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

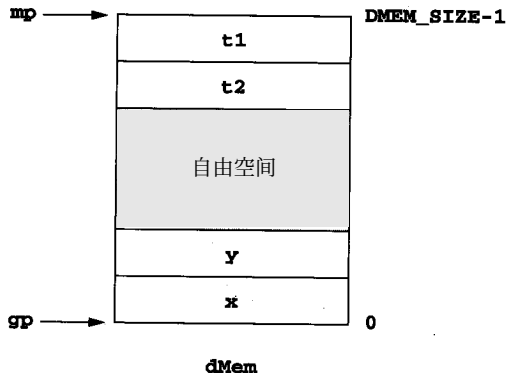
现在要描述TINY语言的代码生成器。我们假设读者熟悉 TINY编译器的先前步骤，特别是第3章中描述的分析器产生的语法树结构、第 6章描述的符号表构造，以及第 7章的运行时环境。

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
```