注意，最右边的叶节点出现在输出的最右一栏，根节点出现在输出的最左边。每一栏输出与前一栏之间有5个空格符。函数outputTree接收一个参数代表总共可以显示的宽度的变量totalSpaces（该变量从0开始，所以根节点在屏幕的最左边）。该函数使用修改过的中序遍历算法——该函数从最右节点开始，向左遍历树，算法如下：

在当前节点指针非空时
　递归调用outputTree函数来显示该节点的右子树并且totalSpaces + 5
　使用for循环，从1 ~ totalSpaces计数来输出空格
　输出当前节点的值
　将当前节点指针指向当前节点的左子树
　totalSpaces自增5

特殊小节：构建自己的编译器

在习题8.18和习题8.19中，我们介绍了Simpletron机器语言（SML），并且实现了一个Simpletron计算机模拟器来执行SML语言编写的程序。在这部分，我们构建一个编译器，它可以将使用高级语言编写的程序转换成SML。这部分把整个编程过程结合到一起。读者将会使用这个新的高级语言来编写程序，在自己构建的编译器上编译这些程序，并且在习题8.19中构建的模拟器上运行它们。你应该尽量使用面向对象的方法来实现自己的编译器。

21.26（Simple语言）在开始构建编译器之前，首先讨论一个简单但强大的高级语言，它与流行的BASIC语言的早期版本很相似。我们称这种语言为Simple。每个Simple语句包含一个行号和一条Simple指令。行号必须以递增的顺序出现。每条指令以下面的某条Simple命令开始：rem，input，let，print，goto，if...goto，end（参见图21.25）。除了end命令之外，所有的命令都可以反复使用。Simple只能计算由运算符“+”、“-”、“*”和“/”组成的整数表达式。这些运算符具有像在C++中一样的优先级。可以使用圆括号来改变表达式的计算顺序。

命令	示例语句	描述
rem	50 rem this is a mark	rem之后的文本是说明性的文字，会被编译器忽略
input	30 input x	输出一个“？”来提示用户输入一个整数。从键盘读入这个整数，并将它存储在x中
let	80 let u = 4 * (j – 56)	将4 * (j – 56)的结果赋给u。注意，等号右边可以是很复杂的表达式
print	10 print w	输出w的值
goto	70 goto 45	将程序控制转到第45行
if...goto	35 if i==z goto 80	比较i与z是否相等，如果条件为真，则将控制转到第80行，否则继续执行下一条语句
end	90 end	结束程序的执行

图21.25　Simple语言的命令

我们的编译器只能识别小写字母。Simple文件中的所有字符都应该是小写的（大写的字母会导致语法错误，除非它们出现在rem语句中，因为在该语句中可以忽略大写字母）。变量的名字是单个字母。Simple不允许描述性的变量名，所以变量应该在注释中解释，以说明它们在程序中的作用。Simple只使用整型变量。Simple没有变量声明——仅仅在程序出现变量名就可以声明变量并将其自动初始化为0。Simple语法不允许字符串操作（读字符串、写字符串、比较字符串等）。如果在一个Simple程序中遇到了字符串（在一个不是rem的命令之后），编译器就会生成一个语法错误。我们第一个版本的编译器将假设Simple程序是被正确输入的。习题21.29要求学生改进这个编译器来执行语法错误检查。

在程序执行期间，Simple使用条件转移语句if...goto和无条件转移语句goto来改变控制流。如果在if...goto语句中的条件为真，控制就会转向到程序中指定的一行。在if...goto语句中，下面的关系和相等运算符是有效的：<、>、<=、>=、==和!=。这些运算符有C++中一样的优先级。

让我们来看几个演示 Simple 特性的程序。第一个程序（参见图 21.26）从键盘读入两个整数，并把值存储在变量 a 和 b 中，然后计算并打印它们的和（存储在变量 c 中）。

```
10 rem   determine and print the sum of two integers
15 rem
20 rem   input the two integers
30 input a
40 input b
45 rem
50 rem   add integers and store result in c
60 let c = a + b
65 rem
70 rem   print the result
80 print c
90 rem   terminate program execution
99 end
```

图 21.26　计算两个整数和的 Simple 程序

图 21.27 中的程序判断并打印两个整数中较大的一个。这些整数从键盘输入并存储在 s 和 t 中。语句 if...goto 测试条件 s >= t 是否成立。如果条件是真的，那么控制转向第 90 行并且输出 s；否则，输出 t 并且将控制转向第 99 行的 end 语句，程序在此结束。

```
10 rem   determine the larger of two integers
20 input s
30 input t
32 rem
35 rem   test if s >= t
40 if s >= t goto 90
45 rem
50 rem   t is greater than s, so print t
60 print t
70 goto 99
75 rem
80 rem   s is greater than or equal to t, so print s
90 print s
99 end
```

图 21.27　找出两个整数中较大一个的 Simple 程序

Simple 不提供循环语句（像 C++ 的 for、while 或者 do...while）。但是，Simple 能够使用 if...goto 和 goto 语句模仿 C++ 中的每一个循环语句。图 21.28 用一个标记控制循环来计算几个整数的平方。每个整数从键盘输入并存储在变量 j 中。如果输入值是标记值 -9999，那么控制就转向第 99 行，程序在此结束。否则，k 被赋值为 j 的平方，k 输出到屏幕上并且将控制转向第 20 行，在那里输入下一个整数。

```
10 rem   calculate the squares of several integers
20 input j
23 rem
25 rem   test for sentinel value
30 if j == -9999 goto 99
33 rem
35 rem   calculate square of j and assign result to k
40 let k = j * j
50 print k
53 rem
55 rem   loop to get next j
60 goto 20
99 end
```

图 21.28　计算几个整数的平方值

以图 21.26、图 21.27 和图 21.28 的例子程序为指导，编写一个 Simple 程序完成如下的每一项：

a) 输入 3 个整数，计算它们的平均值并打印结果。

b) 使用标记控制循环输入 10 个整数计算并打印它们的和。

c) 使用计数器控制循环输入 7 个整数，有正数和负数，计算并打印它们的平均值。

d) 输入一系列整数确定并打印出最大的那个数。第一个输入的整数说明有多少数字要被处理。

e) 输入10个整数并打印出最小的那个数。

f) 计算并打印出从2～30的偶数之和。

g) 计算并打印从1～9的奇数之积。

21.27（**构建一个编译器的先决条件：完成习题8.18、习题8.19、习题21.12、习题21.13和习题21.26**）现在，我们已经展示了Simple语言（习题21.26），下面讨论怎样构建一个Simple编译器。首先，考虑如何将Simple程序翻译成SML并且由Simpletron模拟器（参见图21.29）执行。一个包含Simple程序的文件由编译器读入并且转化成SML代码。SML代码将输出到磁盘的一个文件上，在文件里一条SML指令占据一行。然后将SML文件装载到Simpletron模拟器中，并且将结果发送到磁盘文件和屏幕上。注意在习题8.19开发的Simpletron程序是从键盘获得输入的。必须将其修改成从文件读入以便它能运行编译器产生的程序。

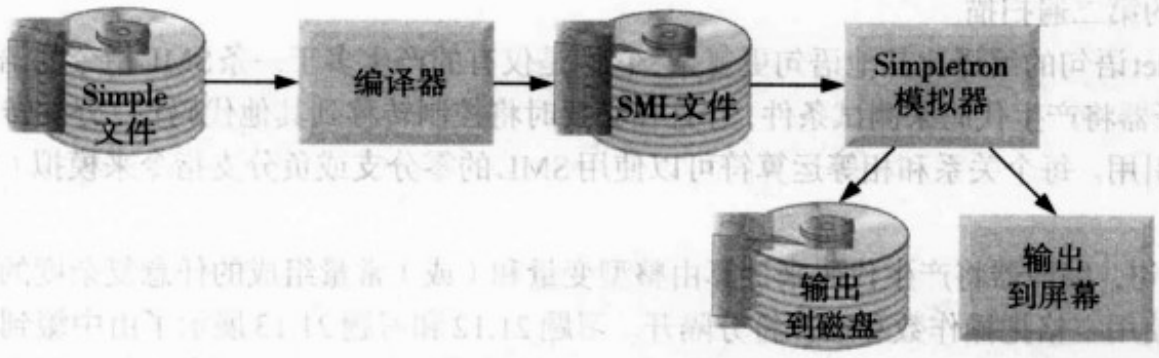

图21.29 编写、编译并执行一个Simple语言程序

Simple编译器执行两遍扫描，把Simple程序转换成SML。第一遍构建符号表（对象），在其中存储了Simple程序的每个行号（对象）、变量名（对象）和常量（对象）及它们的类型和在最终的SML代码中的相应位置（符号表将在下面详细讨论）。第一遍也为每个Simple语句（对象等）产生相应的SML指令对象。如我们将会看到的，如果Simple程序包含将控制转移到程序较后的语句时，第一遍生成的SML程序中会包含一些“未完成”的指令。第二遍编译器定位并完成那些未完成的指令，将SML程序输出到一个文件中。

第一遍扫描

编译器开始会把Simple程序的一个语句读入内存。必须将语句分解成独立的记号（即语句片段）来处理和编辑（标准库函数strtok可以很容易地完成这个任务）。回想一下，每条语句都是以行号开始，后跟一条命令。编译器把语句分解成记号，如果记号是一个行号、变量或常量，就将其放入符号表中。只有行号是语句中的第一个记号时才将其放入符号表。symbolTable对象是一个用来表示程序中符号的tableEntry对象的数组。程序中出现的符号的数目没有限制。因此，某些特殊程序的symbolTable可以很大。暂时先把symbolTable设成100个元素的数组。一旦程序开始运行，可以增大或缩小它的大小。每个tableEntry对象包含3个成员。成员symbol是一个整数，它包含一个变量（记住变量名是单字符）的ASCII值、一个行号或者一个常量。成员type是下面说明符号类型的字符之一：‘C’代表常量，‘L’代表行号，而‘V’代表变量。成员location包含了符号所在的Simpletron内存位置（00～99）。Simpletron内存是一个100个整数的数组，SML指令和数据存储在其中。对于行号来说，location是Simpletron内存数组中的一个元素，Simple语句的SML指令由此开始。对于变量或常量来说，location是Simpletron内存数组中的一个元素，变量或常量存储在那里。变量和常量是在Simpletron内存的后端从后向前分配的。第一个变量或常量存储在位置99，下一个在位置98，依次类推。

符号表在把Simple程序转化到SML的全程中都有作用。我们在第8章学习了一条SML指令，它是一个四位的整数，是由两部分组成的（操作代码和操作数），其操作代码由Simple中的命令来确定。例如，Simple命令input对应SML操作代码10（读取），而Simple命令print对应SML操作代码11（写入）。操作数是一个包含数据的内存位置，操作代码在操作数上执行它的任务（例如，操作代码10从键盘读入一个值，并且把它存储在由操作数指定的内存位置上）。编译器搜索symbolTable来为每个符号确定Simpletron内存位置，所以相应的内存位置可以用来完成SML指令。

每条 Simple 语句的编辑都是基于它的命令。例如，在 rem 语句中的行号被插入到符号表之后，编译器将忽略语句的剩余部分，因为注释只是起说明性文档的作用。语句 input、print、goto 和 end 对应 SML 的 read、write、branch（转移到一个指定的位置）和 halt 指令。包含这些 Simple 命令的语句被直接转化成 SML（注意、如果 goto 语句要转移到的指定行涉及 Simple 程序文件后面的语句，那么 goto 语句可能包含一个未解析的参数；这个参数有时称为向前引用）。

当使用一个未解析引用编译一条 goto 语句时，必须对其 SML 指令进行标记，以指出编译器的第二遍扫描一定要完成该指令。将标记存储在有 100 个元素的 int 型 flags 数组中，该数组中的元素都初始化为 -1。如果 Simple 程序中的行号涉及的内存位置还是未知的（也就是它不在符号表中），那么行号就会存储在 flags 数组中与未完成指令有相同下标的元素中。未完成指令的操作数临时被设置为 00。例如，一个无条件分支指令（涉及到向前引用）将译为 +4000，直到编译器的第二遍扫描改变它。下面将简单描述编译器的第二遍扫描。

if...goto 和 let 语句的编译比其他语句更复杂，它们是仅有的产生多于一条 SML 指令的语句。对于 if...goto 语句，编译器将产生代码来测试条件，并且在必要时将控制转移到其他代码行。分支转移的结果可能是未解析的引用。每个关系和相等运算符可以使用 SML 的零分支或负分支指令来模拟（或者是二者的结合）。

对于 let 语句，编译器将产生代码来计算由整型变量和（或）常量组成的任意复杂度的算术表达式。表达式应该使用空格把操作数和运算符分隔开。习题 21.12 和习题 21.13 展示了由中缀到后缀的转换算法和由编译器计算后缀表达式的算法。在实现自己的编译器之前，应该完成这些练习。当编译器遇到一个表达式时，它把表达式由中缀表示法转换成后缀表示法，然后计算后缀表达式。

编译器是如何产生机器语言来计算包含变量的表达式呢？后缀算法包含一个“异常指令”（hook），在那里编译器能生成 SML 指令而并不是实际地计算表达式。为了使“异常指令”能在编译器中使用，必须改进后缀计算算法，使得它遇到每个符号时都会搜索符号表（可能是把该符号插入），确定该符号对应的内存位置，并把这个内存位置（而不是符号）压入堆栈。当在后缀表达式中遇到一个运算符时，在堆栈顶部的两个内存位置被弹出，并且使用该内存位置作为操作数来生成实现这个运算的机器语言。每个子表达式的结果将存储在内存中的一个临时位置上，并将其压回堆栈以便于后缀表达式的计算能够继续。当后缀计算完成时，包含计算结果的内存位置是唯一留在堆栈中的位置。该内存位置出栈，生成 SML 指令并把结果赋给 let 语句左边的变量。

第二遍扫描

编译器的第二遍扫描执行两个任务：解决任何未解析引用，并将 SML 代码输出到一个文件中。参数的确定步骤如下：

a) 搜索 flags 数组寻找未解析引用（即值不是 -1 的元素）。

b) 定位数组 symbolTable 中的对象，它包含了存储在 flags 数组中的符号（确保行号的符号类型是 ‘L’）。

c) 把成员 location 的内存位置插入到含有未解析引用的指令中（记住，包含未解析引用的指令的操作数是 00）。

d) 重复步骤 1 ~ 3，直到 flags 数组的最后。

在确定过程完成之后，包含 SML 代码的整个数组将被输出到一个磁盘文件中，该文件中的每行都有一条 SML 指令。这个文件可以由 Simpletron 读取并执行（在模拟器改进为从文件中读取输入之后）。将第一个 Simple 程序编译为一个 SML 文件并且执行该文件，这会给你带来真正的成就感。

一个完整的示例

下面的例子用图说明了把 Simple 程序转化为 SML 的完整过程，该 SML 将由 Simple 编译器执行。考虑一个 Simple 程序，要求输入一个整数并计算从 1 到该整数的所有整数之和。程序和 Simple 编译器第一遍扫描所产生的 SML 指令在图 21.30 中说明。第一遍扫描构造的符号表在图 21.31 中展示。

简单程序	SML 位置和指令	描述
5 rem sum 1 to x	无	忽略 rem 语句
10 input x	00 +1099	将 x 读取到地址 99
15 rem check y==x	无	忽略 rem 语句
20 if y == x goto 60	01 +2098	将 y（98）读入累加器中
	02 +3199	从累加器中减去 x（99）
	03 +4200	零分支转移到一个未解析的位置
25 rem increment y	无	忽略 rem 语句
30 let y = y+1	04 +2098	将 y 读入累加器
	05 +3097	累加器加 1（97）
	06 +2196	将累加器中的值暂存在地址 96 中
	07 +2096	取出暂存在地址 96 中的数
	08 +2198	将累加器中的值存入 y 中
35 rem add y to total	无	忽略 rem 语句
40 let t = t+y	09 +2095	将 t（95）读入累加器中
	10 +3098	累加器加 y
	11 +2194	将累加器中的值暂存在地址 94 中
	12 +2094	取出暂存在地址 94 中的数
	13 +2195	将累加器中的值存在 t 中
45 rem loop y	无	忽略 rem 语句
50 goto 20	14 +4001	跳转到地址 01
55 rem output result	无	忽略 rem 语句
60 print t	15 +1195	将 t 输出到屏幕
99 end	16 +4300	执行结束

图 21.30　编译器第一遍扫描产生的 SML 指令

符号	类型	地址
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97
35	L	09
40	L	09
't'	V	95
45	L	14
50	L	14
55	L	15
60	L	15
99	L	16

图 21.31　图 21.30 程序中的符号表

大多数的 Simple 语句直接转化成单一的 SML 指令。但注释、第 20 行的 if...goto 语句和 let 语句是例外。不会将注释翻译成机器语言，但是注释的行号将放在符号表里，以防止在 goto 语句或 if...goto 语句中引用这个行号。第 20 行指明，如果条件 y == x 为真，那么程序控制就转向第 60 行。因为第 60 行在程序中出现较晚，所以编译器第一遍扫描还没把 60 放在符号表里（只有语句行号作为语句中的第一个记号

出现时才将其放到符号表中）。因此，这时不可能确定SML零分支指令的操作数，该指令在SML指令数组的位置03。编译器把60放在flags数组的位置03处，以表明第二遍扫描要完成该指令。

我们必须跟踪下一条指令在SML数组中的位置，因为Simple语句和SML指令不是一一对应的。例如，第20行的if...goto语句编译成3条SML指令。每当生成一条指令，就必须增加指令计数器到SML数组的下一个位置。注意Simpletron内存的大小可能给有很多指令、变量和常量的Simple程序带来问题，很可能使得编译器耗光内存。为了能检测到这种情况，程序应该包含一个数据记数器，以跟踪下一个变量或常量存储在SML数组中的位置。如果指令记数的值大于数据记数器的值，那么SML数组就已经满了。在这种情况下，编译过程应该结束，并且编译器应该打印一个错误信息以说明在编译期间耗尽了内存。这是为了强调这样的事实：尽管程序员依靠编译器从管理内存的负担中解放出来，但是编译器自己要小心确定内存中的指令和数据分布，并且必须检验像编译期间内存耗尽等类型的错误。

一步步观看编译过程

现在让我们完整看一看图21.30中Simple程序的编译过程。编译器把程序的第一行：

```
5 rem sum 1 to x
```

读入内存。该语句中的第一个记号（行号）可用函数strtok来确定（参见第8章和第21章对C++的C风格字符串处理函数的讨论）。由strtok返回的记号使用函数atoi转化成一个整数，所以符号5可以定位在符号表中。如果没有找到该符号，那么就将其插入到符号表中。因为我们在程序的开始而且这是程序的第一行代码，所以在符号表中没有这个符号。因此5要作为类型L（行号）插入到符号表中，并且将其放在SML数组的第一个位置（00）上。尽管这一行是一条注释，但是还会给这个行号分配符号表中的一个位置（防止被一个goto语句或if...goto语句引用）。rem语句没有SML指令生成，所以指令计数器不增加。

语句：

```
10 input x
```

是下一个被记号化的语句。行号10作为类型L插入在符号表中，并且分配给SML数组的第一个位置（00，因为程序是以注释开始的，所以指令计数器当前为00）。命令input指明下一个记号是一个变量（只有变量可以出现在input语句中）。因为input直接对应着一个SML操作代码，编译器必须确定x在SML数组中的位置。没有在符号表中找到符号x，所以它以x的ASCII形式插入到符号表中，类型为V，分配到SML数组中99的位置（数据存储从99开始，从后向前分配）。现在可以给这条语句生成SML代码了。操作代码10（SML读入操作代码）乘以100，然后加上x的位置就形成该指令。这条指令然后被存储在SML数组的位置00上。指令计数器加1，因为已经生成了一条SML指令。

语句：

```
15 rem check y == x
```

是接下来被记号化的语句。对符号表进行搜索来寻找行号15（没有找到），所以这个行号作为类型L插入并分配给数组中的下一个位置01（记住rem语句没有生成代码，所以指令计数器的值没有增加）。

语句：

```
20 if y == x goto 60
```

是下一个被记号化的语句。将行号20插入符号表中并赋予类型L，其位置在SML数组01。命令if指明要计算一个条件。由于不能在符号表中找到变量y，因此就将其插入符号表，类型设为V，SML位置为98。接下来，生成SML指令来测试条件。因为对于if...goto语句来说，在SML中没有直接的等价运算，所以必须将其模拟为执行一个x和y的计算，然后根据结果进行转移。如果y等于x，那么从y中减去x应该等于0，所以零分支指令可以根据计算结果来模拟if...goto语句。第一步要求把y载入（从SML位置98）到一个累加器中，产生指令01 + 2098。接着，从累加器中减去x，生成指令02 + 3199。累加器中的值可以是零、正数或负数。因为运算符是“==”，我们需要零分支指令。首先，搜索符号表来寻

找转移地址（在这里是60），但是没有找到。所以将60放入flags数组中，位置是03，并且生成指令03 + 4200（不能增加转移地址，因为还没为行号60在SML数组中分配地址）。指令计数器增到04。
编译器处理语句：

```
25 rem increment y
```

将行号25以类型L插入到符号表并分配SML地址04。指令计数器不变。
当语句：

```
30 let y = y + 1
```

被记号化时，将行号30以类型L插入到符号表中并分配SML地址04。命令let指明该行是一个赋值语句。首先，将行中的所有符号插入到符号表中（如果它们不在符号表中）。将整数1以类型C添加到符号表中并分配SML地址97。接下来，赋值符号的右边从中缀形式转化到后缀形式。然后，计算后缀表达式(y 1 +)。符号y被定位在符号表中，将其相应的内存地址压入堆栈，同时将符号1定位在符号表，将其对应的内存地址压入堆栈。当遇到运算符“+”时，后缀求值程序弹出堆栈作为该运算符的右侧操作数，又弹出堆栈作为运算符的左侧操作数，并产生SML指令：

```
04 +2098 (载入y)
05 +3097 (增加1)
```

表达式的计算结果存储在内存中的临时地址（96）中，其指令为

```
06 +2196 (存储临时值)
```

并且将临时地址压入堆栈。既然已经计算了这个表达式，结果必须存储在y中（即“=”左边的变量）。所以将临时地址装载到累加器中，累加器存储在y中，利用指令：

```
07 +2096 (载入临时值)
08 +2198 (存储y)
```

读者马上会注意到SML指令看起来比较冗长。我们会简单地谈论这个问题。
当语句：

```
35 rem add y to total
```

被记号化时，将行号35以类型L插入到符号表中并分配地址09。
语句：

```
40 let t = t + y
```

和代码行30相似。将变量t以类型V插入到符号表中并分配SML地址95。指令遵从与代码行30相同的逻辑和格式，生成指令09 + 2095、10 + 3098、11 + 2194、12 + 2094和13 + 2195。注意，在将其赋给t（95）之前，t + y的结果被赋给临时地址94。读者会再次注意到，内存地址11和12的指令看起来也是冗余的。我们将再次简单地讨论这个问题。
语句：

```
45 rem loop y
```

是一行注释，所以将行号45以类型L加入到符号表中并分配SML地址14。
语句：

```
50 goto 20
```

把控制转移到代码行20。将行号50以类型L插入到符号表中并分配SML地址14。goto等价于SML中的无条件转换(40)指令，该指令把控制转移到一个特定的SML地址。编译器为代码行20搜索符号表并发现它对应的SML地址01。运算代码(40)乘以100，并加上地址01，产生指令14 + 4001。
语句：

```
55 rem output result
```

是一行注释，将行号 55 以类型 L 插入到符号表中并分配 SML 地址 15。
语句：

```
60 print t
```

是一条输出语句。将行号 60 以类型 L 插入到符号表中并分配 SML 地址 15。print 等价于 SML 中的运算代码 11（写入）。从符号表中确定 t 的地址并把其加到运算代码乘以 100 的结果上。
语句：

```
99 end
```

是该程序的最后一行代码。将行号99以类型L存储到符号表中并且分配SML地址16。end命令产生SML指令 +4300（在 SML 中 43 是停机），作为最后一条指令写在 SML 内存数组中。
这就完成了编译器的第一遍扫描，现在来考虑第二遍扫描。搜索 flags 来寻找非 -1 的值。地址 03 包含 60，所以编译器知道指令 03 是未完成的。编译器完成该指令通过搜索符号表寻找 60，确定它的地址并把地址加到这个未完成的指令上。在这里，搜索确定了代码行 60 对应于 SML 地址 15，所以生成完整的指令 03 +4215 以代替 03 +4200。这个 Simple 程序现在就完全编译成功了。
为了构建编译器，必须完成如下的每个任务：

a) 改进在习题8.19中编写的Simpletron模拟器程序，使它从用户指定的文件中读取输入（参见第17章）。该模拟器应该把它的结果按照与屏幕输出相同的格式输出到一个磁盘文件中。把该模拟器转化为一个面向对象的程序，特别是把硬件的每个部分看做一个对象。利用继承把指令的类型安排到一个类层次中。然后通过 executeInstruction 消息告诉每个指令执行它自己，从而多态地运行这个程序。

b) 改进习题21.12中的中缀转换成后缀算法，处理多位整型操作数和单字母变量名操作数。[提示：C++ 标准库函数 strtok 可以用来定位表达式中的每个常量和变量，并且使用标准库函数 atio（<csdtlib>）使变量可以从字符串转化成整数。][注意：必须改变后缀表达式的数据表示，以支持变量名和整型常量。]

c) 改进后缀计算算法，处理多位整型的操作数和变量名操作数。并且该算法应该实现前面讨论的“异常指令”，以便生成 SML 指令而不是直接计算表达式。[提示：标准库函数 strtok 可以用来定位表达式中的每个常量和变量，并且使用标准库函数atio使变量可以从字符串转化成整数。][注意：必须改变后缀表达式的数据表示，以支持变量名和整型常量。]

d) 构建编译器。合并 b)部分和 c)部分，计算 let 语句中的表达式。你的程序应该包含一个执行编译器第一遍扫描的函数，以及一个执行编译器第二遍扫描的函数。两个函数都要调用其他函数来完成它们的任务。尽量使用面向对象的方法来编写你的编译器。

21.28（**优化 Simple 编译器**）程序经过编译被转化成 SML 后，就会产生一组指令集合。指令的某种组合会经常出现，通常以称为生成式的三元组形式出现。一个生成式一般包含三个指令，例如 load、add 和 store。例如，图 21.32 描述了五个 SML 指令，它们是在图 21.30 程序的编译过程中产生的。前三条指令是一个生成式，它把 1 加到 y 上。注意，指令 06 和 07 把累加器的值存储在一个临时地址 96 中，并且把该值装回到累加器中，所以指令 08 能把该值存储在地址 98 中。通常一个生成式后面跟着一条与刚刚存储的地址相同的装入指令。这些代码可以通过删除存储指令和后来在相同地址进行操作的装入指令来优化，从而使 Simpletron 可以更快地执行程序。图 21.33 说明了图 21.30 中程序优化后的 SML。注意在优化后的代码中少了四条指令，因此节省 25% 的内存空间。
改进编译器来提供一个优化其生成的Simpletron机器语言代码的选项。手动比较未优化和优化后的代码，并计算缩简的百分比。

```
1  04  +2098  (load)
2  05  +3097  (add)
3  06  +2196  (store)
4  07  +2096  (load)
5  08  +2198  (store)
```

图 21.32　未优化的图 21.30 中的程序代码

Simple 程序	SML 位置和指令	描述
5 rem sum 1 to x	无	忽略 rem 语句
10 input x	00 +1099	将 x 读到地址 99
15 rem check y==x	无	忽略 rem 语句
20 if y == x goto 60	01 +2098	将 y（98）读入累加器中
	02 +3199	从累加器中减去 x（99）
	03 +4211	如果是 0 转移到地址 11
25 rem increment y	无	忽略 rem 语句
30 let y=y+1	04 +2098	将 y 读入累加器中
	05 +3097	累加器加 1（97）
	06 +2198	将累加器中的值存入 y（98）中
35 rem add y to total	无	忽略 rem 语句
40 let t = t+y	07 +2096	将 t（96）读入累加器中
	08 +3098	累加器加 y（98）
	09 +2196	将累加器中的值存于 t（96）中
45 rem loop y	无	忽略 rem 语句
50 goto 20	10 +4001	跳转到地址 01
55 rem output result	无	忽略 rem 语句
60 print t	11 +1195	输出 t（96）到屏幕
99 end	12 +4300	执行结束

图 21.33 已优化的图 21.30 中程序的代码

21.29（**对 Simple 编译器的修改**）执行下面对 Simple 编译器进行的修改。一些修改可能也要求对在习题 8.19 中编写的 Simpletron 模拟器进行修改。

a) 允许在 let 语句中使用取模运算符（%）。必须修改 Simpletron 机器语言以包括取模指令。

b) 允许在 let 语句中使用“^”作为求幂运算符来求幂。必须修改 Simpletron 机器语言以包括求幂指令。

c) 允许编译器识别出在Simple语句中的大写和小写字母（例如，'A'与'a'是等同的）。不允许修改模拟器。

d) 允许 input 语句一次读入多个变量值，例如 input x, y。不允许修改 Simpletron 模拟器。

e) 允许编译器在单一的 print 语句中输出多个值，例如 print a, b, c。不允许修改 Simpletron 模拟器。

f) 给编译器增加语法检查功能，这样在 Simple 程序中出现语法错误时，输出错误信息。不允许修改 Simpletron 模拟器。

g) 允许使用整型数组。不允许修改 Simpletron 模拟器。

h) 允许使用 Simple 命令 gosub 和 return 指定的子程序。命令 gosub 将程序控制传递到一个子程序中，而命令 return 在 gosub 之后把控制传递回该语句。这和C++中的函数调用相似。相同的子程序可以被分布在整个程序中的许多 gosub 命令调用。不允许修改 Simpletron 模拟器。

i) 允许下列形式的循环语句

```
for x = 2 to 10 step 2
    Simple statements
next
```

该语句从 2 ~ 10 以 2 为增量开始循环。next 行标记 for 语句主体的结束。不允许修改 Simpletron 模拟器。

j) 允许下列形式的循环语句：

```
for x = 2 to 10
    Simple statements
next
```

该语句从 2 ~ 10 以一个默认的增量 1 开始循环。不允许修改 Simpletron 模拟器。

k) 允许编译器处理字符串的输入和输出。这要求 Simpletron 模拟器进行改进以处理和存储字符串值。［提示：每个 Simpletron 字可以被划分为两组，每组有一个两位的整数。每个两位整数代表一个字符

的ASCII的十进制表示。增加一个机器语言指令，使其能打印从某个Simpletron内存地址开始的一个字符串。在该地址的字的前半部分是该字符串中字符个数的记数（即字符串的长度）。随后的每半个字包含一个表示为两个十进制数字的ASCII字符。机器语言指令检验该字符串的长度，然后把每个两位数字转化成等价字符并打印出来。]

l) 允许编译器除了整数外还可以处理浮点数。Simpletron模拟器也必须改进以处理浮点值。

21.30（**一个Simple解释器**）解释器是一个程序，它读入一种高级语言程序语句，确定要由语句执行的操作并立即执行该操作。高级语言程序开始并没有转换成机器语言。解释器执行得较慢，因为在程序中遇到的每条语句必须先被解释。如果语句包含在一个循环中，那么每次遇到它时都要解释一次。BASIC编程语言的早期版本就是作为解释器实现的。

为习题21.26中讨论的Simple语言编写一个解释器。该程序应该利用习题21.12中开发的中缀表示到后缀表示转换程序和习题21.13中开发的后缀求值程序，计算let语句中的表达式。在习题21.26中加在Simple语言上的限制在这个程序中继续存在。使用练习题21.26中编写的Simple程序来测试该解释器。比较程序在解释器中的运行结果和在习题8.19构建的Simpletron模拟器中编译运行的结果。

21.31（**在链表的任何地方进行插入或删除操作**）我们的链表类模板只允许在链表的前端和后端进行插入和删除操作。通过重用链表类模板，这些功能在使用private继承和组成产生堆栈类模板和队列类模板时很方便，可以使代码量最小化。事实上，链表比我们提供的那些数据结构更加通用。修改我们在本章开发的链表类模板，使之能够处理在链表中的任何位置进行插入和删除操作。

21.32（**没有尾指针的链表和队列**）我们链表（参见图21.3～图21.5）的实现使用了一个firsrPtr和一个lastPtr。对于链表类的insertAtBack和removeFromBack lastPtr成员函数来说，lastPtr是非常有用的。insertAtBack函数对应于Queue类的enqueue成员函数。重写List类，使它不使用lastPtr。因此，在链表尾部的任何操作必须从前端搜索该链表开始进行。这影响到我们的Queue类（参见图21.16）的实现了吗？

21.33 利用组成版本的堆栈程序（参见图21.15）来形成一个完整的工作堆栈程序。改进这个程序来内联那些成员函数。比较这两种方法。总结内联成员函数的优点和缺点。

21.34（**二叉树排序和搜索的性能**）二叉树排序的一个问题是数据插入的顺序对树的形状的影响：对于相同的数据集合，以不同的顺序插入可以戏剧性地产生不同形状的二叉树。二叉树排序和搜索算法的性能对树的形状是很敏感的。如果其数据按递增的顺序插入二叉树，那么会有什么样的形状呢？按递减顺序呢？为了达到最大的搜索性能，树应该是什么形状的？

21.35（**索引表**）文中曾经介绍过，必须顺序地搜索链表。对于很大的链表，这可能导致很差的性能。为了改进链表搜索性能，一个常用的技术就是创建和维护一个对链表的索引。索引就是一个指向链表中各个关键字位置的指针的集合。例如，一个搜索巨大的名字链表的应用程序，可以通过创建一个有26个项（对应字母表中一个字母）的索引来改善性能。对一个以‘Y’开头的姓的搜索操作，将首先搜索索引来确定‘Y’项在哪里开始并在哪一点进入链表。接着进行线性查找，直到找出所要的名字。这会比从头搜索链表要快得多。使用图21.3～图21.5的List类作为IndexedList类的基础。编写一个程序演示索引表的操作。保证要包含成员函数insertInIndexedList、searchIndexedList和deleteFromIndexedList。