那样在存储器中分配，每一个域(**i**、**c**和**j**)有一个从**x**基地址(可以是它自己在活动记录里的偏移量)开始的偏移量。



注意，域被线性分配(通常从低地址到高地址)，每个域的偏移量是常量，第1个(**x.i**)偏移量为0，注意：域偏移量依赖于目标机器上不同数据类型的大小，这里没有数组中的比例因子。

为了对记录结构域地址计算编写独立与于目标机器的中间代码，必须引入一个新函数返回域的偏移量，并给定结构变量和域名，这个函数称为 **field_offset**，并为它写出两个参数。第1个是变量名，第2个是域名，因此**field_offset(x,j)** 返回**x.j**的偏移量。和其他相似的函数一样，这个函数能由符号表提供。在任何情况下，这只是编译时的量，因此实际产生的中间代码将用常量代替对**field_offset**的调用。

记录结构一般使用指针和动态内存分配来实现动态数据结构 (如链表和树)，因此将描述指针和域地址计算是如何交互作用的。出于这里讨论的目的，一个指针只是建立了一个间接层次，而忽略了指针值产生的分配问题(这些在第7章讨论过)。

1) **结构和指针引用的三地址码** 首先考虑用于域地址计算的三地址码：为了计算**x.j**的地址并存入临时变量**t1**，使用如下的三地址指令：

```
t1 = &x + field_offset (x,j)
```

如C语句的一个域赋值表达式：

```
x.j = x.i ;
```

能被翻译成如下的三地址码:

```
t1 = &x + field_offset (x,j)
t2 = &x + field_offset (x,i)
*t1 = *t2
```

现在考虑指针。例如，假设**x**被定义成整型指针，例如C声明：

```
int * x;
```

再进一步假设**i**是一个普通整型变量， C赋值语句：

```
*x = i;
```

能被翻译成三地址指令:

```
*x = i
```

赋值语句

```
i = *x;
```

翻译成三地址指令

```
i=*x
```

为了看清楚指针的间接手段是怎样同域地址计算相互作用的，可考虑下面的树结构例子，C变量声明如下：

```
typedef struct treeNode
{ int val;
  struct treeNode * lchild, * rchild;
} TreeNode;
...
TreeNode *p;
```

现在考虑两个典型的赋值

```
p -> lchild = p;
p = p -> rchild;
```

这些语句翻译成三地址码如下：

```
t1 = p + field_offset ( *p, lchild )
*t1 = p
t2 = p + field_offset ( *p, rchild )
p = *t2
```

2) **结构和指针引用的P-代码** 给定本次讨论开始时的x定义，**x.j**的直接地址计算被翻译成下面P-代码

```
lda x
lod field_offset (x,j)
ixa l
```

赋值语句

```
x.j=x.i;
```

能被翻译成下面的P-代码

```
lda x
lod field_offset (x,j)
ixa 1
lda x
ind field_offset (x,i)
sto
```

注意**ind**指令在没有计算出**x.i**的完全地址时是怎样被用来获取它的值的。

在指针情形中(**x**被定义为**int ***)。赋值

```
*x = i;
```

翻译成P-代码如下

```
lod x
lod i
sto
```

赋值

```
i = *x;
```

翻译成P-代码如下

```
lda i
lod x
ind 0
sto
```

我们用P-代码可得出赋值

```
p -> lchild = p;
p = p -> rchild;
```

的推断(参见前面**p**的声明)。这些可翻译成如下P-代码：

```
lod p
lod field_offset ( *p, lchild )
ixa 1
lod p
sto
```

```
lda p
lod p
ind field_offset ( *p, rchild )
sto
```

我们将生成这些三地址码或P-代码的代码生成过程细节留作练习。

8.4 控制语句和逻辑表达式的代码生成

在这一节中，我们描述控制语句不同形式的代码生成。这里主要的部分是结构化的 if语句和while语句，这一节的开头将说明它。这个描述也包括了对 break语句的说明，但是因为这种语句能简单地用中间代码或目标代码直接实现，所以不讨论低级控制 (如goto语句)。结构化控制的其他形式，如repeat语句(或do-while语句)、for语句和case语句(或switch语句)的描述将留作练习。在switch语句中用到的额外的实现技术称作转移表(jump table)，也在练习中描述。

控制语句的中间代码生成——无论是在三地址码还是 P-代码中——都涉及到了标号的产生，这种方式与三地址码中临时变量名的生成相类似，但标号在目标代码中代表要转移的地址。假如在目标代码生成中消除了标号，则在转移中将引发一个代码定位问题，这将在本节的第 2部分讨论。逻辑表达式或布尔表达式被用作控制测试，但也可以独立地用作数据，这将在接下来的一部分中讨论。尤其是在短循环求值中，它们不同于算术表达式。

最后，在这节中，我们对if和while语句给出了一个P-代码生成过程的例子。

8.4.1 if和while语句的代码生成

考虑下面两种if和while语句形式，这在很多不同的语言中都是相似的 (现在给出的是类C语法)：

if-stmt → **if** (*exp*) *stmt* | **if** (*exp*) *stmt* **else** *stmt*

while-stmt → ***while*** (*exp*) *stmt*

对这样语句的代码生成的主要问题是：将结构化的控制特性翻译成涉及转移的非结构化等价物，它能被直接实现。编译器以一种标准次序来安排这种语句的代码生成，这种次序有可能高效地使用转移子集，这种转移子集是目标系统所允许的。图 8-1和图8-2是对每一个这种语句的典型代码排列 (图8-1显示一个else部分(虚假的情况)，但根据刚刚给定的文法规则这是一个可选项，这个排列对于缺少的else部分的修改是很容易的)，在这些排列中，仅有两种转移——无条件转移和条件为虚假的转移。条件为真时是不需要转移的“失败”情况。这减少了编译器要产生的转移的数量，这也意味着在中间代码中仅需要两个转移指令。虚假转移已经在程序清单 8-2中的三地址码中出现过了(如**if-false..goto**语句)，此外也在程序清单8-6的P-代码中的三地址码中出现过了(如**fjp**指令)。剩下要介绍的无条件转移，它将在三地址码中用**goto**指令实现，在P-代码中用**ujp** (无条件转移)实现。

1) **控制语句的三地址码** 我们假设代码生成器产生了一系列标号**L1**、**L2**…，对于语句

if (*E*) ***S1*** **else** ***S2***

生成下面的代码模式：

```
<code to evaluate E to t1>
if_false t1 goto L1
<code for S1>
```

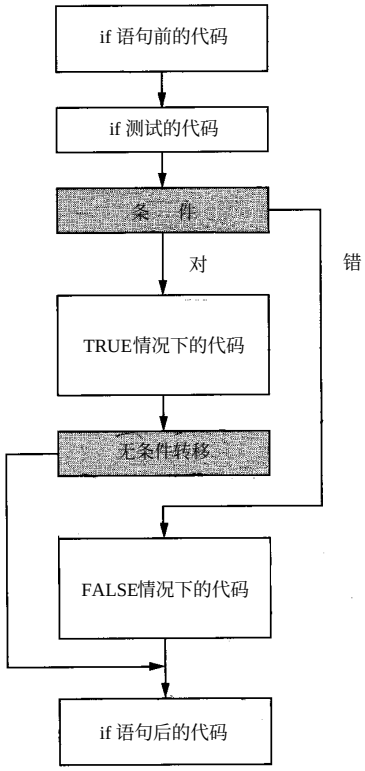


图8-1 if语句的典型代码排列

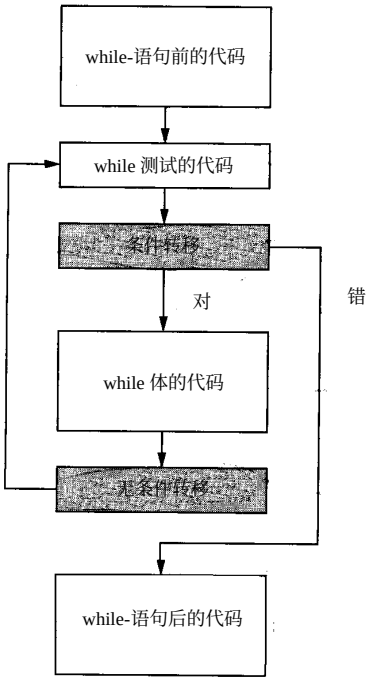


图8-2 while语句的典型代码排列

```
goto L2
label L1
<code for S2>
label L2
```

类似地，while语句

```
while ( E ) S
```

将导致生成下面的三元地址码样式

```
label L1
<code to evaluate E to t1>
if_false t1 goto L2
<code for S>
goto L1
label L2
```

2) **控制语句的P-代码** 对于语句

```
if ( E ) S1 else S2
```

生成下面的P-代码样式

```
<code to evaluate E>
fjp L1
<code for S1>
ujp L2
```

```
lab L1
<code for S2>
lab L2
```

对于语句

```
while ( E ) S
```

生成下面的P-代码样式

```
lab L1
<code to evaluate E>
fjp L2
<code for S>
ujp L1
lab L2
```

注意，所有这些代码序列(三地址码和P-代码)都以标号定义结束。我们称这个标号为控制语句的出口标号(exit label)。许多语言提供一个语言结构允许循环在循环体的任意位置退出。例如，C语言提供了**break**语句(这也能被用在switch语句中)。在这些语言中，出口标号对于所有代码生成例程都是可用的，这些代码生成例程在循环体内可以使用。因此如果遇上一个退出语句如break，就产生一个到出口标号的转移。在代码生成期间，将出口标号放入继承属性，这必须被存入栈或作为一个合适的代码生成例程的参数，关于这方面的更多细节，将在这一节的后面给出。

8.4.2 标号的生成和回填

在目标代码生成期间，引起问题的控制语句代码生成的一个特性实际上是对标号的转移必须在标号本身定义之前生成。在中间代码的生成期间这不会有什么问题。由于因向前转移而需要一个标号时，代码生成例程能只是调用标号生成过程，并且一直保存这个标号名字 (局部的或在栈中)到知道这个标号的位置为止。假如在目标代码生成期间要产生汇编代码，标号能只是传给汇编器。但如果要生成实际的可执行代码，这些标号就必须被解析成绝对的或相对的代码位置。

产生这样一个向前转移的一个标准方法是：在转移发生的代码处留下空位，或者产生一个虚假的转移(针对虚假地址)。然后当知道实际的转移位置时，这个位置用来回填 (backpatch)缺少的代码。这也要求产生代码并保存在内存缓冲区内以易于能频繁地进行回填，或者将代码写入一个临时文件，在需要的时候再重新输入或回填。另一种情况是：回填可能需要在一个栈中缓冲或者在递归过程中局部地保持。

在回填处理中，更深入的问题出现了，这是由于许多结构中有两种转移，一个短转移 (含有128字节)，一个需要更多的代码空间的长转移。在这种情况下，代码生成器可能在短转移时需要插入nop指令或者采取办法缩短代码。

8.4.3 逻辑表达式的代码生成

到目前为止，我们还未提到过逻辑或布尔表达式的代码生成，它们经常作为控制语句的测试而使用。假如中间代码有一个布尔数据类型和逻辑操作符，如 **and**和**or**，那么就能在中间代码中计算布尔表达式的值，这与算术表达式一样。它是一个 P-代码的例子，能类似地设计出中间代码。然而即使是这个例子，因为大多数系统没有内置的布尔值，所以到目标代码翻译仍需

要将布尔值算术地表示出来。做这个的标准方法是将 **true**作为1，**fasle**作为0，然后标准的位运算符**and**和**or**能在大多数系统上用来计算布尔表达式的值。这需要将比较操作符如 <的结果规格化为0或1。在一些系统中，因为比较操作符本身仅仅设置条件码，所以需要显式地装入0或1。在那个例子中，条件转移需要装入合适的值。

如果逻辑操作符是短路(short circuit)的，更深入地使用转移是必须的。如果不再求它的第2个参数，那么这个逻辑操作符就被短路了。例如，假如 *a*是一个布尔表达式，值是 false，那么就能立即断定布尔表达式 *a* **and** *b*为假，也就不用去求*b*了。类似地，如*a*为真，那么立刻就能判断出*a* **or** *b*为真，也不用求*b*了。短路的操作符对代码来说是非常有用的。因为如果操作符没被短路的话，对第2个表达式的求值可能会引起错误，例如，在C中这是很平常的：

```
if ((p!=NULL) && ( p->val==0) ) ...
```

这里当**p**为可空时**p->val** 的求值将引起内存错误。

除了它们还返回值以外，短路布尔操作符类似于if语句，它们经常用if表达式定义，如

a **and** *b* ≡ **if** *a* **then** *b* **else false**

和

a **or** *b* ≡ **if** *a* **then true else** *b*

为产生确保第2个子表达式仅在必要的时候进行求值的代码，我们必须如同在 if语句中一样，使用转移。例如，对于(表达式**(x!=0)&&(y==x)**的短路P-代码如下：

```
lod x
ldc 0
neq
fjp L1
lod y
lod x
equ
ujp L2
lab L1
lod FALSE
lab L2
```

8.4.4 if和while语句的代码生成过程样例

这一节用如下简化的文法展示一个控制语句的代码生成过程。

stmt → *if-stmt* | *while-stmt* | **break** | **other**
if-stmt → **if(** *exp* **)** *stmt* | **if(** *exp* **)** *stmt* **else** *stmt*
while-stmt → **while(** *exp* **)** *stmt*
exp → **true** | **false**

出于简化目的，这个语法用**other**代表没有包括在文法中的语句(如赋值语句)，它也仅包括常量布尔表达式**true**和**false**。为了显示怎样将**break**语句作为参数传递的继承的标号来实现，它包括了一个break语句。

下面的C声明，能被用来为这个文法实现一棵抽象语法树

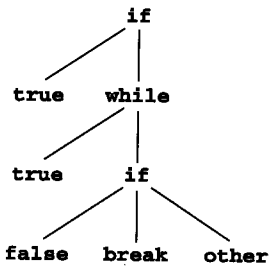
```
typedef enum { ExpKind,IfKind,
     WhileKind, BreakKind, OtherKind } NodeKind;
```

```
typedef struct streenode
       { NodeKind kind;
         struct streenode * child[3] ;
         int val; /* used with ExpKind */
       } STreeNode;
typedef STreeNode * SyntaxTree;
```

在这棵语法树结构中，1个节点可以有3个孩子(1个有else部分的if节点)。表达式节点包含了真假值(在**val**域中作为1或0存储)，例如语句

```
if (true) while (true) if (false) break else other
```

有如下的语法树：



这里我们仅显示了每个节点的非空子树⊖。

用被给定的**typedef**和相应的语法树结构，一个产生P-代码的代码生成过程在程序清单8-10中给出，我们对这段代码作了如下注释：

首先，这段代码假设已存在一个**emitCode**过程(这个过程只是将传给它的字符串打印出来)这段代码也假设返回顺序标号名(如**L1**、**L2**、**L3**…) 的无参过程**genLabel**已存在。

genCode过程有一个额外的标号参数，在为break语句而生成的转移语句中会用到它。仅在处理while语句循环体的递归调用中会改变这个参数。因此，break 语句将总是跳出最近的那个嵌套while语句(对**genCode**的初始调用能用一个空串作为标号参数，任何在 while语句外的break语句将产生一个到空标号的转移，这将引发错误)。

请注意标号变量**Lab1**和**Lab2**怎样用来在转移和(或)定义仍未决定时保存标号名的。

最后，由于other语句没有对应的实际代码，这个过程只是产生了非P-代码指令“**Other**”。

程序清单8-10　控制语句的代码生成过程

```
void genCode( SyntaxTree t, char * label)
{ char codestr[CODESIZE];
  char * lab1, * lab2;
  if (t != NULL) switch (t->kind)
  { case ExpKind:
      if (t->val==0) emitCode("ldc false");
      else emitCode("ldc true");
      break;
    case IfKind:
      genCode(t->child[0],label);
      lab1 = genLabel();
      sprintf(codestr,"%s %s","fjp",lab1);
```

⊖ 在这个语法中，标准的“最近嵌套”规则解决了悬挂的二义性，如语法图所示。

```
        emitCode(codestr);
        genCode(t->child[1],label);
        if (t->child[2] != NULL)
        { lab2 = genLabel();
          sprintf(codestr,"%s %s","ujp",lab2);
          emitCode(codestr);}
        sprintf(codestr,"%s %s","lab",lab1);
        emitCode(codestr);
        if (t->child[2] != NULL)
        { genCode(t->child[2],label);
          sprintf(codestr,"%s %s","lab",lab2);
          emitCode(codestr);}
        break;
      case WhileKind:
        lab1 = genLabel();
        sprintf(codestr,"%s %s","lab",lab1);
        emitCode(codestr);
        genCode(t->child[0],label);
        lab2 = genLabel();
        sprintf(codestr,"%s %s","fjp",lab2);
        emitCode(codestr);
        genCode(t->child[1],lab2);
        sprintf(codestr,"%s %s","ujp",lab1);
        emitCode(codestr);
        sprintf(codestr,"%s %s","lab",lab2);
        emitCode(codestr);
        break;
      case BreakKind:
        sprintf(codestr,"%s %s","ujp",label);
        emitCode(codestr);
        break;
      case OtherKind:
        emitCode("Other");
        break;
      default:
        emitCode("Error");
        break;
    }
  }
```

请读者跟踪程序清单8-10过程中的操作，并将这条语句的操作显示出来。

```
if (true) while (true) if (false) break else other
```

它产生了如下的代码序列

```
ldc true
fjp L1
lab L2
ldc true
fjp L3
ldc false
fjp L4
ujp L3
ujp L5
lab L4
```

```
Other
lab L5
ujp L2
lab L3
lab L1
```

8.5 过程和函数调用的代码生成

在本章中，过程和函数调用是在一般术语中讨论的最后的语言机制，对这个机制的中间代码和目标代码描述的复杂程度甚至超过了其他语言机制，因为不同的目标机器用相当不同的机制来执行调用，而且调用很大程度上依赖于运行时环境的组织。因此，实现一个对任何目标系统和运行时环境都够用的中间代码表示是困难的。

8.5.1 过程和函数的中间代码

函数调用的中间代码表示的要求可明确地表述如下：首先，有两个实际的机制需要说明——函数过程的定义(definition) (也叫声明(declaration)) 和函数过程的调用(call)⊖。定义产生了函数名、参数和代码，但函数却不在那点执行。调用产生了参数的实际值，并且执行一个到函数代码的转移。函数代码被执行和返回。当产生函数代码时，除了它的一般结构，执行的运行时环境还不知道其他情况。这个运行时环境部分由调用者建造，部分也由被调用函数代码建造。任务的分隔是调用次序(calling sequence) 的一部分，这在第7章已研究过了。

定义的中间代码必须包括一条标志开始的指令，或称函数代码的入口点 (entry point)，还要包括一条标志结束的指令，称为函数返回点 (return point)。写出概要如下：

Entry instruction
<code for the function body>
Return instruction

相似地，函数调用必须有一条指令指示参数计算的开始 (为调用而准备的)，然后实际调用指令指示已经构成了参数，到函数代码的实际转移可以发生了。

Begin-argument-computation instruction
<code to compute the arguments>
Call instruction

不同的中间代码版本在括号括起中的指令的实现上是不相同的。关于环境的信息数量、参数尤为如此，作为每一条指令一部分的函数本身的指令更是这样。这些信息的典型例子包括数字尺寸和参数的定位、栈的大小、局部以及临时变量空间的大小和被调用函数使用寄存器的指示。通常地，将产生一些在指令本身中已包含少量信息的中间代码，而任何必要的信息可以单独地放在过程的符号表中。

1) **过程和函数的三地址码** 在三地址码中，入口指令需要给出一个过程入口点的名字，这类似于`label`指令，因此，这是一条地址指令。我们将其简单地称作 `entry`，类似地把返

⊖ 通过这段文字，我们发现函数和过程本质上都代表了相同的机制，如没有特殊的说明，在这一节中认为它们是一样的。当然，仅有的区别是：当函数退出时，返回调用者可利用的返回值，并且调用者必须知道在哪儿能发现它。

回指令叫作**return**。这个指令也是一条地址指令，假如有返回值，那么它必须给出返回值的名字。

例如，考虑C函数定义

```
int f ( int x, int y )
{ return x + y + 1; }
```

这样翻译成如下的三地址码：

```
entry f
t1 = x + y
t2 = t1 + 1
return t2
```

在调用的情况下，实际需要3个不同的三地址指令：一个标志参数计算的起点，称作**begin_args**(是一个零地址指令)；一个被用于重复地说明参数值的名字的指令，我们称其为**arg**(它必须包括参数值的地址或名字)；最后是实际调用指令，简单地写成**call**，它也是一个一地址指令(被调用函数的名字或入口点必须给出)。

例如，假如上例中的函数**f**已经在C中定义，那么，这个调用

```
f ( 2+3, 4)
```

翻译成三地址码如下：

```
begin_args
t1 = 2 + 3
arg t1
arg 4
call f
```

在这里按从左到右顺序列出参数。这个顺序可以是不同的(参看7.3.1节)。

2) **过程和函数的P-代码** P-代码的入口指令是**ent**，返回指令是**ret**，前面C函数**f**的定义可以翻译成P-代码如下：

```
ent f
lod x
lod y
adi
ldc 1
adi
ret
```

注意，**ret**指令不需要一个参数用来指示返回的值。在返回时，返回值被认为已在P机器栈顶上。

用于调用的P-代码指令是**mst**指令和**cup**指令。**mst**指令表示“标志栈”对应于三地址码指令中的**begin-args**。它之所以称作“标志栈”的原因是从这种指令生成的目标代码将为一个新调用在栈上建立一个活动记录，这是调用序列中前几个步骤。这通常意味着，对于参数这样的元素，必须在栈中为它分配或“标志”空间。P-代码指令**cup**是“调用用户过程”指令，它直接对应于三地址码中的**call**指令。取这个名字的原因是P-代码区分两种调用——**cup**和**csp**(调用标准过程)。标准过程是语言定义所需的内部过程，如pascal中的**sin**和**abs**过程(C无内部过程可言)内部过程能用关于它们的操作的特殊知识来提高调用的效率(或者甚至消除这个调用)。这里就不再进一步考虑**csp**操作了。

注意，并没有引入等效于三地址码 **arg**指令的P-代码指令。相反地，在遇到 **cup**指令时，将所有参数值都假想成出现在栈顶 (按适当的顺序)。这导致了与三地址码稍微不同的调用序列顺序(见练习)。

前面所述的函数**f**，它的C调用例子**f(2+3,4)**可以翻译成如下的P-代码：

```
mst
ldc 2
ldc 3
adi
ldc 4
cup f
```

(再次从左向右计算参数)。

8.5.2 函数定义和调用的代码生成过程

和前面几节一样，我们想展示函数定义和调用语法样例的代码生成过程。要用的文法如下：

program → *decl-list exp*
decl-list → *decl-list decl* | ε
decl → **fn** ***id*** **(** *param-list* **)=** *exp*
param-list → *param-list*, ***id*** | ***id***
exp → *exp* + *exp* | *call* | ***num*** | ***id***
call → ***id*** (*arg-list*)
arg-list → *arg-list*, *exp* | *exp*

这个文法定义了一个程序，该程序是函数定义序列，这个序列后跟随着单个表达式。在这个文法中没有变量或赋值，只有参数、函数和可以包括函数调用的表达式。所有值都是整型的，所有函数返回整型，所有函数至少有1个参数。这里只有1个数字操作(除函数调用外)：整数加法。由这个文法定义的程序例子如下所示：

```
fn f(x)=2+x
fn g(x,y)=f(x)+y
g(3,4)
```

这个程序包含两个函数定义，其后跟着函数**g**的调用表达式。在**g**中有一个对函数**f**的调用。我们想对这个文法定义一个语法树结构，通过下面的C声明来进行：

```
typedef enum
 {PrgK, FnK, ParamK, PlusK, CallK, ConstK, IdK}
 NodeKind;
typedef struct streenode
       { NodeKind kind;
         struct streenode *lchild,*rchild,
                          *sibling;
         char * name; /* used with FnK,ParamK,Callk,IdK */
         int val; /* used with ConstK */
       } StreeNode;
typedef StreeNode * SyntaxTree;
```

在这个树结构里有7个不同种类的节点。每个语法树都有一个根节点**PrgK**。这个节点仅用来将声明和程序表达式绑在一起。一棵语法树只包含一个这样的节点。这个节点的左子树是 **FnK**节

点的兄弟链表。右子树是相关联的程序表达式。每一个 FnK节点都有一个左子树，它是ParamK节点的兄弟链表。这些节点定义了参数的名字。每个函数体都是 FnK节点的右子树。除了有一个CallK节点之外，表达式节点和通常的一样。这个节点包含了所谓函数的名字，并且有一个右子树是参数表达式的兄弟链表。例如，前面样例程序的语法树如图 8-3所示。为使之表达清楚，在该图中包括了每个节点的节点种类，还有所有的名字 /值属性。孩子和兄弟用方向区别(兄弟向右，孩子向下)。

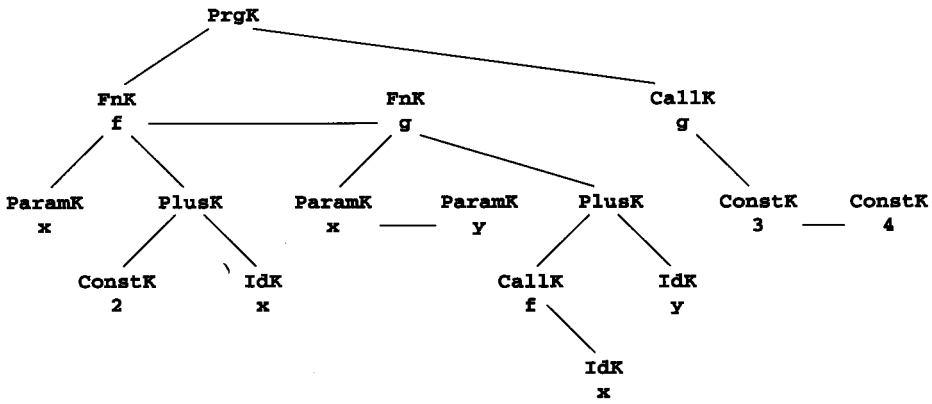


图8-3 上面程序例子的语法树

给定这棵语法树结构，产生P-代码的代码生成过程在程序清单8-11中给出。对这段代段作如下的注释：首先，PrgK节点的代码简单地向树的其余部分递归。Idk、ConstK或PlusK的代码实际上和前面例子中的一样。

程序清单8-11 函数定义和调用的代码生成过程

```
void genCode( SyntaxTree t)
{ char codestr[CODESIZE];
  SyntaxTree p;
  if (t != NULL)
  switch (t->kind)
  { case PrgK:
      p = t->lchild;
      while (p != NULL)
      { genCode(p);
        p = p->sibling; }
      genCode(t->rchild);
      break;
    case FnK:
      sprintf(codestr,"%s %s","ent",t->name);
      emitCode(codestr);
      genCode(t->rchild);
      emitCode("ret");
      break;
    case ParamK: /* no actions */
      break;
    case ConstK:
      sprintf(codestr,"%s %d","ldc",t->val);
      emitCode(codestr);
```

```
        break;
      case PlusK:
        genCode(t->lchild);
        genCode(t->rchild);
        emitCode("adi");
        break;
      case IdK:
        sprintf(codestr,"%s %s","lod",t->name);
        emitCode(codestr);
        break;
      case CallK:
        emitCode("mst");
        p = t->rchild;
        while (p!=NULL)
        { genCode(p);
          p = p->sibling; }
        sprintf(codestr,"%s %s","cup",t->name);
        emitCode(codestr);
        break;
      default:
        emitCode("Error");
        break;
    }
  }
```

这里剩下了**FnK**、**ParamK**和**CallK**的情况。**FnK**节点的代码只用**ent**和**ret**将函数体(右子树)的代码括上了。函数参数没有被访问。实际上参数节点不引起代码的产生，参数已经由调用者产生⊖。这也解释了在程序清单8-11中在**ParamK**节点上为什么没有动作。实际上由于**FnK**代码的行为，在树的遍历中，不会触及到任何一个**ParamK**节点。因此这种情况实际上不用讨论。

最后的例子是**CallK**。代码先发出一个**mst**指令，然后为每一个参数产生代码，最后发出**cup**指令。

请读者显示图8-11中的代码生成过程是怎样生成如下的 P-代码。给定的程序语法图如图8-3。

```
ent f
ldc 2
lod x
adi
ret
ent g
mst
lod x
cup f
lod y
adi
ret
mst
```

⊖ 因为参数节点参数在活动记录中的相对位置和位移，所以它们有着重要的簿记任务。假设这一点是在其他地方处理的。

```
ldc 3
ldc 4
cup g
```

8.6 商用编译器中的代码生成：两个案例研究

这一节检查两个不同商业编译器 (不同的处理器)产生的汇编代码输出。第 1个是Borland公司对80×86处理器的3.0版C编译器。第2个是Sun公司对于SparcStation的2.0版C编译器，我们将示出这些编译器对一些代码例子的汇编输出，那些代码例子和用来阐明三地址码和 P-代码所用的例子一样 ⊖ 。这将对代码生成技术和中间代码到目标代码的转化进行更深入的考察，同时它也提供了与TINY编译器生成的机器代码之间有用的比较， TINY的机器代码将在后面的章节中被讨论。

8.6.1 对于80×86的Borland 3.0版C编译器

我们用8.2.1节中使用的赋值表达式开始关于这个编译器输出的例子。这个赋值式是：

```
(x=x+3)+4
```

假设将变量**x**存于栈中。

对于这个表达式， Borland 3.0 C编译器产生的Intel 80×86上的汇编代码如下：

```
mov     ax, word ptr [bp-2]
add     ax, 3
mov     word ptr [bp-2], ax
add     ax, 4
```

在这段代码中，累加寄存器**ax**在计算中用作主要的临时变量位置。局部变量 **x**的位置是**bp-2**，它反应了寄存器**bp** (基指针)用作框架指针和整型变量在机器中占两个字节。

第1条指令把**x**的值移到**ax**中(地址**[bp-2]**的括号表示一个间接装入而不是一个直接装入)，第2个指令将常量3增加进这个寄存器。第3个指令将这个值移到**x**的位置。最后，第4条指令将4加入**ax**，以致于将表达式最后的值留在了寄存器。它可以为进一步的计算所使用。

注意，在第3个指令中赋值用的**x**的地址不需要预先计算(如P-代码指令**lda**)。中间代码的静态模拟连同可利用的地址模式的知识能将**x**的地址计算推迟到要用的时候。

1) **数组引用** 用C表达式

```
(a[i+1]=2)+a[j]
```

(参见“数组引用的代码生成过程”部分中的中间代码生成的示例。)作为例子，假设**i**、**j**和**a**被作局部变量声明：

```
int i,j;
int a[10];
```

Borland C编译器为这个表达式产生如下的汇编代码 (为使引用变得容易，我们对代码进行了编号)

```
(1)    mov     bx,word ptr [bp-2]
(2)    shl     bx,1
(3)    lea     ax, word ptr [bp-22]
```

⊖ 出于这个目的，不考虑编译器的优化。

```
(4)    add     bx,ax
(5)    mov     ax,2
(6)    mov     word ptr [bx],ax
(7)    mov     bx,word ptr [bp-4]
(8)    shl     bx,1
(9)    lea     dx,word ptr [bp-24]
(10)   add     bx,dx
(11)   add     ax,word ptr [bx]
```

因为在这个系统中整数的大小是2字节。**bp-2**是**i**在局部活动记录中的位置。**bp-4**是**j**的位置，**a**的基地址是**bp-24** (24 = 数组索引尺寸×整型尺寸的2个字节加上**i**和**j**的4个字节)，因此指令1将**i**的值装入**bx**，指令2将这个值乘以2(左移一位)。指令3将**a**的基地址装入**ax** (**lea**表示装入有效地址)，这个基地址由于下标表达式**i+1**中的常量1的缘故，已经增加了2个字节。换句话说，编译器已经实现了这样一个代数事实：

address (**a[i+1]**) = *base_address* (**a**) + (**i+1**)**elem_size* (**a**)
= (*base_address* (**a**) + *elem_size* (**a**)) + ***i*****elem_size* (**a**)

指令4计算**a[i+1]**的结果地址，并将其放入**bx**中。指令5将常数2移入**ax**中，指令6将其存入**a[i+1]**的地址中，指令7将**j**值装入**bx**，指令8将这个值乘2，指令9将**a**的基地址装到**dx**，指令10计算**a[j]**地址并放入**bx**，指令11把这个地址中的内容加入到**ax**中。表达式的结果值留在**ax**中。

2) **指针和域引用** 假设上一个例子的声明为：

```
typedef struct rec
   { int i;
     char c;
     int j;
   } Rec;
typedef struct treeNode
   { int val;
     struct treeNode * lchild, * rchild;
   } TreeNode;
...
Rec x;
TreeNode *p;
```

同时也假设**x**和**p**被声明为局部变量，也进行了适当的指针分配。

首先考虑涉及的数据类型的尺寸。在80×86系统中，整型变量占2个字节，字符变量占1字节，“近”指针占2个字节[⊖]。因此变量**x**有5个字节，变量**p**有2个字节。作为仅有的变量，它们被声明是局部的，**x**在活动记录中分配在**bp-6**位置(不能将局部变量分配在偶数字节界上，这是一个典型的限制，因此将不用额外的字节)，**p**被分配到寄存器**si**中。更进一步，在结构**Rec**中，**i**有偏移量0，**c**有偏移量2，**j**有偏移量3，而在树节点结构中，**val**有偏移量0，**lchild**有偏移量2，**rchild**有偏移量4。

语句

```
x.j =x.i;
```

⊖ 80×86有一个更一般的指针叫远指针，有4个字节。

的生成代码是：

```
mov    ax,word ptr [bp-6]
mov    word ptr [bp-3],ax
```

第1条指令将x.i装入ax，第2条将这个值存到x.j中。请注意，对j的偏移量计算(-6+3=-3)由编译器静态执行。

语句

```
p->l child = p;
```

的生成代码是：

```
mov word ptr [si+2], si
```

请注意怎样将间接和偏移量计算放进同一指令中的。

最后，语句

```
p = p->rchild;
```

的生成代码是

```
mov     si, word ptr [si+4]
```

3) **if和while语句** 这里将显示由Barland C编译器生成的典型控制语句的代码。所用语句是

```
if (x>y) y++;else x--;
```

和

```
while (x<y) y -= x;
```

在两个例子中x和y都是局部整型变量。

Borland C编译器为这个if语句产生了如下的80×86汇编代码。x被放在寄存器bx中，y被放在寄存器dx中：

```
        cmp     bx,dx
        jle     short @1@86
        inc     dx
        jmp     short @1@114
@1@86:
        dec     bx
@1@114:
```

这个代码使用了同图8-1一样的顺序组织。但是请读者注意到这个代码并不计算表达式x<y的实际逻辑值而只是直接使用条件代码。

Borland C编译器产生的while语句代码如下：

```
        jmp     short @1@170
@1@142:
        sub     dx,bx
@1@170:
        cmp     bx,dx
        jl      short @1@142
```

以上运用了不同于图8-2的顺序组织。由于将这个测试放在了最后，所以一个初始的无条件的转移转就到这个测试。

4) **函数定义和调用** 将使用的是C函数定义：

```
int f( int x, int y)
{ return x+y+1;}
```

和一个相应的调用

```
f(2+3, 4)
```

(这些在8.5.1节中已用过。)

首先考虑Borland 编译器对调用**f(2+3,4)**生成的代码：

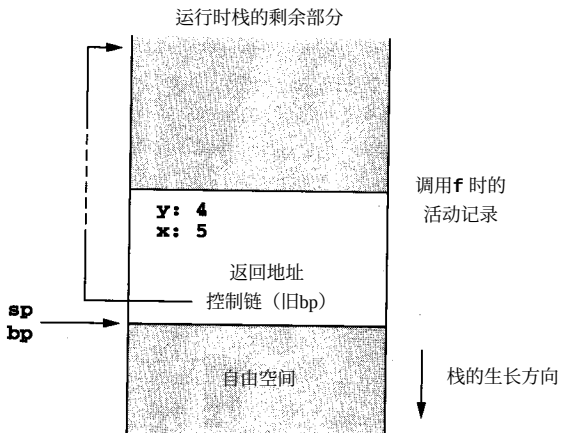
```
mov ax,4
push ax
mov ax,5
push ax
call near ptr _f
pop cx
pop cx
```

注意参数是怎样按相反顺序压入栈的。首先是4，然后再是5(因为5=2+3是一个常量表达式，所以编译器预先计算了它的值)。也要注意到调用者有责任在调用后从栈中清除参数。这就是为什么在调用后会有两个**pop**指令(目标寄存器**cx**被用作垃圾回收：不再使用弹出的值)的原因。调用本身应该是自解释的：名字**_f**在源名字前有一个下划线，这是C编译器的典型约定，**near ptr**声明表示函数在同一段中(因此仅需2字节的地址)，最后注意，80×86上的**call**指令自动将返回地址压入栈(由被调用函数执行的对应的**ret**指令将它弹出)。

现在考虑由Borland C编译器的生成的，函数**f**定义语句的代码：

```
_f      proc    near
        push    bp
        mov     bp,sp
        mov     ax,word ptr [bp+4]
        add     ax,word ptr [bp+6]
        inc     ax
        jmp     short @1@58
@1@58:
        pop     bp
        ret
_f      endp
```

这段代码中的许多指令都是一目了然的。由于调用者除了计算和压入参数外没有做活动记录的构造，这段代码必须在函数体代码被执行前完成它的构造，这是第 2、第3个指令的任务，它们保存控制链(**bp**)到栈中，然后将**bp**设成现在的**sp**。经过这样操作，栈中内容如下：



栈中的返回值在控制链(旧的**bp**)和由调用者**call**指令产生的参数之间。因此，旧的**bp**在栈顶，返回值在**bp+2**(在这个例子中地址是2字节的)。参数**x**在**bp+4**中，参数**y**在**bp+6**。**f**的函数体对应于如下的代码：

```
mov ax,word ptr [bp+4]
add ax,word ptr [bp+6]
inc ax
```

把**x**装进**ax**，再加**y**到**ax**中，然后再把**ax**中内容加1。

最后，代码执行一个转移(尽管这里并不必需要，但总也要产生，这是因为在函数中嵌的**return**语句会需要它)，从这个栈中恢复旧的**bp**，然后再返回到调用者。留在**ax**中的返回值对调用者来说是可用的。

8.6.2 Sun SparcStation的Sun 2.0 C编译器

我们重复使用前一节中的代码例子说明这个编译器的输出。同样地，以赋值语句

```
(x=x+3)+4
```

开始并且假设表达式中的变量**x**储存在局部栈框架中。

Sun C编译器产生汇编代码与Borland代码极为相似：

```
ld      [%fp+-0x4],%o1
add     %o1,0x3,%o1
st      %o1,[%fp+-0x4]
ld      [%fp+-0x4],%o2
add     %o2,0x4,%o3
```

在这些代码中，寄存器名以百分号打头，常数以字符**0x**(**x**=十六进制)打头。所以，例如**0x4**就是十六进制4(与十进制相同)。第1条指令把**x**的值(在位置**fp-4**，由于整形为4个字节长度)赋给了寄存器**o1**㊀。(注意源位置在左而目的位置在右，与Intel 80×86习惯相反。)第2条指令把常数3加到**o1**，而第3条把**o1**存到**x**的位置。最后，**x**的值再次装入，这次是到**o2**寄存器㊁，并且再加上了4，结果被放到寄放器**o3**，这是表达式的最终结果。

1) **数组引用** 表达式

```
(a[i+1]=2)+a[j]
```

以及全部局部变量都被Sun编译器翻译成下面的汇编代码(带有指令编号以便引用)：

```
(1)   add     %fp,-0x2c,%o1
(2)   ld      [%fp+-0x4],%o2
(3)   sll     %o2,0x2,%o3
(4)   mov     0x2,%o4
(5)   st      %o4,[%o1+%o3]
(6)   add     %fp,-0x30,%o5
(7)   ld      [%fp+-0x8],%o7
(8)   sll     %o7,0x2,%l0
(9)   ld      [%o5+%l0],%l1
```

㊀ 在SparcStatin中寄存器由一个小写字母后跟一个数字来表示。不同的字母表示不同的“寄存器窗口”，这会随着上下文的改变而改变并大约对应于寄存器的不同用途。本章除了一个涉及过函数调用(见8.5节)的全部例子外，不同的字母之间的区别都是看不见的。

㊁ 这个步骤没有出现在Borland代码中，并很容易优化掉。参见8.9节。

```
(10)   mov      0x2,%l2
(11)   add      %l2,%l1,%l3
```

这段代码执行的计算受此结构中整形为 4个字节的因素的影响。因此， **i**的位置是**fp-4** (在代码中写为**%fp+-0x4**)，**j**的位置是**fp-8** (**%fp+-0x8**)，而**a**的基地址是**fp-48** (由于十进制48 = 十六制30，写成**%fp+-0x30**)。指令1将**a**的基地址装入到寄存器**o1**，此地址已修改为下标**i+1**中的常数1减去4个字节(如同Borland代码) (2c hex = 44 = 48 - 4)。指令2将**i**的值装入到寄存器**o2**中。指令3把**o3**乘以4 (左移2位)并把结果存放在**o3**中。指令4把常数2装入寄存器**o4**，而指令5将其存入地址**a[i+1]**。指令6计算**a**的基地址并存入**o5**，指令7将**j**的值装入**o7**。而指令8将其乘上4，把结果存到寄存器**l0**，最后，指令9把**a[j]**的值装入**l1**，指令l0重新将2装入到**l2**，指令11将它们相加，把结果(也就是表达式的最终结果)存入寄存器**l3**。

2) **指针和域引用** 考虑与前面相同的Sun例子(参见第8.6.1节的“指针和域引用”部分)。数据类型大小如下：整形变量占4个字节，字符变量占1个字节，指针占4个字节。其实所有的变量(包括结构域)仅分配了4字节。这样，变量**x**占了12字节，变量**p**占了4字节，而**i**、**c**和**j**的偏移分别为0、4和8，这同**val**、**lchild**和**rchild**的偏移一样。编译器在活动记录中定位**x**和**p**：**x**定位在**fp-0xc** (hex c=12)，**p**定位在**fp-0x10** (hex 10=16)。

为赋值语句

```
x.j =x.i;
```

产生的代码

```
ld      [%fp+-0xc], %o1
st      %o1, [%fp+-0x4]
```

这段代码把**x.i**的值装入寄存器**o1**，然后再存储到**x.j** (请注意，**x.j** =-12+8=4的偏移是静态计算的)。

指针赋值

```
p->lchild=p;
```

的目标代码为

```
ld [%fp+-0x10], %o2
ld [%fp+-0x10], %o3
st %03, [%o2+0x4]
```

在这里**p**的值装载到寄存器**o2**和**o3**，其中的一个拷贝(**o2**)用作存储另一个拷贝到**p->lchild**位置的基地址。最后，赋值语句

```
p = p->rchild;
```

的目标代码

```
ld       [%fp+-0x10],%o4
ld       [%o4 +0x8],%o5
st       %o5,[%fp+-0x10]
```

在这段代码中**p**的值装入寄存器**o4**，然后作为基地址以装入**p->rchild**的值。最后一条指令将这个值存入**p**的位置。

3) **If和While 语句** 我们如同Borland编译器一样展示Sun SparcStation C编译器为同一段典型控制语句产生的代码

```
if (x>y) y++; else x--;
```

和

```
while (x<y) y -= x;
```

x和**y**为局部整形变量。

Sun SparcStation编译器为if语句产生下面代码，**x**和**y**定位在局部动态记录中偏移为-4和-8：

```
        ld      [%fp+-0x4],%o2
        ld      [%fp+-0x8],%o3
        cmp     %o2,%o3
        bg      I16
        nop
        b       L15
        nop
    L16:
        ld      [%fp+-0x8],%o4
        add     %o4,0x1,%o4
        st      %o4,[ %fp+-0x8]
        b       L17
        nop
    L15:
        ld      [%fp+-0x4],%o5
        sub     %o5,0x1,%o5
        st      %o5,[ %fp+-0x4]
    L17:
```

这里的**nop**指令之所以跟随在每个分支之后是因为Sparc是流水线的（分支延迟且下一条指令总是在分支生效前执行)。请注意后续的组织与图8-10相反，真状态放在虚假的状态之后。

Sun编译器为while循环产生的代码为

```
        ld      [%fp+-0x4],%o7
        ld      [%fp+-0x8],%10
        cmp     %o7,%10
        bl      L21
        nop
        b       L20
        nop
    L21:
    L18:
        ld      [%fp+-0x4],%11
        ld      [%fp+-0x8],%12
        sub     %12,%11,%12
        st      %12,[ %fp+-0x8]
        ld      [%fp+-0x4],%13
        ld      [%fp+-0x8],%14
        cmp     %13,%14
        bl      L18
        nop
        b       L22
        nop
    L22:
    L20:
```

除了测试代码也在开始处复制之外，代码使用类似Borland编译器的安排。此外在代码结尾处，

“什么也不做”分支(到标号**L22**)会被优化步骤轻易删除。

4) **函数定义与调用** 我们使用与前面相同的定义

```
int f ( int x, int y )
{ return x+y+1; }
```

和同样的调用

```
f (2+3,4)
```

Sun编译器产生的调用代码为：

```
mov      0x5, %o0
mov      0x4, %o1
call     _f, 2
```

为定义**f**产生的代码为

```
_f:
     !#PROLOGUE# 0
     sethi    %hi(LF62),%gl
     add      %gl,%lo(LF62),%gl
     save     %sp,%gl,%sp
     !#PROLOGUE# 1
     st       %i0,[%fp+0x44]
     st       %i1, [%fp+0x48]
L64:
     .seg     "text"
     ld       [%fp+0x44],%o0
     ld       [%fp+0x48] ,%o1
     add      %o0,%o1,%o0
     add      %o0,0x1,%o0
     b        LE62
     nop
LF62:
     mov      %o0,%i0
     ret
     restore
```

我们将不详细讨论这段代码，而只给出以下的说明：首先，调用通过寄存器 **o0**和**o1**而不是用栈来传递参数。调用使用数字**2**指出用于这个目的寄存器数。调用还执行了一些簿记功能，这些都不再深入讨论，除了指出调用之后“ o ”寄存器(如**o1**和**o2**)变成“ i ”寄存器(例如**i0**和**i1**)(调用完成后“ i ”寄存器又变回“ o ”寄存器)[⊖]。

定义**f**的代码也以一些完成调用序列的簿记指令(在**!#PROLOGUE#**注释之间)开始，这些也不进一步讨论了。代码然后将参数值存入栈中(对于“ i ”寄存器，与调用者的“ o ”寄存器等同)。在返回值计算完之后，将其存入寄存器**i0**(返回后就可以寄存器**o0**访问)。

8.7 TM：简单的目标机器

本节之后将展示TINY语言的代码生成器。为了使这成为有意义的工作，我们产生的目标

⊖ 不规范地说“ o ”代表“ output ”，“ i ”代表“ input ”，这种寄存器在调用前后的变换由 Sparc结构的寄存器窗口(register window) 机制执行。

代码可直接用于易于模拟的简单机器。这个机器称为TM(Tiny Machine)。附录C提供了TM模拟器的完整的C程序，它可以用来运行TINY代码生成器产生的代码。本节描述完整的TM结构和指令集以及附录C中的模拟器。为了便于理解，我们用C代码片段辅助说明，而TM指令本身总是以汇编代码而不是二进制或十六进制形式给出(模拟器总是只读入汇编代码)。

8.7.1 Tiny Machine的基本结构

TM由只读指令存储区、数据区和8个通用寄存器构成。它们都使用非负整数地址且以0开始。寄存器7为程序记数器，它也是唯一的专用寄存器，如下面所描述的那样。C的声明：

```
#define IADDR_SIZE ...
     /* size of instruction memory */
#define DADDR_SIZE...
     /* size of data memory */
#define NO_REGS 8 /* number of registers */
#define PC_REG 7

Instruction iMem[IADDR_SIZE];
int dMem[DADDR_SIZE];
int reg[NO_REGS];
```

将用于描述。

TM执行一个常用的取指-执行循环

```
do
  /* fetch */
  current Instruction = iMem [reg[pcRegNo]++];
  /* execute current instruction */
  ...
while (!(halt||error));
```

在开始点，Tiny Machine将所有寄存器和数据区设为0，然后将最高正规地址的值(名为**DADDR_SIZE-1**)装入到**dMem[0]**中。(由于程序可以知道在执行时可用内存的数量，所以它允许将存储器很便利地添加到TM上)。TM然后开始执行**iMem[0]**指令。机器在执行到**HALT**指令时停止。可能的错误条件包括 **IMEM_ERR**(它发生在取指步骤中若 **reg[PC_REG]**< 0或**reg[PC_REG]**≥**IADDR_SIZE**时)，以及两个条件**DMEM_ERR**和**ZERO-DIV**，它们发生在下面描述的指令执行中。

程序清单8-12给出TM的指令集以及每条指令效果的简短描述。基本指令格式有两种：寄存器，即RO指令。寄存器-存储器，即RM指令。寄存器指令有如下格式：

```
opcode r, s, t
```

程序清单8-12 Ting Machine 的完全指令集

RO **指令**	
格式	***opcode r,s,t***
操作码	效果
HALT	停止执行(忽略操作数)
IN	**reg[*r*]** ← 从标准读入整形值(***s***和***t***忽略)
OUT	**reg[*r*]** → 标准输出(***s***和***t***忽略)

ADD	reg[r] = reg[s] + reg[t]
SUB	reg[r] = reg[s] - reg[t]
MUL	reg[r] = reg[s] * reg[t]
DIV	reg[r] = reg[s] / reg[t](可能产生ZERO_DIV)

RM 指令

格式 *opcode r,d(s)*

(*a*=*d*+reg[*s*]; 任何对dmem[*a*]的引用在*a*<0或*a*≥DADDR-SIZE时产生DMEM-ERR)

操作码 效果

LD	reg[r] = dMem[a](将*a*中的值装入*r*)
LDA	reg[r] = *a* (将地址*a*直接装入*r*)
LDC	reg[r] = *d* (将常数*d*直接装入*r*, 忽略*s*)
ST	dMem[*a*] = reg[r](将*r*的值存入位置*a*)
JLT	if (reg [r]<0) reg [PC_REG] =*a* (如果*r*小于零转移到*a*，以下类似)
JLE	if (reg [r]<=0 reg [PC_REG] =*a*
JGE	if (reg [r]>0) reg [PC_REG] =*a*
JGT	if (reg [r]>0) reg [PC_REG] =*a*
TEQ	if (reg [r]==0) reg [PC_REG] =*a*
JNE	if (reg [r]!=0) reg [PC_REG] =*a*

这里操作数*r*、*s*、*t*为正规寄存器(在装入时检测)。这样这种指令有3个地址，且所有地址都必须为寄存器。所用算术指令被限制到这种格式，以及两个基本输入 /输出指令。

一条寄存器-存储器指令有如下格式：

opcode r,d(s)

在代码中*r*和*s*必须为正规的寄存器(装入时检测)，而*d*为代表偏移的正、负整数。这种指令为两地址指令，第1个地址总是一个寄存器，而第2个地址是存储器地址*a*，用*a* = *d* + reg[s]给出，这里*a*必须为正规地址(0≤*a*＜DADDR_SIZE)。如果*a*超出正规的范围，在执行中就会产生DMEM_ERR。

RM指令包括对应于3种地址模式的3种不同的装入指令：“装入常数”(LDC)，“装入地址”(LDA)和“装入内存”(LD)。另外，还有1条存储指令和6条转移指令。

在RD和RM中，即使其中一些可能被忽略，所有的 3个操作数也都必须表示出来。这是由于简化了装载器，它仅区分两类指令(RO和RM)而不允许在一类中有不同的指令格式⊖。

程序清单8-12和到此为止的TM讨论表示了完整的TM结构。特别需要指出的是：除了 pc之外没有特殊寄存器(没有sp或fp)，其他再也没有硬件栈和其他种类的要求。因此， TM的编译器必须完全手工维护运行时环境组织。虽然这看起来有点不切实际，但它有所有操作在需要时必须显式产生的优点。

由于指令集是最小的，这就需要一些说明来指出它们如何被用来构造大部分标准程序语言操作(实际上，这个机器如果不去满足少量高级语言的话已经足够了)。

1) 算术运算中目标寄存器、IN以及装入操作先出现，然后才是源寄存器。这类似于80×86而不同于Sun SparcStation。对用于目标和源的寄存器没有限制：特别地，目标和源寄存器可以相同。

⊖ 这也使代码生成更容易了，因为对两类指令只需两种例程。

2) 所有的算术操作都限制在寄存器之上。没有操作 (除了装入和存储操作)是直接作用于内存的。这一点TM与诸如Sun Sparc Station的RISC机器相似。另一方面，TM只有8个寄存器，而大部分RISC处理器有至少32个[⊖]。

3) 没有浮点操作和浮点寄存器。为TM增加浮点操作和寄存器的协处理器并不困难。在普通寄存器和内存之间转换浮点数时要小心一些。请参阅练习。

4) 与其他一些汇编代码不同，这里没有在操作数中指定地址模式的能力 (比如**LD#1**表示立即模式，或**LD @a**表示间接)。作为代替的是对应不同模式的不同指令：**LD**是间接，**LDA**是直接，而**LDC**是立即。实际上TM只有很少的地址选择。

5) 在指令中没有限制使用pc。实际上由于没有无条件转移指令，因此必须由将pc作为**LDA**指令的目标寄存器来模拟：

```
LDA 7,d(s)
```

这条指令效果为转移到位置***a*** = ***d*** + **reg[s]**。

6) 这里也没有间接转移指令，不过也可以模拟，如果需要也可以使用**LD**指令，例如，

```
LD 7,0(1)
```

转移到由寄存器1指示地址的指令。

7) 条件转移指令(**JLT**等)可以与程序中当前位置相关，只要把pc作为第2个寄存器，例如

```
JEQ 0,4(7)
```

导致TM在寄存器0的值为0时向前转移5条指令，无条件转移也可以与pc相关，只要pc两次出现在**LDA**指令中，这样，

```
LDA 7,-4(7)
```

执行无条件转移回退3条指令。

8) 没有过程和**JSUB**指令，作为代替，必须写出

```
LD 7,D(S)
```

其效果是转移到过程，其入口地址为**dMem[d+reg[s]]**。当然，要记住先保存返回地址，类似于执行

```
LDA 0,1(7)
```

它将当前pc 值加1放到**reg[0]**(那是我们要返回地地方，假设下一条指令是实际的转移到过程)。

8.7.2 TM模拟器

这个模拟器接受包含上面所述的TM指令的文本文件， 并有以下约定：

1) 忽略空行。

2) 以星号打头的行被认为是注释而忽略。

3) 任何其他行必须包含整数指示位置后跟冒号再接正规指令。任何指令后的文字都被认为是注释而被忽略掉。

TM模拟器没有其他特征了——特别是没有符号标号和宏能力。程序清单 8-13为一个手写的TM程序对应于程序清单8-1的TINY程序。严格地说，程序清单8-13中代码尾部不需要**HALT**

⊖ 由于TM寄存器的数量增加很容易，因为基本代码生成无需如此，所以我们不必这样做。见本章最后的练习。

指令，由于TM模拟器在装入程序之前已设置了所有指令位置直到**HALT**。然而，将其作为一个提醒是很有用的——以及作为转移退出程序的目标。

程序清单8-13　显示约定的程序

```
* This program inputs an integer, computes
* its factorial if it is positive,
* and prints the result
0:     IN   0, 0, 0      r0 = read
1:     JLE  0, 6 (7)     if 0 < r0 then
2:     LDC  1,1,0          r1 = 1
3:     LDC  2, 1, 0        r2=1
                         * repeat
4:     MUL 1, 1, 0          r1 = r1*r0
5:     SUB 0, 0, 2          r0 = r0-r2
6:     JNE  0, -3 (7)     until r0 == 0
7:     OUT  1, 0, 0        write r1
8:     HALT 0, 0, 0      halt
* end of program
```

此外没有必要如程序清单 8-13中那样将位置升序排列。每个输入行足够指出“ 将这条指令存在这个位置 ”：如果TM程序打在卡片上，那么在掉到地板上之后再读入还是工作得很完美。TM模拟器的这种特性可能引起阅读程序时的混淆，为了在没有符号标号的情况下反填转移，代码要能不必回翻代码文件就能完成反填。例如，代码生成器可能产生程序清单 8-13代码如下：

```
0:  IN   0,0,0
2:  LDC  1,1,0
3:  LDC  2,1,0
4:  MUL  1,1,0
5:  SUB  0,0,2
6:  JNE  0,-3(7)
7:  OUT  1,0,0
1:  JLE  0,6(7)
8:  HALT 0,0,0
```

因为在知道if语句体后面的位置之前，向前转移的指令1没法生成。

如果程序清单8-13的程序在文件**fact.tm**中，那么这个文件可以用下面的示例任务装入并执行(如果没有扩展名，TM模拟器自动假设**.tm**)：

```
tm fact
TM simulation ( enter h for help ) ...
Enter command: g
Enter value for IN instruction: 7
OUT instruction prints: 5040
HALT: 0, 0, 0
Halted
Enter command: q
Simulation done
```

g命令代表“ go ”，它表示程序用当前pc中的内容(装入之后为0)开始执行到看到**HALT**指令为止。完整的模拟器命令可以用**h**命令得到，将打出以下列表：

```
Commands are:
   s(tep <n>     Execute n ( default 1 ) TM instructions
   g(o           Execute TM instructions until HALT
   r(egs         print the contents of the registers
   i(Mem <b <n>> Print n iMem locations strarting at b
   d(Mem <b <n>> Print n dMem locations strarting at b
   t(race        Toggle instructions trace
   p(rint        Toggle print of total instructions executed ('go' only)
   c(lear        Reset simulator for new execution of program
   h(elp         Cause this list of commands to printed
   q(uit         Terminate the simulation
```

命令中的右括号指示命令字母衍生的记忆法 (使用多个字母也可以，但模拟器只检查首字母)。尖括号**< >**表示可选参数。

8.8 TINY语言的代码生成器

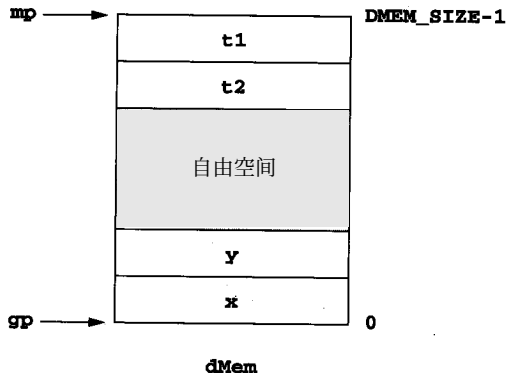
现在要描述TINY语言的代码生成器。我们假设读者熟悉 TINY编译器的先前步骤，特别是第3章中描述的分析器产生的语法树结构、第 6章描述的符号表构造，以及第 7章的运行时环境。

本节首先描述TINY代码生成器和TM的接口以及代码生成必需的实用函数。然后再说明代码生成的步骤。接着，描述 TINY编译器和TM模拟器的结合。最后讨论全书使用的示例 TINY程序的目标代码。

8.8.1 TINY代码生成器的TM接口

一些代码生成器需要知道的有关 TM的信息已封装在文件 **code.h** 和**code.c** 中，在附录B中有该程序，分别是第 1600行到第1685行和第1700行到第1796行。此外还在文件中放入了代码发行函数。当然，代码生成器还是要知道 TM指令的名字，但是这些文件分离了指令格式的详细说明和目标代码文件的位置以及运行时使用特殊寄存器。 **code.c**文件完全可以将指令序列放到特别的**iMem**位置，而代码生成器就不必追踪细节了。如果 TM装载器要改进，也就是说允许符号标号并去掉数字编号，那么将很容易将标号生成和格式变化加入到**code.c**文件中。

现在我们复习一下**code.h**文件中的常数和函数定义。首先是寄存器值的定义(1612，1617，1623，1626和1629行)。明显地，代码生成器和代码发行实用程序必须知道 pc。另外还有TINY语言的运行时环境，如前一节所述，将数据存储时的顶部分配给临时存储 (以栈方式)而底部则分配给变量。由于TINY中没有活动记录(于是也就没有fp)(没有作用域和过程调用)，变量和临时存储的位置可认为是绝对的。然而， TM机的**LD**操作不允许绝对地址，而必须有一个寄存器基值来计算存储装入的地址。这样我们分配两个寄存器，称为 mp(内存指针)和gp(全程指针)来指示存储区的顶部和底部。 mp将用于访问临时变量，并总是包含最高正规内存位置，而 gp用于所有命名变量访问，并总是包含 0。这样由符号表计算的绝对地址可以生成相对 gp的偏移来使用。例如，如果程序使用两个变量 **x**和**y**，并有两个临时值存在内存中，那么 **dMem**将如下所示：



在本图中，**t1**的地址为**0(mp)**，**t2**为**-1(mp)**，**x**的地址为**0(gp)**，而**y**为**1(gp)**。在这个实现中，gp是寄存器5，mp是寄存器6。

另两个代码生成器将使用的寄存器是寄存器0和1，称之为“累加器”并命令名为ac和ac1。它们被当作相等的寄存器来使用。通常计算结果存放在ac中。注意寄存器2、3和4没有命名(且从不使用1)。

现在来讨论7个代码发行函数，原型在**code.h**文件中给出。如果**TraceCode**标志置位，**emitComment**函数会以注释格式将其参数串打印到代码文件中的新行中。下两个函数**emitRO**和**emitRM**为标准的代码发行函数用于RO和RM指令类。除了指令串和3个操作数之外，每个函数还带有1个附加串参数，它被加到指令中作为注释(如果**TraceCode**标志置位)。

接下来的3个函数用于产生和反填转移。**emitSkip**函数用于跳过将来要反填的一些位置并返回当前指令位置且保存在**code.c**内部。典型的应用是调用**emitSkip(1)**，它跳过一个位置，这个位置后来会填上转移指令，而**emitSkip(0)**不跳过位置，调用它只是为了得到当前位置以备后来的转移引用。函数**emitBackup**用于设置当前指令位置到先前位置来反填，**emitRestore**用于返回当前指令位置给先前调用**emitBackup**的值。典型地，这些指令在一起使用如下：

```
emitBackup(savedLoc) ;
/* generate backpatched jump instruction here */
emitRestore() ;
```

最后代码发行函数(**emitRM_Abs**)用来产生诸如反填转移或任何由调用**emitSkip**返回的代码位置的转移的代码。它将绝对代码地址转变成pc相关地址，这由当前指令位置加1(这是pc继续执行的地方)减去传进的位置参数，并且使用pc做源寄存器。通常地，这个函数仅用于条件转移，比如**JEQ**或使用**LDA**和pc作为目标寄存器产生无条件转移，如前一小节所述的那样。

这样就描述完了TINY代码生成实用程序，我们来看一看TINY代码生成器本身的描述。

8.8.2 TINY代码生成器

TINY代码生成器在文件**cgen.c**中，其中提供给TINY编译器的唯一接口是**CodeGen**，其原型为：

```
void CodeGen (void);
```

在接口文件**cgen.h**中给出了唯一的定义。附录B中有完整的**cgen.c**文件，参见第1900行到第2111行。

函数**CodeGen**本身(第2095行到第2111行)所做的事极少：产生一些注释和指令(称为标准

序言(standard prelude))、设置启动时的运行时环境，然后在语法树上调用 **cGen**，最后产生**HALT**指令终止程序。标准序言由两条指令组成：第1条将最高正规内存位置装入mp寄存器(TM模拟器在开始时置0)。第2条指令清除位置0(由于开始时所有寄存器都为0，gp不必置0)。

函数**cGen**(第2070行到第2084行)负责完成遍历并以修改过的顺序产生代码的语法树，回想TINY语法树定义给出的格式：

```
typedef enum { StmtK, ExpK } NodeKind;
typedef enum { IfK, RepeatK, AssignK, ReadK, WriteK } StmtKind;
typedef enum { OpK, ConstK, IdK } ExpKind;

#define MAXCHILDREN 3
typedef struct treeNode
   { struct treeNode * child[MAXCHILDREN] ;
     struct treeNode * sibling;
     int lineno;
     NodeKind nodekind;
     union { StmtKind stmt; ExpKind exp; } kind;
     union { TokenType op;
             int val;
             char * name; } attr;
     ExpType type;
   } TreeNode;
```

这里有两种树节点：句子节点和表达式节点。如果节点为句子节点，那么它代表5种不同TINY语句(if、repeat、赋值、read或write)中的一种，如果节点为表达式节点，则代表3种表达式(标识符、整形常数或操作符)中的一种。函数**cGen**仅检测节点是句子或表达式节点(或空)，调用相应的函数**genStmt**或**genExp**，然后在同属上递归调用自身(这样同属列表将以从左到右格式产生代码)。

函数**genStmt**(第1924行到第1994行)包含大量**switch**语句来区分5种句子，它产生代码并在每种情况递归调用**cGen**，**genExp**函数(第1997行到第2065行)也与之类似。在任意情况下，子表达式的代码都假设把值存到ac中而可以被后面的代码访问。当需要访问变量时(赋值和read语句以及标识符表达式)，通过下面操作访问符号表：

```
loc = lookup(tree->attr.name);
```

loc的值为问题中的变量地址并以gp寄存器基准的偏移装入或存储值。

其他需要访问内存的情况是计算操作符表达式的结果，左边的操作数必须存入临时变量直到右边操作数计算完成。这样操作符表达式的代码包含下列代码生成序列在操作符应用 (第2021行到第2027行)之前：

```
cGen(p1); /* p1 = left child */
emitRM("ST",ac,tmpOffset--,mp,"op: push left");
cGen(p2); /* p2 = right child */
emitRM("LD",ac1,++tmpOffset,mp,"op: load left");
```

这里的**tmpOffset**为静态变量，初始为用作下一个可用临时变量位置对于内存顶部(由mp寄存器指出)的偏移。注意**tmpOffset**如何在每次存入后递减和读出后递增。这样 **tmpOffset**可以看成是“临时变量栈”的顶部指针，对 **emifRM**函数的调用与压入和弹出该栈相对应。这在临时变量在内存中时保护它们。在以上代码之前执行实际动作，左边的操作数将在寄存器1(acl)中而右边操作数在寄存器0(ac)中。如果是算术操作的话，就产生相应的RO操作。

比较操作符的情况有少许差别。TINY语言的语法(如语法分析器中的实现，参见前面章节)仅在if语句和while语句的测试表达式中允许比较操作符。在这些测试之外也没有布尔变量或值，比较操作符可以在这些语句的代码生成内部处理。然而，这里我们用更通常的方法，它更广泛应用于包含逻辑操作与/或布尔值的语言，并将测试结果表示为0(假)或1(真)，如同在C中一样。这要求常数据0或1显式地装入ac，用转移到执行正确装载来实现这一点。例如，在小于操作符的情况中，产生了以下代码，代码产生将计算左边操作数存入寄存器 1，并计算右边操作数存入寄存器0：

```
SUB  0,1,0
JLT  0,2(7)
LDC  0,0(0)
LDA  7,1(7)
LDC  0,1(0)
```

第1条指令将左边操作数减去右边操作数，结果放入寄存器0，如果 < 为真，结果应为负值，并且指令**JLT 0,2(7)**将导致跳过两条指令到最后一条，将值1装入ac，如果 < 为假，将执行第3条和第4条指令，将0装入ac然后跳过最后一条指令(回忆TM的描述，**LDA**使用pc为寄存器引起无条件转移)。

我们将以if-语句(第1930行到第1954行)的讨论来结束TINY代码生成器的描述。其余的情况留给读者。

代码生成器为if语句所做的第1个动作是为测试表达式产生代码。如前所述测试代码，在假时将0存入ac，真时将1存入。生成代码接下来要产生一条**JEQ**到if语句的else部分。然而这些代码的位置当前是未知的，这是因为 then部分的代码还要生成。因此，代码生成器用 emitSkip来跳过后面语句并保存位置用于反填：

```
savedLoc1 = emitSkip(1) ;
```

代码生成继续处理if算语句的then部分。之后必须无条件转移跳过else部分。同样转移位置未知，于是这个转移的位置也要跳过并保存位置：

```
savedLoc2 = emitSkip(1) ;
```

现在，下一步是产生else部分的代码，于是当前代码位置是正确的假转移的目标，要反填到位置**savedLoc1**。下面的代码处理之：

```
currentLoc = emitSkip (0) ;
emitBack up(savedLoc1) ;
emitRM_Abs("JEQ",ac,currentLoc,"if: jmp to else") ;
emitRestors() ;
```

注意**emitSkip(0)**调用是如何用来获取当前指令位置的，以及 **emitRM_Abs**过程如何用于将绝对地址转移变换成pc相关的转移，这是**JEQ**指令所需的。之后就可以为else部分产生代码了，然后用类似的代码将绝对转移(**LDA**)反填到**savedLoc2**。

8.8.3 用TINY编译器产生和使用TM代码文件

TINY代码生器可以合谐地与 TM模拟器一起工作。当主程序标志 **NO_PARSE**、**NO_ANALYZE**和**NO_CODE**都置为假时，编译器创建**.tm**后缀的代码文件(假设源代码中无错误)并将TM指令以TM模拟器要求的格式写入该文件。例如，为编译并执行**sample.tny**程序，只要发出下面命令：

```
tiny sample
<listing produced on the standard output>
tm sample
<execution of the tm simulator>
```

为了跟踪的目的，有一个**TraceCode**标志在**globals.h**中声明，其定义在**main.c**中。如果标志为**TRUE**，代码生成器将产生跟踪代码，在代码文件中表现为注释，指出每条指令或指令序列在代码生成器的何处产生以及产生原因。

8.8.4 TINY编译器生成的TM代码文件示例

为了详细说明代码生成是如何工作的，我们在程序清单 8-14中展示了TINY代码生成器生成的程序清单8-1中示例程序的代码，由于**TraceCode** = **TRUE**，所以也产生了代码注释。这个代码文件有42条指令，其中包括来自标准序言的两条指令。将它与程序清单 8-13中手写的程序中的漂亮指令对比，我们可以明显看出一些不够高效之处。特别地，程序清单 8-13的程序高效地使用了寄存器，除了寄存器之外没有再也用到内存。程序清单 8-14代码正相反，没有使用超过两个寄存器并执行了许多不必要的存储和装入。特别愚蠢的是处理变量值的方法，其装入只是为了再次存储到临时变量中，如下所示：

```
16:     LD   0,1(5) load id value
17:     ST   0,0(6) op: push left
18:     LD   0,0(5) load id value
19:     LD   1,0(6) op: load left
```

这可以用两条指令代替：

```
LD  1,1(5) load id value
LD  0,0(5) load id value
```

它们具有同样的效果。

更多潜在的不足将由生成的测试和转移代码引起。不完整的笨例子是指令：

```
40:   LDA 7,0(7) jmp to end
```

这是一个煞费苦心的**NOP**(一条“无操作”指令)。

然而，程序清单8-14的代码有一个重要理由：它是正确的。在匆忙提高生成代码的效率时，编译编写者忘记了这个原则并允许生成只有效率却不总是能正确执行的代码。这种行为如果不做好文档并预测可能会导致灾难。

由于学习所有改进编译器代码产生的方法超出了本书的范围，本章最后的两节将仅考察可以做出这些改进的主要范围和实现它们的技术，并简要说明某些方法如何用于 TINY代码生成器来改进生成的代码。

程序清单8-14 程序清单8-1示例程序的代码输出

```
* TINY Compilation to TM Code
* File: sample.tm
* Standard prelude:
   0:     LD 6,0(0)     load maxaddress from location 0
   1:     ST 0,0(0)     clear location 0
* End of standard prelude.
   2:     IN 0,0,0      read integer value
```

```
   3:     ST 0,0(5)     read: store value
* -> if
* -> Op
* -> Const
   4:    LDC 0,0(0)     load const
* <- Const
   5:     ST 0,0(6)     op: push left
* -> Id
   6:     LD 0,0(5)     load id value
* <- Id
   7:     LD 1,0(6)     op: load left
   8:    SUB 0,1,0      op <
   9:    JLT 0,2(7)     br if true
  10:    LDC 0,0(0)     false case
  11:    LDA 7,1(7)     unconditional jmp
  12:    LDC 0,1(0)     true case
* <- Op
* if: jump to else belongs here
* -> assign
* -> Const
  14:    LDC 0,1(0)     load const
* <- Const
  15:     ST 0,1(5)     assign: store value
* <- assign
* -> repeat
* repeat: jump after body comes back here
* -> assign
* -> Op
* -> Id
  16:     LD 0,1(5)     load id value
* <- Id
  17:     ST 0,0(6)     op: push left
* -> Id
  18:     LD 0,0(5)     load id value
* <- Id
  19:     LD 1,0(6)     op: load left
  20:    MUL 0,1,0      op *
* <- Op
  21:     ST 0,1(5)     assign: store value
* <- assign
* -> assign
* -> Op
* -> Id
  22:     LD 0,0(5)     load id value
* <- Id
  23:     ST 0,0(6)     op: push left
* -> Const
  24:    LDC 0,1(0)     load const
* <- Const
  25:    LD 1,0(6)     op: load left
  26:    SUB 0,1,0      op -
```

```
* <- Op
  27:     ST 0,0(5)     assign: store value
* <- assign
* -> Op
* -> Id
  28:     LD 0,0(5)     load id value
* <- Id
  29:     ST 0,0(6)     op: push left
* -> Const
  30:    LDC 0,0(0)     load const
* <- Const
  31:     LD 1,0(6)     op: load left
  32:    SUB 0,1,0      op = =
  33:    JEQ 0,2(7)     br if true
  34:    LDC 0,0(0)     false case
  35:    LDA 7,1(7)     unconditional jmp
  36:    LDC 0,1(0)     true case
* <- Op
  37:    JEQ 0,-22(7)   repeat: jmp back to body
* <- repeat
* -> Id
  38:     LD 0,1(5)     load id value
* <- Id
  39:    OUT 0,0,0      write ac
* if: jump to end belongs here
  13:    JEQ 0,27(7)    if: jmp to else
  40:    LDA 7,0(7)     jmp to end
* <- if
* End of execution.
  41:   HALT 0,0,0
```

8.9 代码优化技术考察

自从50年代出现第1个编译器以来，生成的代码质量一直受到重视。质量由目标代码的速度和大小来衡量，虽然通常速度更重要，现代编译器表明代码质量受编译过程中某些点的处理过程影响，这一系列步骤包括收集源代码信息然后利用这些信息执行代码改进变换 (code improve transformation)改进代码中的数据结构，许多年来开发了大量的提高代码质量的技术，称为代码优化技术(code optimization techniques)。这个术语有些误导，由于仅在一些很特殊的情况下这些技术才能产生数学上的优化代码。不过这个名称很常用，我们还是继续使用它吧。

因为存在着许多代码优化技术，我们只能大概浏览一些最重要和最广泛使用的，甚至对这些也不给出细节和实现。本章有更多信息的参考书目。必须认识一点，编译器编写者不能希望包含每一种单个优化技术，而要根据语言的实际情况作出判断。哪种技术可以在增加最小编译器复杂度的情况下大大提高代码质量，有许多描述需要极复杂实现的优化技术的论文，而这些技术仅产生了相对有一点改进的目标代码 (也就是减少一点运行时间)。经验通常显示一些基本方法虽然看起来是简单方式应用却可以带来重大提高，有时甚至减少一半或更多执行时间。

衡量某个优化技术的实现是否太复杂依赖实际代码改进的代价，不仅要确定实现的数据结构和额外代码开销的复杂度还要考虑优化步骤对编译器本身速度的影响。我们所学的所有分析

技术都与编译程序大小成正比。有些优化技术可能相对程序大小的平方或3次方增加编译时间，于是完全优化编译一个大程序时要延长好几分钟(最差时要几个小时)。这将导致用户回避使用优化(或不再使用这个编译器)，用于实现优化的时间是巨大的浪费。

下面几节我们将先描述优化的主要来源然后介绍几种经典优化方法。接着是几种重要技术和实现的主要数据结构。自始至终都将给出简单示例来说明讨论的内容。下一节将给出更详细的例子说明讨论的一些技术如何应用于前面章节的TINY代码生成器，以及实现方法的建议。

8.9.1 代码优化的主要来源

下面将列出某些代码生成器不能产生好代码的地方，粗略地减少“代价”，也就是在这些地方代码可以获得多大改进。

1) **寄存器分配** 合理使用寄存器是高效代码的最重要特征。由于历史原因，可用寄存器很少——通常只8个或16个，这其中包括了特殊用途寄存器，如pc、sp和fp。这使得寄存器合理分配很困难，因为变量和临时变量竞争寄存器空间很激烈。这种情况仍存在于一些处理器中，特别是微处理器，解决这一问题的一个方法是可以增加直接在内存执行的操作的数量和速度，这样编译器一旦耗尽寄存器空间，就可以避免存储寄存器值到临时变量以释放寄存器值然后再装新值(称为寄存器溢出(register spill)操作)的代价。另一个方法(称为RISC方法)减少在内存直接执行的操作的数量(常常为0)，但同时增加可用寄存器到32、64或128。在这种结构中，合理配寄存器变得至关重要，因为它可以保存全部或大部分全程变量在寄存器中。这种结构中分配寄存器失误的代价是频繁装载和存入值。同时因为有了很多可用的寄存器，寄存器分配工作也变得简单了。这样，提高代码质量的重要努力应着眼于合理分配寄存器。

2) **不必要操作** 代码改进的第2个主要来源是避免产生冗余或不必要的操作代码。这种优化从很简单的搜索局部代码到分析整个程序的语法特性各不相同。确认这些操作的方法很多，也有许多对应技术。这种优化的一种典型例子是代码中重复出现的表达式，而且它们的值相同，可以保存的第1次的计算值并删除重复计算(称为公共子表达式消除(common subexpression elimination))⊖。另一个例子是避免存储不再使用的变量或临时变量的值(这要与前面的优化共同进行)。

全程的优化涉及识别不可到达(unreachable)或死代码(dead code)，典型例子是使用常量标志开关调试信息：

```
#define DEBUG 0
...
if (DEBIG)
{...}
```

如果**DEBUG**设为0(如代码中所示)，那么在if语句的大括号内的代码将不可到达，于是就不必产生这段目标代码，甚至连if语句也可以省掉。另一个不可到达代码例子是不调用的过程(或只在不可到达的代码处调用)的消除，不可到达代码不总是对速度产生重大影响，不过可以真正减少目标代码大小，这是值得的，特别是当分析中很小的代价就可以识别大部分明显的情况。有时为了识别不必要操作，用代码生成器处理然后检测目标代码的冗余更容易些。一种情况是对在生成跳向表示结构控制语句时产生冗余作出预测是很困难的。这些代码包含转移到相邻的下一

⊖ 一个好的程序员可以避免公共表达式在源码中这样扩散，读者不应认为优化只是用于帮助差劲的编程者。许多来自地址计算的公共子表达式由编译器产生，并不能由好的源代码来消除它们。

条语句或转移到转移语句。转移优化(jump optimization)步骤可以去除这些不必要的转移。

3) **高代价操作** 代码生成器不只要寻找不必要操作，还要利用机会减少必要操作的代价。要用比源代码更简单实现更低的代价实现操作。典型例子用低代价操作代替算术操作，例如，乘2可以用移位操作实现。小整数据幂，比如x3，可以用连乘x*x*x实现。这种优化称为减轻强度(reduction in strength)，它可以扩展到许多方面，比如将涉及小的整数乘法替换为移位和加法(例如用2*2*x代替5*x + x ——两个移位和一个加法)。

一个相关的优化是用有关常数信息删除可能的操作或预先计算一些操作。例如，两个常数相加，比如2+3，可以由编译器计算并作常数5代替(这称为常数据合并(constant folding))。

有时有必要确定一个变量在程序局部或全程是否有恒定值，这样变换就可应用于涉及该变量的表达式(称为常量传播(constant propagation))。

有时相对昂贵的操作是过程调用，这里许多调用操作序列必须执行。现代处理器通过提供支持标准调用的硬件减少了这些代价。但是去除频繁调用小过程还是能产生可观的加速。有两种标准方法可以去掉过程调用。一个是用过程体代替过程调用 (使用合适的参数代替形式参数)。这称为过程内嵌(procedure in lining)，有时这是语言选项比如C++。另一个消除调用的方法是识别尾部递归(tail recursion)，也就是过程最后的操作是调用自身，例如，过程：

```
int gcd( int u, int v)
{ if (v==0) return u;
  else return gcd(v,u % v); }
```

是尾递归的，而过程

```
int fact( int n )
{ if (n==0) return 1;
  else return n * fact(n-1); }
```

则不是。尾递归等同过将新的调用参数赋给形式参数并转移到过程体的开始。例如尾递归过程**gcd**可以被编译器重写为等同代码：

```
int gcd( int u, int v)
{ begin:
if (v==0) return u;
else
{ int t1 = v, t2 = u%v;
  u = t1; v = t2;
  goto begin;
}
}
```

(注意代码中临时变量的微妙利用)。这个处理称为尾递归消除(tail recursion removal)。

要提及的问题是过程调用与寄存器分配有关。过程调用之前必须准备保存和恢复在调用内部使用的寄存器。如果提供的是多寄存器分配，将增加过程调用代价，因为更多的寄存器要保存和恢复。有时在寄存器分配中包含调用考虑会减少花费㊀。但这是一个普遍现象：有时优化会引起反面效果，因此必须考虑取舍。

在代码生成的最后阶段，通过使用目标机器上的特殊指令可以减少某些操作的代价。例如，许多结构包括块移动操作比单独拷贝或数组元素要快许多。还有地址计算有时可以优化，当结

㊀ Sun SparcStation中的寄存器窗口表示硬件支持过程调用的寄存器分配的一个例子。参见8.6.2节最后。

构允许几种地址模式或偏移计算结合到一条指令中时。同样，用于索引的自动增量和减量也是很实用的 (VAX结构甚至有为循环设的增量比较分支)。这些优化来自先前的指令选择(instruction selection)或机器语言使用(use of machine idioms)。

4) **预测程序行为** 为了实施前面描述的一些优化，编译器必须收集有关程序中变量、值和过程使用的信息：表达式是否重用 (可变成公共子表达式)、变量是否改变、何时改变或一直不变、过程是否调用。编译器必须在计算技术范围内作最坏的假设，收集的信息有可能产生错误的代码：变量在特定点可能恒定也可能不恒定、编译器必须假设它不恒定，这意味着必须将编译器做成不是最优化的，甚至常常对程序的行为信息利用较差。实际上对程序的分析越透彻，代码优化器可以得到的信息越多。然而，即使今天最先进的编译器也会无法发现程序中的一些可改进之处。

另一个许多编译器采用的方法是：实际运行中采集程序行为的统计信息，然后用于预测哪条分支最有可能运行、哪个过程经常调用以及哪部分代码最经常执行。这些信息可以用于调整转移结构，循环和过程代码来最小化最经常执行部分的运行时间，当然这个处理要求类似梗概编译器(profiling compiler)访问适当的数据以及(至少一部分)包含产生这些数据的指示代码的可执行代码。

8.9.2 优化分类

由于有许多优化方法和技术，有必要采用不同分类规划强调优化的不同质量以减少学习难度。两个有用的分类是在编译过程中何时可以应用优化和优化应用于程序的哪些部分。

首先我们考虑应用程序在编译过程中的时间。优化可以在编译的每阶段分别执行。例如，常数合并可以在分析时进行 (虽然通常是迟一些，这样编译器可以得到与源代码相同的表示)。另一方面，一些优化可以延迟到目标代码生成之后—检查并重写目标代码以反映优化。例如，转移优化可以以这种方式进行 (有时因为通常只观察目标代码的一小部分来进行优化，所以在目标代码上进行的优化称作为窥孔优化(peephole optimization))。

通常，主要的优化工作在中间代码生成部分、中间代码生成后或目标代码生成部分。为了使优化不依赖于目标机器的特性(称为源代码级优化(source-level optimization))，可以在依赖目标机器结构的动作(目标代码级优化(target-level optimization))之前执行。有时一种优化可以同时有源码级部分和目标码级部分。例如，在寄存器分配中，经常要计算变量引用次数并将高引用率的变量放入寄存器。这个任务又分成了一个源码级部分，这里为选择的变量分配寄存器不必知道有多少可用寄存器。然后寄存器赋值步骤依赖于目标机器为这些标记的变量分配实际的寄存器，或到称为伪寄存器(pseudoregister)的内存(万一没有可用寄存器的话)。

在不同的优化中，考虑某一优化对其他优化产生的影响相当重要。例如，应在执行不可到达代码消除之前进行传播常量操作，这是因为在测试变量被发现是常数据之后，某些代码会变得不可大到达。在偶然情况下会发生两种优化无法为对方发现进一步优化机会的阶段问题(phase problem)。如下例

```
x = 1;
...
y = 0;
...
if (y) x = 0;
...
if (x) y = 1;
```

首次常量传播会产生如下代码

```
x = 1;
...
y = 0;
...
if (0) x = 0;
...
if (x) y = 1;
```

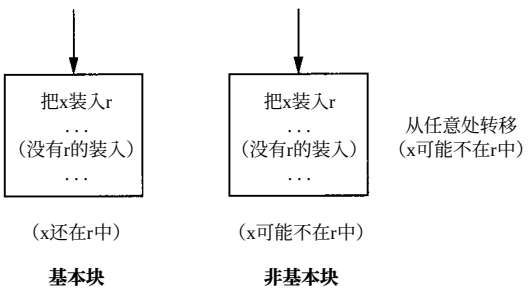
现在第1个**if**体变成不可到达；消除之

```
x = 1;
...
y = 0;
...
if (x) y = 1;
```

现在可以进行进一步的常量传播和不可到达代码消除步骤。由于这个原因，一些编译器反复执行几组优化以确保大部分可以利用的优化机会已经找到。

第2类优化考虑的是优化应用的程序范围。这类优化还分为局部 (local)、全程(global)和过程间(interprocedural)优化。局部优化定义为应用于代码的线性部分 (straight-line segment of code)的优化，也就是代码中没有跳进或跳出语句[⊖]。一个最大的线性代码序列称为基本块(basic block)。局部优化的定义限制这些基本块。扩展超出基本块，但限制在单个过程中的优化称为全程优化(因为限制在过程中，所以这不是真正的“全程”)。优化扩展出过程边界到整个程序称为过程间优化。

局部优化相对易于进行，因为代码的线性特征允许信息以简单方式下传。例如，基本块中前一条指令将一个变量值装入寄存器，只要寄存器不被再次装入，就可以在块的后面继续认为值还存在。这个结论在转移的干扰代码时就不正确了。如下图所示：



全程优化相对要困难一些，通常要求一种称为数据流分析(data flow analysis)的技术，它试图透过转移边界收集信息。过程间优化更困难，因为要涉及可能不同的参数传递机制，非局部变量访问以及计算相同调用的过程的同步信息。过程间优化的另一个复杂之处是许多过程可能分别编译最后才链接到一起。于是编译器在没有链接程序基于编译阶段收集的信息的基础上得出的优化信息时不能进行优化。由于这个原因，许多编译器只执行很基本的过程间分析或者根本就不执行。

全程优化的一个特别部分是循环。由于循环通常多次执行，应该特别注意循环内部的代码，

⊖ 过程调用表示一种特殊跳转，通常它们打断直线执行代码。然而，由于它们总是返回到相邻的下一条指令，经常可以包含在直线代码中间并由代码生成过程以后处理。

特别是减少复杂运算。典型的循环优化策略着眼于识别每次执行增长值固定的变量（也称为归纳变量(induction variable)）。这包括循环控制变量和其他依赖于循环变量的变量，选择的归纳变量可以放入寄存器，使其计算简化。这些代码重写包括从循环中删除常量计算（称为代码移动(code motion)）。实际上，重新安排代码也有利于提高基本块中的代码效率。通常，循环优化的额外任务是区分程序中的循环，然后就可以进行优化了。因为缺乏结构化控制和使用 goto实现循环，这种循环发现(loop discovery)是必要的。虽然循环发现在少数语言(如FORTRAN)中需要，但在大部分语言中，语法本身可以用来定位循环结构。

8.9.3 优化的数据结构和实现技术

一些优化可以在语法树的变换上实现。这些包括常数合并和不可到达代码消除，通过删除或以简单形式替换相应的子树来实现。用于后来优化的信息也可以在构造和遍历语法树时收集，比如引用次数和其他有用信息，保存到树的属性或符号表项中。

对于前面提到的一些优化，语法树不广泛或结构不适合于收集信息并进行优化。作为代替执行全程优化的优化器使用从中间代码构造的过程图形表示，称为流图 (flow graph)。流图的节点为基本块，边则是来自条件或非条件转移(目的是作为其他基本块的开始)。每个基本块节点包含块的中间代码序列。例如，图 8-4给出了对应程序清单8-2中间代码的流图(图中基本块标记为了以后引用)。

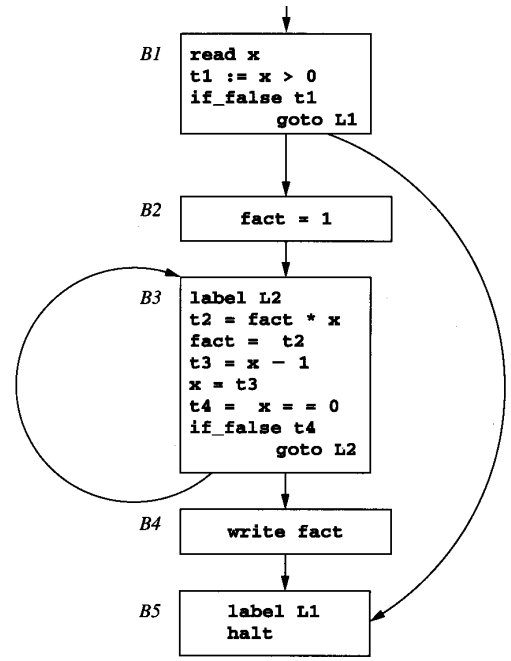


图8-4 程序清单8-2中间代码的流图

流图以及每个基本块都可以从中间代码一次遍历中构造完成。每个新的基本块识别如下[⊖]：

1) 第1条指令开始一个新基本块。

2) 每个转移目的标签开始一个新基本块。

3) 每条跟随在转移之后的指令开始一个新基本块。

可以为还没达到的向前转移构造新的空节点，并插入到符号表的标号名下，以备标号到达时查找。

流图是数据流分析的主要数据结构，积累信息用于优化。不同的信息要求对流图作不同处理，而且收集的信息各不相同，对应于不同的优化要求。因为没有足够的空间在这个概览中描述数据流分析技术的细节 (参见本章尾部“注意与参考”小节)，描述这种过程可以处理的数据的例子将是很有意义的。

计算所有变量的可达定义(reaching definition)集是一个标准的数据流分析问题，变量在每个基本块的开始处。这里一个定义是一条中间代码指令，可以设置变量值，比如赋值或读入[⊜]。例

⊖ 这个标准允许过程调用包含在基本块中。由于调用不对流图增加新路径，所以这是可行的。然后，当基本块分别处理时，调用可以在需要时分离出来特别处理。

⊜ 请不要与C定义混淆，这是一种声明。

如，图8-4中定义的变量fact是基本块*B2*(fact=1)中的一条指令以及块*B3*(**fact=t2**)中的第3条指令。让我们调用定义*d1*和*d2*。如果在块开始处变量保持定义时建立的值，则这个定义称为到达(reach)基本块。图8-4的流图中，可以建立fact的定义到达*B1*或*B2*，*d1*和*d2*到达*B3*以及只有*d2*到达*B4*和*B5*。到达定义可以用于许多优化—常数传播。例如，如果到达块唯一定义为单个常数值，那么这个变量可以用这个值来代替(至少在块中另一个定义到达前)。

流图很适用于表示有关每个过程的全程信息，不过基本块仍旧用简单代码序列表示。一旦执行数据流分析，每个基本块的代码也产生了，另一个数据结构经常被构造出来，称为基本块的**DAG** (DAG of a basic block)(DAG = directed acyclic graph直接非循环图) (DAG可以在没有构造流图时为每个基本块构造)。

DAG数据结构跟踪基本块中值和变量的计算和赋给。块中使用的来自别处的值表示为叶子节点。其上的操作和其他值表示为内部节点。赋给新值通过把目标变量或临时变量的名字附加到表示赋值的节点上来表示(416页描述了这种结构的一个特例)⊖。

例如，图8-4中基本块B3可以用图8-5中的DAG表示(基本块开头和尾部的转移的标号通常包含在DAG中)。注意在这个DAG中拷贝操作如**fact=t2**和**x=t3**不创建新节点，而只是简单地用标号**t2**和**t3**为节点加上新标签。还要注意标为**x**的叶子节点有两上父节点，其原因在于外来值**x**用于两条分别的指令中。这样重复使用同一个值也在DAG结构中表示出来了。DAG的这个特点允许它表示公共子表达式的重复使用。例如C赋值语句：

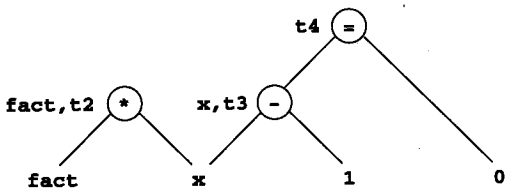


图8-5 图8-18中基本块*B3* 的DAG

```
x = (x+1)*(x+1)
```

翻译成三地址指令

```
t1 = x + 1
t2 = x + 1
t3 = t1 * t2
x = t3
```

图8-6给出这个指令序列的 DAG，显示了表达式**x+1**的重复使用。

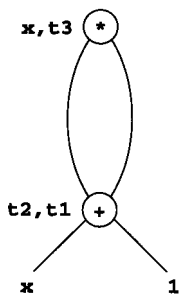


图8-6 与C赋给**x = (x+1)*(x+1)**相对应的三地址指令的DAG

基本块的DAG可以通过维护两个目录来构造。第 1个是包含变量名和常数的表，带有可返回当前赋值变量的DAG节点的查找操作 (符号表可以用作这个表)。第2个是DAG节点表，带有给出操作和子节点的查找功能，返回操作和孩子节点，若没有则返回空。这个操作允许查找已存在的值，不用在 DAG中构造新节点。例如，一旦图 8-6中的**+**节点及其子节点**x**和**1**被构造并分配名字 **t1**(作为三地址指令**t1=x+1**处理的结果)。在第2个表中查找(**+**、**x**、**1**将返回这个已构造的节点，三地址指令 **t2=x+1**只是使**t2**

⊖ 这个DAG结构的描述适合使用三地址指令作为中间代码，不过类似的 DAG可以定义用于P-代码或目标汇编代码。

也被赋给这个节点。这个构造的细节可以在别处找到(参见“注意与参考”部分)。

目标代码或修订过的中间代码，可以通以一种可能的拓朴顺序遍历DAG的非叶子节点来产生(DAG的拓扑顺序是一种遍历，节点的孩子先于双亲被访问)。因为有多种拓朴顺序，可以从DAG中产生许多不同代码序列。哪一种更好依赖于多种因素，包括目标机器结构的细节。例如，图8-5中3个非叶子节点的一个正规遍历序列将产生下面的三地址指令。这将代替原始的基本块：

```
t3 = x - 1
t2 = fact * x
x = t3
t4 = x == 0
fact = t2
```

当然，我们会希望避免使用临时变量，于是要产生下面的等价三地址码，其顺序必须固定：

```
fact = fact * x
x = x - 1
t4 = x == 0
```

图8-6中的DAG的一个类似遍历产生下示修正三地址码

```
t1 = x + 1
x = t1 * t1
```

使用DAG的基本块产生目标代码，自动得到局部公共子表达式的消除。 DAG表示法也使消除冗余存储(赋值)成为可能并告诉我们对每个变量的引用次数(节点的父节点数表示引用数)。这为合理分配寄存器提供了有用的信息(例如，假设一个值有多个引用，则放入寄存器；如果看到了所有引用，这个值已消亡不必再维护了等等)。

最后一个常用于辅助寄存器分配作为代码生成中数据维护的方法称为寄存器描述器(register descriptor)和地址描述器(address descriptor)。寄存器描述器为每个寄存器建立一个列表，表中列出当前值在寄存器中的变量名(当然，在那一点它们有同样的值)。地址描述器则为每个变量与内存地址建立联系。这些可以是寄存器(这种情况下，变量可以在相应的寄存器描述器中找到)或内存或两者都有(如果变量刚从内存中装入寄存器，但值尚未改变)。这种描述器允许跟踪值在内存和寄存器之间的移动，并可以重用已装入寄存器的值，还有回收寄存器，不管是不再包含后续使用变量的值或将值存入到适当内存位置中(溢出操作)。

例如，图8-5中的基本块DAG，并考虑，使用3个寄存器0、1和2对应从左到右遍历内部节点产生TM代码。假设有4个地址描述：**inRey(reg-no)**、**isGlobal(global-offset)**、**isTemp(temp-offset)**和**isConst(value)**(这对应于TM机上的TINY运行时刻环境，见前面章节的讨论)。进一步假设**x**在全程位置0，**fact**在全程位置1，全程位置通过gp寄存器访问，临时变量位置则通过mp寄存器访问。最后，假设所有寄存器中都无初始值，这样在基本块代码产生开始之前，变量和常量的地址描述器如下所示：

变量/常量	地址描述器
fact	isGlobal(1)
x	isGlobal(0)
t2	-
t3	-
t4	-
1	isConst(1)
0	isConst(0)

寄存器描述器表为空，就不再列出了。

现在假设产生了如下代码：

```
LD 0,1(gp) load fact into reg 0
LD 1,0(gp) load x into reg 1
MUL 0,0,1
```

那么地址描述器就变成

变量/常量	地址描述器
fact	inReg(0)
x	isGlobal(0),inReg(1)
t2	intReg(0)
t3	-
t4	-
1	isConst(1)
0	isConst(0)

寄存器描述器则为

寄存器	包含的变量
0	fact, t2
1	x
2	-

现在给出后续代码

```
LDC 2,1(0) load constant 1 into reg 2
ADD 1,1,2
```

地址描述器变成：

变量/常量	地址描述器
fact	inReg(0)
x	inReg(1)
t2	inReg(0)
t3	inReg(1)
t4	-
1	isConst(1),intReg(2)
0	isConst(0)

寄存器描述器变成：

寄存器	变量/常量
0	fact, t2
1	x, t3
2	1

我们把计算DAG中剩余节点值的代码以及描述结果地址和寄存器描述器留给读者完成。

我们的代码优化技术概览到为止。

8.10 TINY代码生成器的简单优化

8.8节中给出的TINY语言代码生成器产生的代码效率很低。这可以从程序清单 8-14的42条指令与程序清单8-13中9条手写的等价程序指令的比较中看出来。基本上，低效率有两个来源：

1) TINY代码生成器对TM机的寄存器的运用较差(实际上，从来不使用寄存器2、3或4)。

2) TINY代码生成器为测试产生不必要的逻辑值 0和1，由于这些测试只出现在 if语句和while语句中，简单代码即可实现。

本节希望指出相对粗糙的技术如何实质地提高 TINY编译器产生的代码的性能。实际上，不必生成基本块或流图而是直接从语法树产生代码。唯一需要的机制是附加的属性数据和一些稍显复杂的编译代码。我们不给出此处描述的改进的实现细节，还是留给读者练习。

8.10.1 将临时变量放入寄存器

我们首先描述的优化是一个简单方法，把临时变量保存在寄存器中，而不是经常操作内存读出和写入。在TINY代码生成器中临量变量总是存在位置：

```
tmpOffset(mp)
```

这里**tmpOffset**是初始的静态值，每次临时变量存储后递减，而每次读出后则递增 (见附录B，第2033行和第2027行)。将寄存器作为临时变量存储位置的一个简单方法是把 **tmpOffset**翻译成初始指向寄存器，只在可用寄存器用完时使用实际的内存偏移。例如，假设我们要将所有可用寄存器用于临时变量 (除pc、gp和mp)，那么**tmpOffsef**值0~4将翻译成寄存器0~4的引用，当值从-5开始时就使用偏移(在值上加上5)。这种机制可以直接应用于代码生成器的相应测试，可封装进辅助过程(将命名为**saveTmp**和**loadTmp**)。还要注意，在产生递归代码后子表达式的计算结果应保存除0外的其他寄存器中。

有了这些改进，TINY代码生成器产生如程序清单8-15所示的TM代码序列(请与程序清单8-14比较)。这个代码缩短20%并没有临时变量的存储(也没有使用寄存器5，即mp)。寄存器2、3和4仍没有使用。这并不奇怪：程序中的表达式很少复杂到同时需要两个或 3个临时变量。

程序清单8-15 临时变量保存在寄存器中的示例TINY程序的TM代码

```
0:    LD   6,0(0)        17:   ST   0,1(5)
1:    ST   0,0(0)        18:   LD   0,0(5)
2:    IN   0,0,0         19:   LDC  1,1(0)
3:    ST   0,0(5)        20:   SUB  0,0,1
4:    LDC  0,0(0)        21:   ST   0,0(5)
5:    LD   1,0(5)        22:   LD   0,0(5)
6:    SUB  0,0,1         23:   LDC  1,0(0)
7:    JLT  0,2(7)        24:   SUB  0,0,1
8:    LDC  0,0(0)        25:   JEQ  0,2(7)
9:    LDA  7,1(7)        26:   LDC  0,0(0)
10:   LDC  0,1(0)        27:   LDA  7,1(7)
11:   JEQ  0,21(7)       28:   LDC  0,1(0)
12:   LDC  0,1(0)        29:   JEQ  0,-16(7)
13:   ST   0,1(5)        30:   LD   0,1(5)
14:   LD   0,1(5)        31:   OUT  0,0,0
15:   LD   1,0(5)        32:   LDA  7,0(7)
16:   MUL  0,0,1         33:   HALT 0,0,0
```

8.10.2 在寄存器中保存变量

进一步的改进可以将TM的一些寄存器用于变量存储。这比前面的优化所做的工作要多，因为变量位置必须在代码生成和存储符号表之前确定。一个基本方案是简单选取几个寄存器存放程序中最常用的几个变量。为了确定哪些变量“最常用”，必须给出引用次数(使用和赋值)。在循环中引用的变量(在循环体或测试表达式中)应优先考虑，因为引用在循环执行时将重复进行。在许多现在编译器中工作出色的一个简单方法是将所有循环内引用都乘以10，在两层嵌套循环中乘100，如此类推。引用计数可以在语法分析时完成。之后作出分别的变量传递，符号表中储存的位置属性必须能表明那些定位于寄存器的变量与定位于内存的变量的差别。一个简单的方案使用枚举类型指示变量位置：本例中，只有两种可能 **inReg**和**inMem**。另外，第1种情况要记录寄存器号，第2种情况要记录内存地址(这是变量地址描述器的一个简单例子：不需要寄存描述器，因为它们在代码生成过程中保持不变)。

有了这些改变，示例程序的代码将使用寄存器保存变量 **x**，寄存器4保存**fact** (这里只有两个变量，所以可以都放入寄存器)。假设寄存器0到2仍保留给临时变量使用。对程序清单8-2代码的修改给出在程序清单8-16中。这段代码又比前面代码缩短许多，不过仍比手写的代码长。

程序清单8-16 临时变量和变量保存在寄存器中的示例 TINY**程序的**TM**代码**

```
 0:    LD  6,0(0)      13:   LDC  0,1(0)
 1:    ST  0,0(0)      14:   SUB  0,3,0
 2:    IN  3,0,0       15:   LDA  3,0(0)
 3:   LDC  0,0(0)      16:   LDC  0,0(0)
 4:   SUB  0,0,3       17:   SUB  0,3,0
 5:   JLT  0,2(7)      18:   JEQ  0,2(7)
 6:   LDC  0,0(0)      19:   LDC  0,0(0)
 7:   LDA  7,1(7)      20:   LDA  7,1(7)
 8:   LDC  0,1(0)      21:   LDC  0,1(0)
 9:   JEQ  0,15(7)     22:   JEQ  0,-12(7)
10:   LDC  4,1(0)      23:   OUT  4,0,0
11:   MUL  0,4,3       24:   LDA  7,0(7)
12:   LDA  4,0(0)      25:  HALT  0,0,0
```

8.10.3 优化测试表达式

我们讨论的最后一个优化是简化生成的if语句和while语句代码。因为这些表达式产生的代码很通用，布尔值真和假应用为0和1，尽管TINY没有布尔变量且不需要这种通用级别。这还导致了额外的装入常量0和1，以及由**genStmt**代码独立产生用于控制语句的额外测试。

此处描述的改进依赖于比较操作符必须为测试表达式的根节点。这个操作符的 **genExp**代码只是简单地产生代码将左操作数减去右操作数，把结果放入寄存器 0。if语句或while语句的代码将检查使用了哪个比较算符并产生相应的条件转移代码。

这样，程序清单8-16中TINY代码

```
if 0<x then
```

现在对应于TM代码

```
4: SUB 0,0,3
5: JLT 0,2(7)
6: LDC 0,0(0)
7: LDA 7,1(7)
8: LDC 0,1(0)
9: JEQ 0,15(7)
```

将由简单的TM代码代替

```
4: SUB 0,0,3
5: JGE 0,10(7)
```

(注意：假情况转移必须补充条件**JGE**到测试算符 < 中)。

有了这个优化，为测试程序生成的代码变成了程序清单 8-17所示(我们还在这一步骤中包括了去除代码尾部的空转移，对应于TINY代码中无else部分的if语句。这只要在**genStmt**中增加对if语句的简单检查即可)。

程序清单8-17的代码相对接近手写代码了。即使如此，还有一些特别情况可以优化，这将放在练习中。

程序清单8-17　变量与临时变量放入寄存器并简化表达式测试的示例 TINY程序的TM代码

```
0:     LD   6,0(0)        9:    LDC   0,1(0)
1:     ST   0,0(0)       10:    SUB   0,3,0
2:     IN   3,0,0        11:    LDA   3,0(0)
3:    LDC   0,0(0)       12:    LDC   0,0(0)
4:    SUB   0,0,3        13:    SUB   0,3,0
5:    JGE   0,10(7)      14:    JNE   0,-8(7)
6:    LDC   4,1(0)       15:    OUT   4,0,0
7:    MUL   0,4,3        16:   HALT   0,0,0
8:    LDA   4,0(0)
```

练习

8.1 为Lex表示法中的C注释写出一个正则表达式(提示：参见2.2.3节中的讨论)。给出对应下面算术表达式的三地址指令序列：

a. `2+3+4+5`

b. `2+(3+(4+5))`

c. `a*b+a*b*c`

8.2 给出对应前一个练习中算术表达式的P-代码序列。

8.3 给出对应下列C表达式的P-代码指令：

a. `(x = y = 2)*(x=4)`

b. `a(a)i=b(i=n)`

c. `p->next->next=p->next`

(假设相应的结构定义。)

8.4 给出前一练习中表达式的三地址指令。

8.5 给出对应下列TINY程序的(a)三地址码或(b)P-代码

```
{ Gcd program in TINY language }
read u;
```

```
read v; { input two integer }
if v = 0 then v := 0 { do nothing }
else
  repeat
    temp : = v;
    v := u - u/v*v; { computes u mod v }
    u := temp
  until v = 0
end;
write u { output gcd of original v & v }
```

8.6 参照程序清单8-4的四元式给出程序清单8-5三元式的C数据结构定义。

8.7 扩展表8-1P-代码的属性文法(8.2.1节)成 **a** 8.3.2节的子描述语法；**b**8.4.4节的控制结构语法。

8.8 对表8-2的三地址码属性文法重复上面练习。

8.9 描述代码生成如何采用6.5.2节的普通遍历过程，这是否有意义？

8.10 增加地址操作符**&**和***** (用C语法)以及二元结构域选择操作符.到

a. 8.2.1节的表达式语法。

b. 8.2.2节的语法树结构。

8.11 **a.** 为8.4.4节的控制语法添加repeat-until(或do-while语句)，并画出对应图8-2的合适控制图。

b. 为语法(8.4.4节)重写语法树结构定义以包含**a**部分的新结构。

8.12 **a.** 描述如何系统地将for语句转换成对应的while语句，用于产生代码是否可行？

b. 描述如何系统地将case或switch语句转换嵌套的if语句，用于产生代码是否可行？

8.13 **a.** 参照图8-2的Borland 80×86 C编译器显示的循环结构画出控制图。

b. 参照图8-2为8.6.2节中Sun SparcStatin C编译器显示的循环结构画出控制图。

c. 假设一个条件转移执行时间3倍于代码“穿过”(比如条件为假)。那么**a**和**b**部分的转移组织是否比图8-2有时间优势？

8.14 一种代替case或switch语句为每个case顺序测试的实现称为转移表(jump table)，其中case索引被用于索引转移的偏移对应到绝对转移。

a. 这种实现方法只有在大量不同case在相对较小的索引范围内密集发生时才有优势，为什么？

b. 代码生成器只是在超过10个case时才产生这种代码。确定你的C编译器是否有一个最小值决定是否产生switch语句的转移表。

8.15 **a.** 开发类似于8.3.2节的多维数组元素地址计算公式。说明你的所有假设。

b. 假设有如下用C代码定义的数组变量a

```
int a[12][100][5]
```

假设一个整数在内存中占两个字节。用你在 **a**部分中的公式确定下面变量相对 **a**基址的偏移

```
a[5][42][2]
```

8.16 参照8.5.2节的函数定义/调用语法给出下面程序：

```
fn  f(x)=x+1
fn  g(x,y)=x+y
```

```
g f(x(3),4+5)
```

a. 写出程序清单的**genCode**过程为该程序产生的P-代码指令序列。

b. 写出此程序的三地址指令代码。

8.17 文中没有指出**arg**三地址指令在函数调用中是否使用：有些版本的三地址码要求所有**arg**语句混合出现(参见8.5.1节)。讨论这两种方法的利弊。

8.18 **a.** 列出本章所用的全部P-代码指令，以及其意义和使用的描述。

b. 列出本章所用全部三地址指令，以及意义和使用的描述。

8.19 写出练习8.5中TINY gcd程序的等价TM程序：

8.20 **a.** TM没有寄存器到寄存器移动指令，说明这是如何实现的。

b. TM没有调用和返回指令，说明如何模拟实现。

8.21 为TM设计一个浮点协处理器，可以在不改变现存寄存器和内存定义的情况下使用(参见附录C)。

8.22 写出TINY编译器为下列TINY表达式和赋值产生的TM指令序列：

a. `2+3+4+5`

b. `2+(3+(4+5))`

c. `x:= x+(y+2*z)` 假设**x**,**y**和**z**分别在**dMem**位置0、1和2。

d. `v := u - u/v*v;`

(来自练习8.5TINY gcd程序中的一行；假设标准的TINY运行时环境)。

8.23 为8.5.2节的函数调用设计TM运行时环境。

8.24 Borland 3.0编译器产生如下80×86代码来计算**x<y**比较的逻辑结果，假设**x**和**y**为整数，在本地活动记录中的偏移为-2和-4：

```
        mov     ax,word ptr [bp-2]
        cmp     ax,word ptr [bp-4]
        jge     short @1@86
        mov     ax,1
        jmp     short @1@114
@1@86:
        xor     ax,ax
@1@114:
```

请与TINY编译器为同一表达式产生的TM代码比较。

8.25 检查你的C编译器如实现短回路布尔操作，并与8.4节中的控制结构实现比较。

8.26 为练习8.5中TINY gcd程序对应的三地址码画出流图。

8.27 为练习8.5中TINY gcd程序的repeat语句体的基本块画出DAG。

8.28 考虑8.9.3节的图8-5的DAG，假设最右节点的相等操作符在TM机中用相减模拟，对应这个节点的TM代码将如下所示：

```
LDC 2,0(0) load constant 0 into reg 2
SUB 2,1,2
```

写出执行上述指令后寄存器和地址描述器(基于8.9.3节)。

8.29 确定你的C编译器执行的优化，并与8.9节描述比较。

8.30 两个附加的优化可以应于TINY代码生成器，如下所示：

1) 如果测试表达式的一个操作数为常数0，则在产生跳条件转移之前不必执行相减。

2) 如果一条赋值语句的目标已在寄存器中，则右边表达式可以计算到此寄存器，这

样节省了寄存器到寄存器移动。

给出这两个优化增加到代码生成器中后，由示例 TINY程序产生的代码，这些代码与手写的程序清单8-13中的代码相比如何？

编程练习

8.31 重写程序清单8-7(8.2.2节)中的代码来产生P-代码作为同步字符串属性对应于表8-1的属性文法，并与程序清单8-7中代码比较复杂度。

8.32 重写下列P-代码产生过程来生成三地址指令：

a. 程序清单8-7(简单C表达式)。

b. 程序清单8-9(带数给的表达式)。

c. 程序清单8-10(控制语句)。

d. 程序清单8-11(函数)。

8.33 写出类似程序清单8-8的Yacc说明，对应以下代码生成过程。

a. 程序清单8-9(带数组的表达式)。

b. 程序清单8-10(控制语句)。

c. 程序清单8-11(函数)。

8.34 重写程序清单8-8的Yacc说明出产生三地址码代替P-代码。

8.35 为程序清单8-7的代码生成过程增加练习**8.10**中的操作符。

8.36 重写程序清单8-10的代码生成过程以包含练习**8.11**中的新控制结构。

8.37 重写程序清单8-7的代码，产生TM代码代替P-代码(假设代码生成实用程序在TINY编译器的`code.h`文件中)。

8.38 使用练习**8.23**中设计的运行时环境重写程序清单8-11的代码以产生TM代码。

8.39 **a**. 为TINY语言和编译器增加简单数组。这要求在语句之前添加数组定义，如

```
array a[10];
i := 1;
repeat
  read a[i];
  i := i + 1;
until 10 < i
```

b. 为(a)部分代码添加边界检查，于是越界下标将引起TM机停机。

8.40 **a.** 实现练习**8.21**中设计的TM浮点协处理器。

b. 用TM的浮点能力将TINY语言和编译器中的整数替换为实数。

c. 重写TINY语言和编译器使之同时包含整形和浮点值。

8.41 编写一个P-代码到三地址码的转换器。

8.42 编写一个三地址码到P-代码的转换器。

8.43 重写TINY代码生成器以生成P-代码。

8.44 编写P-代码到TM机代码的转换器，假设有前面练习中描述的 P-代码生成器以及文本描述的TINY运行时环境。

8.45 重写TINY代码生成器以产生三地址码。

8.46 编写三地址码到TM代码的转换器，假设有前面练习中的三地址码生成和文本描述的TINY运行时环境。

8.47 实现8.10节中描述的3种TINY代码生成器优化：

a. 将前3个TM寄存器用于临时变量。

b. 将寄存器3和4用于最常用变量。

c. 优化测试表达式代码，不再产生布尔值0和1。

8.48 在TINY编译器中实现常数合并。

8.49 **a.** 实现练习**8.30**中优化1。

b. 实现练习**8.30**中优化2。

注意与参考

代码生成和优化技术有很多；本章只是一个介绍。这些技术 (特别是数据流分析)的概览，理论观点上的把握，包含在AHO、Sethi和Ullman(1986)。一些更详细的部分主题参见Fischer和LeBlanc(1991)。case/switch语句的转移表(**8.14**)也有描述。专门处理器(MIPS、Sparc和PC)代码生成的例子见Fraser和Hanson(1995).代码生成作为属性分析在Slonneger和Kurtz(1995)。

自从第1个编译器出现，中间代码随编译器不同而不同成为移植问题的一个来源。最初，认为可以开发出通用中间代码用于所有编译器并解决移植问题 (Strong(1958)、Steel(1961)。不幸的是没有取得进展，三元式和四元式是中间代码的传统形式并用于许多编译器。 P-代码详细描述在Nori et al(1981)。一种更先进的 P-代码称为U-代码。允许更好的目标代码优化，在Perkins和Sites(1977)中描述。一个类似版本的 P-代码在Modula-2优化编译器中使用(Powell(1984).一个特殊的用于Ada编译器的中间代码，称为Diana,在Goos和Wulf(1981)中有描述。一种使用LISP风格前缀表达式的称为寄存器转换语言或RTL的中间代码在GNU编译器中使用(Stallman[1994])；这在Davidson和Fraser(1984a,b)中有描述。其余的可以用C编译器编译的中间代码例子参Holub(1990)。

没有综合的最新优化技术参考，尽管标准参考 Aho、Sethi和Ullman(1986)以及Fischer和LeBlanc(1991)包含了很好的总结。许多强大和实用的技术在 ACM Programming languages Design and Implementation Conferene Proceedings (先前称为Complier Construction Conference)中发表，这是作为ACM SEGPLAN Notices的一部分出现的。额外的优化技术来源参见 ACM Principles of Programing Languages Conference Proceedings 和ACM Transacfions on Programming Languages and Systems。