```
// Exercise 15.25 solution
#include <iostream>

using std::cout;

#include <cstdlib>
#include <ctime>
#include "tree2.h"

int main()
{
   srand( time( 0 ) );  // randomize the random number generator

   Tree2< int > intTree;
   int intVal;

   cout << "The values being placed in the tree are:\n";

   for ( int i = 1; i <= 15; ++i ) {
      intVal = rand() % 100;
      cout << intVal << ' ';
      intTree.insertNode( intVal );
   }

   cout << "\n\nThe tree is:\n";
   intTree.outputTree();

   return 0;
}
```

```
The values being placed in the tree are:
12 14 56 18 12 97 12 82 96 27 99 90 22 87 15

The tree is:
                    99
               97
                         96
                              90
                                   87
                    82
          56
                    27
                         22
               18
                    15
     14
12
     12
          12
```

SPECIAL SECTION—BUILDING YOUR OWN COMPILER

In Exercises 5.18 and 5.19, we introduced Simpletron Machine Language (SML) and you implemented a Simpletron computer simulator to execute programs written in SML. In this section, we build a compiler that converts programs written in a high-level programming language to SML. This section “ties” together the entire programming process. You will write programs in this new high-level language, compile these programs on the compiler you build and run the programs on the simulator you built in Exercise 7.19. You should make every effort to implement your compiler in an object-oriented manner.

15.26 (*The Simple Language*) Before we begin building the compiler, we discuss a simple, yet powerful, high-level language similar to early versions of the popular language BASIC. We call the language *Simple*. Every Simple *statement* consists of a *line number* and a Simple *instruction*. Line numbers must appear in ascending order. Each instruction begins with one of the following Simple *commands*: **rem**, **input**, **let**, **print**, **goto**, **if/goto** and **end** (see Fig. 15.20). All commands except **end** can be used repeatedly. Simple evaluates only integer expressions using the **+**, **-**, ***** and **/** operators. These operators have the same precedence as in C. Parentheses can be used to change the order of evaluation of an expression.

Our Simple compiler recognizes only lowercase letters. All characters in a Simple file should be lowercase (uppercase letters result in a syntax error unless they appear in a **rem** statement, in which case they are ignored). A *variable name* is a single letter. Simple does not allow descriptive variable names, so variables should be explained in remarks to indicate their use in a program. Simple uses only integer variables. Simple does not have variable declarations—merely mentioning a variable name in a program causes the variable to be declared and initialized to zero automatically. The syntax of Simple does not allow string manipulation (reading a string, writing a string, comparing strings, etc.). If a string is encountered in a Simple program (after a command other than **rem**), the compiler generates a syntax error. The first version of our compiler will assume that Simple programs are entered correctly. Exercise 15.29 asks the student to modify the compiler to perform syntax error checking.

Command	Example statement	Description
rem	**50 rem this is a remark**	Text following **rem** is for documentation purposes and is ignored by the compiler.
input	**30 input x**	Display a question mark to prompt the user to enter an integer. Read that integer from the keyboard and store the integer in **x**.
let	**80 let u = 4 * (j - 56)**	Assign **u** the value of **4 * (j - 56)**. Note that an arbitrarily complex expression can appear to the right of the equals sign.
print	**10 print w**	Display the value of **w**.
goto	**70 goto 45**	Transfer program control to line **45**.
if/goto	**35 if i == z goto 80**	Compare **i** and **z** for equality and transfer control to line **80** if the condition is true; otherwise, continue execution with the next statement.
end	**99 end**	Terminate program execution.

Fig. 15.20 Simple commands.

Simple uses the conditional **if/goto** statement and the unconditional **goto** statement to alter the flow of control during program execution. If the condition in the **if/goto** statement is true, control is transferred to a specific line of the program. The following relational and equality operators are valid in an **if/goto** statement: **<**, **>**, **<=**, **>=**, **==** and **!=**. The precedence of these operators is the same as in C++.

Let us now consider several programs that demonstrate Simple's features. The first program (Fig. 15.21) reads two integers from the keyboard, stores the values in variables **a** and **b** and computes and prints their sum (stored in variable **c**).

The program of Fig. 15.22 determines and prints the larger of two integers. The integers are input from the keyboard and stored in **s** and **t**. The **if/goto** statement tests the condition **s >= t**. If the condition is true, control is transferred to line **90** and **s** is output; otherwise, **t** is output and control is transferred to the **end** statement in line **99** where the program terminates.

Simple does not provide a repetition structure (such as C++'s **for**, **while** or **do/while**). However, Simple can simulate each of C++'s repetition structures using the **if/goto** and **goto** statements. Figure 15.23 uses a sentinel-controlled loop to calculate the squares of several integers. Each integer is input from the keyboard and stored in variable **j**. If the value entered is the sentinel
-9999, control is transferred to line **99**, where the program terminates. Otherwise, **k** is assigned the square of **j**, **k** is output to the screen and control is passed to line **20**, where the next integer is input.

```
10 rem   determine and print the sum of two integers
15 rem
```

Fig. 15.21 Simple program that determines the sum of two integers (part 1 of 2).

```
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
```

Fig. 15.21 Simple program that determines the sum of two integers (part 2 of 2).

```
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
```

Fig. 15.22 Simple program that finds the larger of two integers.

```
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
```

Fig. 15.23 Calculate the squares of several integers.

Using the sample programs of Fig. 15.21, Fig. 15.22 and Fig. 15.23 as your guide, write a Simple program to accomplish each of the following:

a) Input three integers, determine their average and print the result.
b) Use a sentinel-controlled loop to input 10 integers and compute and print their sum.
c) Use a counter-controlled loop to input 7 integers, some positive and some negative, and compute and print their average.
d) Input a series of integers and determine and print the largest. The first integer input indicates how many numbers should be processed.
e) Input 10 integers and print the smallest.
f) Calculate and print the sum of the even integers from 2 to 30.
g) Calculate and print the product of the odd integers from 1 to 9.

15.27 (*Building A Compiler; Prerequisite: Complete Exercises 5.18, 5.19, 15.12, 15.13 and 15.26*) Now that the Simple language has been presented (Exercise 15.26), we discuss how to build a Simple compiler. First, we consider the process by which a Simple program is converted to SML and executed by the Simpletron simulator (see Fig. 15.24). A file containing a Simple program is read by the compiler and converted to SML code. The SML code is output to a file on disk, in which SML instructions appear one per line. The SML file is then loaded into the Simpletron simulator, and the results are sent to a file on disk and to the screen. Note that the Simpletron program developed in Exercise 5.19 took its input from the keyboard. It must be modified to read from a file so it can run the programs produced by our compiler.

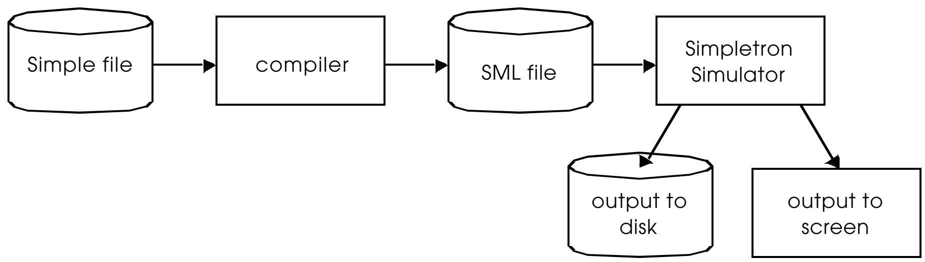

Fig. 15.24 Writing, compiling and executing a Simple language program.

The Simple compiler performs two *passes* of the Simple program to convert it to SML. The first pass constructs a *symbol table* (object) in which every *line number* (object), *variable name* (object) and *constant* (object) of the Simple program is stored with its type and corresponding location in the final SML code (the symbol table is discussed in detail below). The first pass also produces the corresponding SML instruction object(s) for each of the Simple statements (object, etc.). As we will see, if the Simple program contains statements that transfer control to a line later in the program, the first pass results in an SML program containing some "unfinished" instructions. The second pass of the compiler locates and completes the unfinished instructions, and outputs the SML program to a file.

First Pass

The compiler begins by reading one statement of the Simple program into memory. The line must be separated into its individual *tokens* (i.e., "pieces" of a statement) for processing and compilation (standard library function **strtok** can be used to facilitate this task). Recall that every statement begins with a line number followed by a command. As the compiler breaks a statement into tokens, if the token is a line number, a variable or a constant, it is placed in the symbol table. A line number is placed in the symbol table only if it is the first token in a statement. The **symbolTable** object is an array of **tableEntry** objects representing each symbol in the program. There is no restriction on the number of symbols that can appear in the program. Therefore, the **symbolTable** for a particular program could be large. Make the **symbolTable** a 100-element array for now. You can increase or decrease its size once the program is working.

Each **tableEntry** object contains three members. Member **symbol** is an integer containing the ASCII representation of a variable (remember that variable names are single characters), a line number or a constant. Member **type** is one of the following characters indicating the symbol's type: **'C'** for constant, **'L'** for line number and **'V'** for variable. Member **location** contains the Simpletron memory location (**00** to **99**) to which the symbol refers. Simpletron memory is an array of 100 integers in which SML instructions and data are stored. For a line number, the location is the element in the Simpletron memory array at which the SML instructions for the Simple statement begin. For a variable or constant, the location is the element in the Simpletron memory array in which the variable or constant is stored. Variables and constants are allocated from the end of Simpletron's memory backwards. The first variable or constant is stored in location at **99**, the next in location at **98**, etc.

The symbol table plays an integral part in converting Simple programs to SML. We learned in Chapter 5 that an SML instruction is a four-digit integer composed of two parts—the *operation code* and the *operand*. The operation code is determined by commands in Simple. For example, the simple command **input** corresponds to SML operation code **10** (read), and the Simple command **print** corresponds to SML operation code **11** (write). The operand is a memory location containing the data on which the operation code performs its task (e.g., operation code **10** reads a value from the keyboard and stores it in the memory location specified by the operand). The compiler searches **symbolTable** to determine the Simpletron memory location for each symbol so the corresponding location can be used to complete the SML instructions.

The compilation of each Simple statement is based on its command. For example, after the line number in a **rem** statement is inserted in the symbol table, the remainder of the statement is ignored by the compiler because a remark is for documentation purposes only. The **input**, **print**, **goto** and **end** statements correspond to the SML *read*, *write*, *branch* (to a specific location) and *halt* instructions. Statements containing these Simple commands are converted directly to SML (note that a **goto** statement may contain an unresolved reference if the specified line number refers to a statement further into the Simple program file; this is sometimes called a forward reference).

When a **goto** statement is compiled with an unresolved reference, the SML instruction must be *flagged* to indicate that the second pass of the compiler must complete the instruction. The flags are stored in 100-element array **flags** of type **int** in which each element is initialized to **-1**. If the memory location to which a line number in the Simple program refers is not yet known (i.e., it is not in the symbol table), the line number is stored in array **flags** in the element with the same subscript as the incomplete instruction. The operand of the incomplete instruction is set to **00** temporarily. For example, an unconditional branch instruction (making a forward reference) is left as **+4000** until the second pass of the compiler. The second pass of the compiler is described shortly.

Compilation of **if/goto** and **let** statements is more complicated than other statements—they are the only statements that produce more than one SML instruction. For an **if/goto**, the compiler produces code to test the condition and to branch to another line if necessary. The result of the branch could be an unresolved reference. Each of the relational and equality operators can be simulated using SML's *branch zero* and *branch negative* instructions (or a combination of both).

For a **let** statement, the compiler produces code to evaluate an arbitrarily complex arithmetic expression consisting of integer variables and/or constants. Expressions should separate each operand and operator with spaces. Exercises 15.12 and 15.13 presented the infix-to-postfix conversion algorithm and the postfix evaluation algorithm used by compilers to evaluate expressions. Before proceeding with your compiler, you should complete each of these exercises. When a compiler encounters an expression, it converts the expression from infix notation to postfix notation and then evaluates the postfix expression.

How is it that the compiler produces the machine language to evaluate an expression containing variables? The postfix evaluation algorithm contains a "hook" where the compiler can generate SML instructions rather than actually evaluating the expression. To enable this "hook" in the compiler, the postfix evaluation algorithm must be modified to search the symbol table for each symbol it encounters (and possibly insert it), determine the symbol's corresponding memory location and *push the memory location onto the stack (instead of the symbol)*. When an operator is encountered in the postfix expression, the two memory locations at the top of the stack are popped and machine language for effecting the operation is produced using the memory locations as operands. The result of each subexpression is stored in a temporary location in memory and pushed back onto the stack so the evaluation of the postfix expression can continue. When postfix evaluation is complete, the memory location containing the result is the only location left on the stack. This is popped and SML instructions are generated to assign the result to the variable at the left of the **let** statement.

Second Pass

The second pass of the compiler performs two tasks: resolve any unresolved references and output the SML code to a file. Resolution of references occurs as follows:

a) Search the **flags** array for an unresolved reference (i.e., an element with a value other than **-1**).
b) Locate the object in array **symbolTable**, containing the symbol stored in the **flags** array (be sure that the type of the symbol is **'L'** for line number).
c) Insert the memory location from member **location** into the instruction with the unresolved reference (remember that an instruction containing an unresolved reference has operand **00**).
d) Repeat steps 1, 2 and 3 until the end of the **flags** array is reached.

After the resolution process is complete, the entire array containing the SML code is output to a disk file with one SML instruction per line. This file can be read by the Simpletron for execution (after the simulator is modified to read its input from a file). Compiling your first Simple program into an SML file and then executing that file should give you a real sense of personal accomplishment.

A Complete Example

The following example illustrates a complete conversion of a Simple program to SML as it will be performed by the Simple compiler. Consider a Simple program that inputs an integer and sums the values from 1 to that integer. The program and the SML instructions produced by the first pass of the Simple compiler are illustrated in Fig. 15.25. The symbol table constructed by the first pass is shown in Fig. 15.26.

Simple program	SML location and instruction	Description
5 rem sum 1 to x	*none*	**rem** ignored

Fig. 15.25 SML instructions produced after the compiler's first pass.

Simple program	SML location and instruction	Description
`10 input x`	`00 +1099`	read **x** into location **99**
`15 rem check y == x`	*none*	**rem** ignored
`20 if y == x goto 60`	`01 +2098`	load **y** (**98**) into accumulator
	`02 +3199`	sub **x** (**99**) from accumulator
	`03 +4200`	branch zero to unresolved location
`25 rem increment y`	*none*	**rem** ignored
`30 let y = y + 1`	`04 +2098`	load **y** into accumulator
	`05 +3097`	add **1** (**97**) to accumulator
	`06 +2196`	store in temporary location **96**
	`07 +2096`	load from temporary location **96**
	`08 +2198`	store accumulator in **y**
`35 rem add y to total`	*none*	**rem** ignored
`40 let t = t + y`	`09 +2095`	load **t** (**95**) into accumulator
	`10 +3098`	add **y** to accumulator
	`11 +2194`	store in temporary location **94**
	`12 +2094`	load from temporary location **94**
	`13 +2195`	store accumulator in **t**
`45 rem loop y`	*none*	**rem** ignored
`50 goto 20`	`14 +4001`	branch to location **01**
`55 rem output result`	*none*	**rem** ignored
`60 print t`	`15 +1195`	output **t** to screen
`99 end`	`16 +4300`	terminate execution

Fig. 15.25 SML instructions produced after the compiler's first pass.

Symbol	Type	Location
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**

Fig. 15.26 Symbol table for program of Fig. 15.25 (part 1 of 2).

Symbol	Type	Location
`45`	`L`	`14`
`50`	`L`	`14`
`55`	`L`	`15`
`60`	`L`	`15`
`99`	`L`	`16`

Fig. 15.26 Symbol table for program of Fig. 15.25 (part 2 of 2).

Most Simple statements convert directly to single SML instructions. The exceptions in this program are remarks, the **`if/goto`** statement in line **`20`** and the **`let`** statements. Remarks do not translate into machine language. However, the line number for a remark is placed in the symbol table in case the line number is referenced in a **`goto`** statement or an **`if/goto`** statement. Line **`20`** of the program specifies that if the condition **`y == x`** is true, program control is transferred to line **`60`**. Because line **`60`** appears later in the program, the first pass of the compiler has not as yet placed **`60`** in the symbol table (statement line numbers are placed in the symbol table only when they appear as the first token in a statement). Therefore, it is not possible at this time to determine the operand of the SML *branch zero* instruction at location **`03`** in the array of SML instructions. The compiler places **`60`** in location **`03`** of the **`flags`** array to indicate that the second pass completes this instruction.

We must keep track of the next instruction location in the SML array because there is not a one-to-one correspondence between Simple statements and SML instructions. For example, the **`if/goto`** statement of line **`20`** compiles into three SML instructions. Each time an instruction is produced, we must increment the *instruction counter* to the next location in the SML array. Note that the size of Simpletron's memory could present a problem for Simple programs with many statements, variables and constants. It is conceivable that the compiler will run out of memory. To test for this case, your program should contain a *data counter* to keep track of the location at which the next variable or constant will be stored in the SML array. If the value of the instruction counter is larger than the value of the data counter, the SML array is full. In this case, the compilation process should terminate and the compiler should print an error message indicating that it ran out of memory during compilation. This serves to emphasize that although the programmer is freed from the burdens of managing memory by the compiler, the compiler itself must carefully determine the placement of instructions and data in memory, and must check for such errors as memory being exhausted during the compilation process.

A Step-by-Step View of the Compilation Process

Let us now walk through the compilation process for the Simple program in Fig. 15.25. The compiler reads the first line of the program

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

into memory. The first token in the statement (the line number) is determined using **`strtok`** (see Chapters 5 and 16 for a discussion of C++'s string manipulation functions). The token returned by **`strtok`** is converted to an integer using **`atoi`** so the symbol **`5`** can be located in the symbol table. If the symbol is not found, it is inserted in the symbol table. Since we are at the beginning of the program and this is the first line, no symbols are in the table yet. So, **`5`** is inserted into the symbol table as type **`L`** (line number) and assigned the first location in SML array (**`00`**). Although this line is a remark, a space in the symbol table is still allocated for the line number (in case it is referenced by a **`goto`** or an **`if/goto`**). No SML instruction is generated for a **`rem`** statement, so the instruction counter is not incremented.

The statement

```
10 input x
```

is tokenized next. The line number **`10`** is placed in the symbol table as type **`L`** and assigned the first location in the SML array (**`00`** because a remark began the program so the instruction counter is currently **`00`**). The command **`input`** indicates that the next token is a variable (only a variable can appear in an **`input`** statement). Because **`input`** corresponds directly to an SML operation code, the compiler has to determine the location of **`x`** in the SML array. Symbol **`x`** is not found in the symbol table. So, it is inserted into the symbol table as the ASCII representation of **`x`**, given type **`V`**, and assigned location **`99`** in the SML array (data storage begins at **`99`** and is allocated backwards). SML code can now be generated for this statement. Operation code **`10`** (the SML read operation code) is multiplied by 100, and the location of **`x`** (as determined in the symbol table) is added to complete the instruction. The instruction is then stored in the SML array at location **`00`**. The instruction counter is incremented by 1 because a single SML instruction was produced.

The statement

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

is tokenized next. The symbol table is searched for line number **15** (which is not found). The line number is inserted as type **L** and assigned the next location in the array, **01** (remember that **rem** statements do not produce code, so the instruction counter is not incremented).

The statement

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

is tokenized next. Line number **20** is inserted in the symbol table and given type **L** with the next location in the SML array **01**. The command **if** indicates that a condition is to be evaluated. The variable **y** is not found in the symbol table, so it is inserted and given the type **V** and the SML location **98**. Next, SML instructions are generated to evaluate the condition. Since there is no direct equivalent in SML for the **if/goto**, it must be simulated by performing a calculation using **x** and **y** and branching based on the result. If **y** is equal to **x**, the result of subtracting **x** from **y** is zero, so the *branch zero* instruction can be used with the result of the calculation to simulate the **if/goto** statement. The first step requires that **y** be loaded (from SML location **98**) into the accumulator. This produces the instruction **01 +2098**. Next, **x** is subtracted from the accumulator. This produces the instruction **02 +3199**. The value in the accumulator may be zero, positive or negative. Since the operator is **==**, we want to *branch zero.* First, the symbol table is searched for the branch location (**60** in this case), which is not found. So, **60** is placed in the **flags** array at location **03**, and the instruction **03 +4200** is generated (we cannot add the branch location, because we have not assigned a location to line **60** in the SML array yet). The instruction counter is incremented to **04**.

The compiler proceeds to the statement

```
25 rem   increment y
```

The line number **25** is inserted in the symbol table as type **L** and assigned SML location **04**. The instruction counter is not incremented.

When the statement

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

is tokenized, the line number **30** is inserted in the symbol table as type **L** and assigned SML location **04**. Command **let** indicates that the line is an assignment statement. First, all the symbols on the line are inserted in the symbol table (if they are not already there). The integer **1** is added to the symbol table as type **C** and assigned SML location **97**. Next, the right side of the assignment is converted from infix to postfix notation. Then the postfix expression (**y 1 +**) is evaluated. Symbol **y** is located in the symbol table and its corresponding memory location is pushed onto the stack. Symbol **1** is also located in the symbol table and its corresponding memory location is pushed onto the stack. When the operator **+** is encountered, the postfix evaluator pops the stack into the right operand of the operator, pops the stack again into the left operand of the operator and then produces the SML instructions

04 +2098 *(load* **y***)*
05 +3097 *(add* **1***)*

The result of the expression is stored in a temporary location in memory (**96**) with instruction

06 +2196 *(store temporary)*

and the temporary location is pushed on the stack. Now that the expression has been evaluated, the result must be stored in **y** (i.e., the variable on the left side of **=**). So, the temporary location is loaded into the accumulator and the accumulator is stored in **y** with the instructions

07 +2096 *(load temporary)*
08 +2198 *(store* **y***)*

The reader will immediately notice that SML instructions appear to be redundant. We will discuss this issue shortly.

When the statement

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

is tokenized, line number **35** is inserted in the symbol table as type **L** and assigned location **09**.

The statement

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

is similar to line **30**. The variable **t** is inserted in the symbol table as type **V** and assigned SML location **95**. The instructions follow the same logic and format as line **30**, and the instructions **09 +2095**, **10 +3098**, **11 +2194**, **12 +2094** and **13 +2195** are generated. Note that the result of **t + y** is assigned to temporary location **94** before being assigned to **t** (**95**). Once again, the reader will note that the instructions in memory locations **11** and **12** appear to be redundant. Again, we will discuss this shortly.

The statement

```
45 rem   loop y
```

is a remark, so line **45** is added to the symbol table as type **L** and assigned SML location **14**.

The statement

```
50 goto 20
```

transfers control to line **20**. Line number **50** is inserted in the symbol table as type **L** and assigned SML location **14**. The equivalent of **goto** in SML is the *unconditional branch* (**40**) instruction that transfers control to a specific SML location. The compiler searches the symbol table for line **20** and finds that it corresponds to SML location **01**. The operation code (**40**) is multiplied by 100, and location **01** is added to it to produce the instruction **14 +4001**.

The statement

```
55 rem   output result
```

is a remark, so line **55** is inserted in the symbol table as type **L** and assigned SML location **15**.

The statement

```
60 print t
```

is an output statement. Line number **60** is inserted in the symbol table as type **L** and assigned SML location **15**. The equivalent of **print** in SML is operation code **11** (*write*). The location of **t** is determined from the symbol table and added to the result of the operation code multiplied by 100.

The statement

```
99 end
```

is the final line of the program. Line number **99** is stored in the symbol table as type **L** and assigned SML location **16**. The **end** command produces the SML instruction **+4300** (**43** is *halt* in SML), which is written as the final instruction in the SML memory array.

This completes the first pass of the compiler. We now consider the second pass. The **flags** array is searched for values other than **-1**. Location **03** contains **60**, so the compiler knows that instruction **03** is incomplete. The compiler completes the instruction by searching the symbol table for **60**, determining its location and adding the location to the incomplete instruction. In this case, the search determines that line **60** corresponds to SML location **15**, so the completed instruction **03 +4215** is produced, replacing **03 +4200**. The Simple program has now been compiled successfully.

To build the compiler, you will have to perform each of the following tasks:

a) Modify the Simpletron simulator program you wrote in Exercise 5.19 to take its input from a file specified by the user (see Chapter 14). The simulator should output its results to a disk file in the same format as the screen output. Convert the simulator to be an object-oriented program. In particular, make each part of the hardware an object. Arrange the instruction types into a class hierarchy using inheritance. Then execute the program polymorphically by telling each instruction to execute itself with an **executeInstruction** message.

b) Modify the infix-to-postfix conversion algorithm of Exercise 15.12 to process multi-digit integer operands and single-letter variable name operands. (Hint: Standard library function **strtok** can be used to locate each constant and variable in an expression, and constants can be converted from strings to integers using standard library function **atoi**.) (Note: The data representation of the postfix expression must be altered to support variable names and integer constants.)

c) Modify the postfix evaluation algorithm to process multidigit integer operands and variable name operands. Also, the algorithm should now implement the "hook" discussed above so that SML instructions are produced rather than directly evaluating the expression. (Hint: Standard library function **`strtok`** can be used to locate each constant and variable in an expression, and constants can be converted from strings to integers using standard library function **`atoi`**.) (*Note:* The data representation of the postfix expression must be altered to support variable names and integer constants.)

d) Build the compiler. Incorporate parts (b) and (c) for evaluating expressions in **`let`** statements. Your program should contain a function that performs the first pass of the compiler and a function that performs the second pass of the compiler. Both functions can call other functions to accomplish their tasks. Make your compiler as object oriented as possible.

```
// STACKND.H
// Definition of template class StackNode
#ifndef STACKND_H
#define STACKND_H

template < class T > class Stack;

template < class T >
class StackNode {
   friend class Stack< T >;
public:
   StackNode( const T & = 0, StackNode * = 0 );
   T getData() const;
private:
   T data;
   StackNode *nextPtr;
};

// Member function definitions for class StackNode
template < class T >
StackNode< T >::StackNode( const T &d, StackNode< T > *ptr )
{
   data = d;
   nextPtr = ptr;
}

template < class T >
T StackNode< T >::getData() const { return data; }

#endif
```

```
// STACK.H
// Definition of class Stack
#ifndef STACK_H
#define STACK_H

#include <iostream>
using std::cout;

#include <cassert>
#include "stacknd.h"

template < class T >
class Stack {

public:
   Stack();                // default constructor
   ~Stack();               // destructor
   void push( T & );       // insert item in stack
```

```
   T pop();                // remove item from stack
   bool isEmpty() const;  // is the stack empty?
   T stackTop() const;    // return the top element of stack
   void print() const;    // output the stack
private:
   StackNode< T > *topPtr;     // pointer to fist StackNode
};

// Member function definitions for class Stack
template < class T >
Stack< T >::Stack() { topPtr = 0; }

template < class T >
Stack< T >::~Stack()
{
   StackNode< T > *tempPtr, *currentPtr = topPtr;

   while ( currentPtr != 0 ) {
      tempPtr = currentPtr;
      currentPtr = currentPtr -> nextPtr;
      delete tempPtr;
   }
}

template < class T >
void Stack< T >::push( T &d )
{
   StackNode< T > *newPtr = new StackNode< T >( d, topPtr );

   assert( newPtr != 0 );  // was memory allocated?
   topPtr = newPtr;
}

template < class T >
T Stack< T >::pop()
{
   assert( !isEmpty() );

   StackNode< T > *tempPtr = topPtr;

   topPtr = topPtr -> nextPtr;
   T poppedValue = tempPtr -> data;
   delete tempPtr;
   return poppedValue;
}

template < class T >
bool Stack< T >::isEmpty() const { return topPtr == 0; }

template < class T >
T Stack< T >::stackTop() const
{ return !isEmpty() ? topPtr -> data : 0; }

template < class T >
void Stack< T >::print() const
{
   StackNode< T > *currentPtr = topPtr;

   if ( isEmpty() )           // Stack is empty
      cout << "Stack is empty\n";
   else {                    // Stack is not empty
      cout << "The stack is:\n";
```

```
      while ( currentPtr != 0 ) {
         cout << currentPtr -> data << ' ';
         currentPtr = currentPtr -> nextPtr;
      }

      cout << '\n';
   }
}

#endif
```

```
// compiler.h

const int MAXIMUM = 81;             // maximum length for lines
const int SYMBOLTABLESIZE = 100;    // maximum size of symbol table
const int MEMORYSIZE = 100;         // maximum Simpletron memory

// Definition of structure for symbol table entries
struct TableEntry {
   int location;              // SML memory location 00 to 99
   char type;                 // 'C' = constant, 'V' = variable,
   int symbol;                // or 'L' = line number
};

// Function prototypes for compiler functions
int checkSymbolTable( TableEntry *, int, char );
int checkOperand( TableEntry *, char *, int *, int *, int * );
void addToSymbolTable( char, int, int, TableEntry *, int );
void addLineToFlags( int, int *, int *, const int * );
void compile( ifstream &, char * );
void printOutput( const int [], char * );
void lineNumber( char *, TableEntry *, int, int );
void initArrays( int *, int *, TableEntry * );
void firstPass( int *, int *, TableEntry *, ifstream & );
void secondPass( int *, int *, TableEntry * );
void separateToken( char *, int *, int *, TableEntry *, int *, int * );
void keyWord( char *, int *, int *, TableEntry *, int *, int * );
void keyLet( char *, int *, TableEntry *, int *, int * );
void keyInput( char *, int *, TableEntry *, int *, int * );
void keyPrint( char *, int *, TableEntry *, int *, int * );
void keyGoto( char *, int *, int *, TableEntry *, int * );
void keyIfGoto( char *, int *, int *, TableEntry *, int *, int * );
void evaluateExpression( int, int, char *, int *, TableEntry *,
                         int *, int *, char * );
int createLetSML( int, int, int *, int *, int *, char );
void infixToPostfix( char *, char *, int, TableEntry *, int *, int *, int * );
void evaluatePostfix( char *, int *, int *, int *, int, TableEntry * );
bool isOperator( char );
bool precedence( char, char );

// Simpletron Machine Language (SML) Operation Codes
enum SMLOperationCodes { READ = 10, WRITE = 11, LOAD = 20, STORE = 21, ADD = 30,
                         SUBTRACT = 31, DIVIDE = 32, MULTIPLY = 33, BRANCH = 40,
                         BRANCHNEG = 41, BRANCHZERO = 42, HALT = 43 };
```

```
// Exercise 15.27 Solution
// Non-optimized version.
#include <iostream>

```

```
using std::cout;
using std::cin;
using std::ios;
using std::cerr;

#include <iomanip>

using std::setw;

#include <fstream>

using std::ifstream;
using std::ofstream;

#include <cstring>
#include <cctype>
#include <cstdlib>
#include "stack.h"

#include "compiler.h"

int main()
{
   char inFileName[ 15 ] = "", outFileName[ 15 ] = "";
   int last = 0;

   cout << "Enter Simple file to be compiled: ";
   cin >> setw( 15 ) >> inFileName;

   while ( isalnum( inFileName[ last ] ) != 0 ) {
      outFileName[ last ] = inFileName[ last ];
      last++;            // note the last occurance
   }

   outFileName[ last ] = '\0';     // append a NULL character
   strcat( outFileName, ".sml" );  // add .sml to name

   ifstream inFile( inFileName, ios::in );

   if ( inFile )
      compile( inFile, outFileName );
   else
      cerr << "File not opened. Program execution terminating.\n";

   return 0;
}

// compile function calls the first pass and the second pass
void compile( ifstream &input, char *outFileName )
{
   TableEntry symbolTable[ SYMBOLTABLESIZE ]; // symbol table
   int flags[ MEMORYSIZE ];            // array for forward references
   int machineArray[ MEMORYSIZE ];     // array for SML instructions

   initArrays( flags, machineArray, symbolTable );

   firstPass( flags, machineArray, symbolTable, input );
   secondPass( flags, machineArray, symbolTable );

   printOutput( machineArray, outFileName );
}

```

```
// firstPass constructs the symbol table, creates SML, and flags unresolved
// references for goto and if/goto statements.
void firstPass( int flags[], int machineArray[], TableEntry symbolTable[],
                ifstream &input )
{
   char array[ MAXIMUM ];              // array to copy a Simple line
   int n = MAXIMUM;                    // required for fgets()
   int dataCounter = MEMORYSIZE - 1; // 1st data location in machineArray
   int instCounter = 0;                // 1st instruction location in machineArray

   input.getline( array, n );

   while ( !input.eof() ) {
      separateToken( array, flags, machineArray, symbolTable, &dataCounter,
                     &instCounter );
      input.getline( array, n );
   }
}

// Separate Tokens tokenizes a Simple statement, process the line number,
// and passes the next token to keyWord for processing.
void separateToken( char array[], int flags[], int machineArray[],
                    TableEntry symbolTable[], int *dataCounterPtr,
                    int *instCounterPtr )
{
   char *tokenPtr = strtok( array, " " );     // tokenize line
   lineNumber( tokenPtr, symbolTable, *instCounterPtr, *dataCounterPtr );
   tokenPtr = strtok( 0, " \n" );       // get next token
   keyWord( tokenPtr, flags, machineArray, symbolTable, dataCounterPtr,
            instCounterPtr );
}

// checkSymbolTable searches the symbol table and returns
// the symbols SML location or a -1 if not found.
int checkSymbolTable( TableEntry symbolTable[], int symbol, char type )
{
   for ( int loop = 0; loop < SYMBOLTABLESIZE; ++loop )
      if ( ( symbol == symbolTable[ loop ].symbol ) &&
           ( type == symbolTable[ loop ].type ) )
         return symbolTable[ loop ].location;  // return SML location

   return -1;                                  // symbol not found
}

// lineNumber processes line numbers
void lineNumber( char *tokenPtr, TableEntry symbolTable[],
                 int instCounter, int dataCounter )
{
   const char type = 'L';
   int symbol;

   if ( isdigit( tokenPtr[ 0 ] ) ) {
      symbol = atoi( tokenPtr );

      if ( -1 == checkSymbolTable( symbolTable, symbol, type ) )
         addToSymbolTable( type, symbol, dataCounter, symbolTable,
                           instCounter );
   }

}

// keyWord determines the key word type and calls the appropriate function
void keyWord( char *tokenPtr, int flags[], int machineArray[],
```

```
                 TableEntry symbolTable[], int *dataCounterPtr,
                 int *instCounterPtr )
{
   if ( strcmp( tokenPtr, "rem" ) == 0 )
      ; // no instructions are generated by comments
   else if ( strcmp( tokenPtr, "input" ) == 0 ) {
      tokenPtr = strtok( 0, " " );  // assign pointer to next token
      keyInput( tokenPtr, machineArray, symbolTable, dataCounterPtr,
                instCounterPtr );
   }
   else if ( strcmp( tokenPtr, "print" ) == 0 ) {
      tokenPtr = strtok( 0, " " );  // assign pointer to next token
      keyPrint( tokenPtr, machineArray, symbolTable, dataCounterPtr,
                instCounterPtr );
   }
   else if ( strcmp( tokenPtr, "goto" ) == 0 ) {
      tokenPtr = strtok( 0, " " );  // assign pointer to next token
      keyGoto( tokenPtr, flags, machineArray, symbolTable, instCounterPtr );
   }
   else if ( strcmp( tokenPtr, "if" ) == 0 ) {
      tokenPtr = strtok( 0, " " );  // assign pointer to next token
      keyIfGoto( tokenPtr, flags, machineArray, symbolTable, dataCounterPtr,
                 instCounterPtr );
   }
   else if ( strcmp( tokenPtr, "end" ) == 0 ) {
      machineArray[ *instCounterPtr ] = HALT * 100;
      ++( *instCounterPtr );
      tokenPtr = 0;       // assign tokenPtr to 0
   }
   else if ( strcmp( tokenPtr, "let" ) == 0 ) {
      tokenPtr = strtok( 0, " " );  // assign pointer to next token
      keyLet( tokenPtr, machineArray, symbolTable, dataCounterPtr,
              instCounterPtr );
   }
}

// keyInput process input keywords
void keyInput( char *tokenPtr, int machineArray[], TableEntry symbolTable[],
               int *dataCounterPtr, int *instCounterPtr )
{
   const char type = 'V';

   machineArray[ *instCounterPtr ] = READ * 100;
   int symbol = tokenPtr[ 0 ];
   int tableTest = checkSymbolTable( symbolTable, symbol, type );

   if ( -1 == tableTest ) {
      addToSymbolTable( type, symbol, *dataCounterPtr, symbolTable,
                        *instCounterPtr );
      machineArray[ *instCounterPtr ] += *dataCounterPtr;
      --( *dataCounterPtr );
   }
   else
      machineArray[ *instCounterPtr ] += tableTest;

   ++( *instCounterPtr );
}

// keyPrint process print keywords
void keyPrint( char *tokenPtr, int machineArray[], TableEntry symbolTable[],
               int *dataCounterPtr, int *instCounterPtr )
{
```

```
   const char type = 'V';

   machineArray[ *instCounterPtr ] = WRITE * 100;
   int symbol = tokenPtr[ 0 ];
   int tableTest = checkSymbolTable( symbolTable, symbol, type );

   if ( -1 == tableTest ) {
      addToSymbolTable( type, symbol, *dataCounterPtr, symbolTable,
                        *instCounterPtr );
      machineArray[*instCounterPtr] += *dataCounterPtr;
      --( *dataCounterPtr );
   }
   else
      machineArray[ *instCounterPtr ] += tableTest;

   ++( *instCounterPtr );
}

// keyGoto process goto keywords
void keyGoto( char *tokenPtr, int flags[], int machineArray[],
              TableEntry symbolTable[], int *instCounterPtr )
{
   const char type = 'L';

   machineArray[*instCounterPtr] = BRANCH * 100;
   int symbol = atoi( tokenPtr );
   int tableTest = checkSymbolTable( symbolTable, symbol, type );
   addLineToFlags( tableTest, symbol, flags, machineArray, instCounterPtr );
   ++( *instCounterPtr );
}

// keyIfGoto process if/goto commands
void keyIfGoto( char *tokenPtr, int flags[], int machineArray[],
                TableEntry symbolTable[], int *dataCounterPtr,
                int *instCounterPtr )
{
   int operand1Loc = checkOperand( symbolTable, tokenPtr, dataCounterPtr,
                                   instCounterPtr, machineArray );

   char *operatorPtr = strtok( 0, " " );   // get the operator

   tokenPtr = strtok( 0, " " );  // get the right operand of comparison operator

   int operand2Loc = checkOperand( symbolTable, tokenPtr, dataCounterPtr,
                                   instCounterPtr, machineArray );

   tokenPtr = strtok( 0, " " );  // read in the goto keyword

   char *gotoLinePtr = strtok( 0, " " );  // read in the goto line number

   evaluateExpression( operand1Loc, operand2Loc, operatorPtr, machineArray,
                       symbolTable, instCounterPtr, flags, gotoLinePtr );
}

// checkOperand ensures that the operands of an if/goto statement are
// in the symbol table.
int checkOperand( TableEntry symbolTable[], char *symPtr, int *dataCounterPtr,
                  int *instCounterPtr, int machineArray[] )

{
   char type;
   int tableTest, operand, temp;

```

```
   if ( isalpha( symPtr[ 0 ] ) ) {
      type = 'V';
      operand = symPtr[ 0 ];
      tableTest = checkSymbolTable( symbolTable, operand, type );

      if ( tableTest == -1 ) {
         addToSymbolTable( type, operand, *dataCounterPtr, symbolTable,
                           *instCounterPtr );
         temp = *dataCounterPtr;
         --( *dataCounterPtr );
         return temp;
      }
      else
         return tableTest;
   }
   // if the symbol is a digit or a signed digit
   else if ( isdigit( symPtr[ 0 ] ) ||
            ( ( symPtr[ 0 ] == '-' || symPtr[ 0 ] == '+' ) &&
            isdigit( symPtr[ 1 ] ) != 0 ) ) {
      type = 'C';
      operand = atoi( symPtr );
      tableTest = checkSymbolTable( symbolTable, operand, type );

      if ( tableTest == -1 ) {
         addToSymbolTable( type, operand, *dataCounterPtr, symbolTable,
                          *instCounterPtr );
         machineArray[ *dataCounterPtr ] = operand;
         temp = *dataCounterPtr;
         --( *dataCounterPtr );
         return temp ;
      }
      else
         return tableTest;
   }

   return 0;       // default return for compilation purposes
}

// evaluateExpression creates SML for conditional operators
void evaluateExpression( int operator1Loc, int operator2Loc, char *operandPtr,
                         int machineArray[], TableEntry symbolTable[],
                         int *instCounterPtr, int flags[], char *gotoLinePtr )
{
   const char type = 'L';
   int tableTest, symbol;

   if ( strcmp( operandPtr, "==" ) == 0 ) {
      machineArray[ *instCounterPtr ] = LOAD * 100;
      machineArray[ *instCounterPtr ] += operator1Loc;
      ++( *instCounterPtr );

      machineArray[ *instCounterPtr ] = SUBTRACT * 100;
      machineArray[ *instCounterPtr ] += operator2Loc;
      ++( *instCounterPtr );

      machineArray[ *instCounterPtr ] = BRANCHZERO * 100;

      symbol = atoi( gotoLinePtr );

      tableTest = checkSymbolTable( symbolTable, symbol, type );
      addLineToFlags( tableTest, symbol, flags, machineArray, instCounterPtr );
      ++( *instCounterPtr );
   }
```

```
    else if ( strcmp( operandPtr, "!=" ) == 0 ) {
       machineArray[ *instCounterPtr ] = LOAD * 100;
       machineArray[ *instCounterPtr ] += operator2Loc;
       ++( *instCounterPtr );

       machineArray[ *instCounterPtr ] = SUBTRACT * 100;
       machineArray[ *instCounterPtr ] += operator1Loc;
       ++( *instCounterPtr );

       machineArray[ *instCounterPtr ] = BRANCHNEG * 100;

       symbol = atoi( gotoLinePtr );
       tableTest = checkSymbolTable( symbolTable, symbol, type );

       addLineToFlags( tableTest, symbol, flags, machineArray, instCounterPtr );

       ++( *instCounterPtr );

       machineArray[ *instCounterPtr ] = LOAD * 100;
       machineArray[ *instCounterPtr ] += operator1Loc;
       ++( *instCounterPtr );

       machineArray[ *instCounterPtr ] = SUBTRACT * 100;
       machineArray[ *instCounterPtr ] += operator2Loc;
       ++( *instCounterPtr );

       machineArray[ *instCounterPtr ] = BRANCHNEG * 100;

       symbol = atoi( gotoLinePtr );
       tableTest = checkSymbolTable( symbolTable, symbol, type );

       addLineToFlags( tableTest, symbol, flags, machineArray, instCounterPtr );

       ++( *instCounterPtr );
    }
    else if ( strcmp( operandPtr, ">" ) == 0 ) {
       machineArray[ *instCounterPtr ] = LOAD * 100;
       machineArray[ *instCounterPtr ] += operator2Loc;
       ++( *instCounterPtr );

       machineArray[ *instCounterPtr ] = SUBTRACT * 100;
       machineArray[ *instCounterPtr ] += operator1Loc;
       ++( *instCounterPtr );

       machineArray[ *instCounterPtr ] = BRANCHNEG * 100;

       symbol = atoi( gotoLinePtr );
       tableTest = checkSymbolTable( symbolTable, symbol, type );

       addLineToFlags( tableTest, symbol, flags, machineArray, instCounterPtr );
       ++( *instCounterPtr );
    }
    else if ( strcmp( operandPtr, "<" ) == 0 ) {
       machineArray[ *instCounterPtr ] = LOAD * 100;
       machineArray[ *instCounterPtr ] += operator1Loc;
       ++( *instCounterPtr );

       machineArray[ *instCounterPtr ] = SUBTRACT * 100;
       machineArray[ *instCounterPtr ] += operator2Loc;
       ++( *instCounterPtr );

       machineArray[ *instCounterPtr ] = BRANCHNEG * 100;
```

```

         symbol = atoi( gotoLinePtr );
         tableTest = checkSymbolTable( symbolTable, symbol, type );

         addLineToFlags( tableTest, symbol, flags, machineArray, instCounterPtr );
         ++( *instCounterPtr );
      }
      else if ( strcmp( operandPtr, ">=" ) == 0 ) {
         machineArray[ *instCounterPtr ] = LOAD * 100;
         machineArray[ *instCounterPtr ] += operator2Loc;
         ++( *instCounterPtr );

         machineArray[ *instCounterPtr ] = SUBTRACT * 100;
         machineArray[ *instCounterPtr ] += operator1Loc;
         ++( *instCounterPtr );

         machineArray[ *instCounterPtr ] = BRANCHNEG * 100;

         symbol = atoi( gotoLinePtr );
         tableTest = checkSymbolTable( symbolTable, symbol, type );

         addLineToFlags( tableTest, symbol, flags, machineArray, instCounterPtr );
         ++( *instCounterPtr );

         machineArray[ *instCounterPtr ] = BRANCHZERO * 100;

         addLineToFlags( tableTest, symbol, flags, machineArray, instCounterPtr );
         ++( *instCounterPtr );
      }
      else if ( strcmp( operandPtr, "<=" ) == 0 ) {
         machineArray[ *instCounterPtr ] = LOAD * 100;
         machineArray[ *instCounterPtr ] += operator1Loc;
         ++( *instCounterPtr );

         machineArray[ *instCounterPtr ] = SUBTRACT * 100;
         machineArray[ *instCounterPtr ] += operator2Loc;
         ++( *instCounterPtr );

         machineArray[ *instCounterPtr ] = BRANCHNEG * 100;

         symbol = atoi( gotoLinePtr );
         tableTest = checkSymbolTable( symbolTable, symbol, type );

         addLineToFlags( tableTest, symbol, flags, machineArray, instCounterPtr );
         ++( *instCounterPtr );

         machineArray[ *instCounterPtr ] = BRANCHZERO * 100;

         addLineToFlags( tableTest, symbol, flags, machineArray, instCounterPtr );
         ++( *instCounterPtr );
      }
   }

   // secondPass resolves incomplete SML instructions for forward references
   void secondPass( int flags[], int machineArray[], TableEntry symbolTable[] )
   {
      const char type = 'L';

      for ( int loop = 0; loop < MEMORYSIZE; ++loop ) {
         if ( flags[ loop ] != -1 ) {
            int symbol = flags[ loop ];
            int flagLocation = checkSymbolTable( symbolTable, symbol, type );
```

```
         machineArray[ loop ] += flagLocation;
      }
   }
}

// keyLet processes the keyword let
void keyLet( char *tokenPtr, int machineArray[], TableEntry symbolTable[],
             int *dataCounterPtr, int *instCounterPtr )
{
   const char type = 'V';
   char infixArray[ MAXIMUM ] = "", postfixArray[ MAXIMUM ] = "";
   int tableTest, symbol, location;
   static int subscript = 0;

   symbol = tokenPtr[ 0 ];
   tableTest = checkSymbolTable( symbolTable, symbol, type );

   if ( -1 == tableTest ) {
      addToSymbolTable( type, symbol, *dataCounterPtr, symbolTable,
                        *instCounterPtr );
      location = *dataCounterPtr;
      --( *dataCounterPtr );
   }
   else
      location = tableTest;

   tokenPtr = strtok( 0, " " );      // grab equal sign
   tokenPtr = strtok( 0, " " );      // get next token

   while ( tokenPtr != 0 ) {
      checkOperand( symbolTable, tokenPtr, dataCounterPtr,
                    instCounterPtr, machineArray );
      infixArray[ subscript ] = tokenPtr[ 0 ];
      ++subscript;
      tokenPtr = strtok( 0, " " );  // get next token
   }

   infixArray[ subscript ] = '\0';

   infixToPostfix( infixArray, postfixArray, location, symbolTable,
                   instCounterPtr, dataCounterPtr, machineArray );

   subscript = 0;    // reset static subscript when done
}

void addToSymbolTable( char type, int symbol, int dataCounter,
                       TableEntry symbolTable[], int instCounter )
{
   static int symbolCounter = 0;

   symbolTable[ symbolCounter ].type = type;
   symbolTable[ symbolCounter ].symbol = symbol;

   if ( type == 'V' || type == 'C' )
      symbolTable[ symbolCounter ].location = dataCounter;
   else
      symbolTable[ symbolCounter ].location = instCounter;


   ++symbolCounter;
}

void addLineToFlags( int tableTest, int symbol, int flags[],
```

```
                        int machineArray[], const int *instCounterPtr )
{
   if ( tableTest == -1 )
      flags[ *instCounterPtr ] = symbol;
   else
      machineArray[ *instCounterPtr ] += tableTest;
}

void printOutput( const int machineArray[], char *outFileName )
{
   ofstream output( outFileName, ios::out );

   if ( !output )
      cerr << "File was not opened.\n";
   else                   // output every memory cell
      for ( int loop = 0; loop <= MEMORYSIZE - 1; ++loop )
         output << machineArray[ loop ] << '\n';
}

void initArrays( int flags[], int machineArray[], TableEntry symbolTable[] )
{
   TableEntry initEntry = { 0, 0, -1 };

   for ( int loop = 0; loop < MEMORYSIZE; ++loop ) {
      flags[ loop ] = -1;
      machineArray[ loop ] = 0;
      symbolTable[ loop ] = initEntry;
   }
}

//////////////////////////////////////////////////////////////////////////////
// INFIX TO POSTFIX CONVERSION and POSTFIX EVALUATION FOR THE LET STATEMENT //
//////////////////////////////////////////////////////////////////////////////

// infixToPostfix converts an infix expression to a postfix expression
void infixToPostfix( char infix[], char postfix[], int getsVariable,
                     TableEntry symbolTable[], int *instCounterPtr,
                     int *dataCounterPtr, int machineArray[] )
{
   Stack< int > intStack;
   int infixCount, postfixCount, popValue;
   bool higher;
   int leftParen = '(';   // made int

   // push a left paren onto the stack and add a right paren to infix
   intStack.push( leftParen );
   strcat( infix, ")" );

   // convert the infix expression to postfix
   for ( infixCount = 0, postfixCount = 0; intStack.stackTop();
         ++infixCount ) {

      if ( isalnum( infix[ infixCount ] ) )
         postfix[ postfixCount++ ] = infix[ infixCount ];
      else if ( infix[ infixCount ] == '(' )
         intStack.push( leftParen );
      else if ( isOperator( infix[ infixCount ] ) ) {
         higher = true;   // used to store value of precedence test

         while ( higher ) {
            if ( isOperator( static_cast< char >( intStack.stackTop() ) ) )
               if ( precedence( static_cast< char >( intStack.stackTop() ),
```

```
                        infix[ infixCount ] ) )

                     postfix[ postfixCount++ ] =
                                       static_cast< char >( intStack.pop() );
                  else
                     higher = false;
               else
                  higher = false;
            }

            // See chapter 21 for a discussion of reinterpret_cast
            intStack.push( reinterpret_cast< int & > ( infix[ infixCount ] ) );
         }
         else if ( infix[ infixCount ] == ')' )
            while ( ( popValue = intStack.pop() ) != '(' )
               postfix[ postfixCount++ ] = static_cast< char >( popValue );
   }

   postfix[ postfixCount ] = '\0';

   evaluatePostfix( postfix, dataCounterPtr, instCounterPtr,
                    machineArray, getsVariable, symbolTable );
}

// check if c is an operator
bool isOperator( char c )
{
   if ( c == '+' || c == '-' || c == '*' || c == '/' || c == '^' )
      return true;
   else
      return false;
}

// If the precedence of operator1 is >= operator2,
bool precedence( char operator1, char operator2 )
{
   if ( operator1 == '^' )
      return true;
   else if ( operator2 == '^' )
      return false;
   else if ( operator1 == '*' || operator1 == '/' )
      return true;
   else if ( operator1 == '+' || operator1 == '-' )
      if ( operator2 == '*' || operator2 == '/' )
         return false;
      else
         return true;

   return false;
}

// evaluate postfix expression and produce code
void evaluatePostfix( char *expr, int *dataCounterPtr,
                      int *instCounterPtr, int machineArray[],
                      int getsVariable, TableEntry symbolTable[] )
{
   Stack< int > intStack;
   int popRightValue, popLeftValue, accumResult, symbolLocation, symbol;
   char type, array[ 2 ] = "";
   int i;

   strcat( expr, ")" );
```

```
   for ( i = 0; expr[ i ] != ')'; ++i )
      if ( isdigit( expr[ i ] ) ) {
         type = 'C';
         array[ 0 ] = expr[ i ];
         symbol = atoi( array );

         symbolLocation = checkSymbolTable( symbolTable, symbol, type );
         intStack.push( symbolLocation );
      }
      else if ( isalpha( expr[ i ] ) ) {
         type = 'V';
         symbol = expr[ i ];
         symbolLocation = checkSymbolTable( symbolTable, symbol, type );
         intStack.push( symbolLocation );
      }
      else {
         popRightValue = intStack.pop();
         popLeftValue = intStack.pop();
         accumResult = createLetSML( popRightValue, popLeftValue, machineArray,
                                     instCounterPtr, dataCounterPtr,
                                     expr[ i ] );
         intStack.push( accumResult );
      }

      machineArray[ *instCounterPtr ] = LOAD * 100;
      machineArray[ *instCounterPtr ] += intStack.pop();
      ++( *instCounterPtr );
      machineArray[ *instCounterPtr ] = STORE * 100;
      machineArray[ *instCounterPtr ] += getsVariable;
      ++( *instCounterPtr );
}

int createLetSML( int right, int left, int machineArray[],
                  int *instCounterPtr, int *dataCounterPtr, char oper )
{
   int location;

   switch( oper ) {
      case '+':
         machineArray[ *instCounterPtr ] = LOAD * 100;
         machineArray[ *instCounterPtr ] += left;
         ++( *instCounterPtr );
         machineArray[ *instCounterPtr ] = ADD * 100;
         machineArray[ *instCounterPtr ] += right;
         ++( *instCounterPtr );
         machineArray[ *instCounterPtr ] = STORE * 100;
         machineArray[ *instCounterPtr ] += *dataCounterPtr;
         location = *dataCounterPtr;
         --( *dataCounterPtr );
         ++( *instCounterPtr );
         return location;
      case '-':
         machineArray[ *instCounterPtr ] = LOAD * 100;
         machineArray[ *instCounterPtr ] += left;
         ++( *instCounterPtr );
         machineArray[ *instCounterPtr ] = SUBTRACT * 100;
         machineArray[ *instCounterPtr ] += right;
         ++( *instCounterPtr );
         machineArray[ *instCounterPtr ] = STORE * 100;
         machineArray[ *instCounterPtr ] += *dataCounterPtr;
         location = *dataCounterPtr;
```

```
            --( *dataCounterPtr );
            ++( *instCounterPtr );
            return location;
         case '/':
            machineArray[ *instCounterPtr ] = LOAD * 100;
            machineArray[ *instCounterPtr ] += left;
            ++( *instCounterPtr );
            machineArray[ *instCounterPtr ] = DIVIDE * 100;
            machineArray[ *instCounterPtr ] += right;
            ++( *instCounterPtr );
            machineArray[ *instCounterPtr ] = STORE * 100;
            machineArray[ *instCounterPtr ] += *dataCounterPtr;
            location = *dataCounterPtr;
            --( *dataCounterPtr );
            ++( *instCounterPtr );
            return location;
         case '*':
            machineArray[ *instCounterPtr ] = LOAD * 100;
            machineArray[ *instCounterPtr ] += left;
            ++( *instCounterPtr );
            machineArray[ *instCounterPtr ] = MULTIPLY * 100;
            machineArray[ *instCounterPtr ] += right;
            ++( *instCounterPtr );
            machineArray[ *instCounterPtr ] = STORE * 100;
            machineArray[ *instCounterPtr ] += *dataCounterPtr;
            location = *dataCounterPtr;
            --( *dataCounterPtr );
            ++( *instCounterPtr );
            return location;
         default:
            cerr << "ERROR: operator not recognized.\n";
            break;
      }

      return 0;    // default return
   }
```

15.28 (*Optimizing the Simple Compiler*) When a program is compiled and converted into SML, a set of instructions is generated. Certain combinations of instructions often repeat themselves, usually in triplets called *productions.* A production normally consists of three instructions such as *load*, *add* and *store*. For example, Fig. 15.27 illustrates five of the SML instructions that were produced in the compilation of the program in Fig. 15.25 The first three instructions are the production that adds **1** to **y**. Note that instructions **06** and **07** store the accumulator value in temporary location **96** and then load the value back into the accumulator so instruction **08** can store the value in location **98**. Often a production is followed by a load instruction for the same location that was just stored. This code can be *optimized* by eliminating the store instruction and the subsequent load instruction that operate on the same memory location, thus enabling the Simpletron to execute the program faster. Figure 15.28 illustrates the optimized SML for the program of Fig. 15.25. Note that there are four fewer instructions in the optimized code—a memory-space savings of 25%.

Modify the compiler to provide an option for optimizing the Simpletron Machine Language code it produces. Manually compare the nonoptimized code with the optimized code, and calculate the percentage reduction.

15.29 (*Modifications to the Simple compiler*) Perform the following modifications to the Simple compiler. Some of these modifications may also require modifications to the Simpletron Simulator program written in Exercise 5.19.

a) Allow the modulus operator (**%**) to be used in **let** statements. Simpletron Machine Language must be modified to include a modulus instruction.
b) Allow exponentiation in a **let** statement using **^** as the exponentiation operator. Simpletron Machine Language must be modified to include an exponentiation instruction.

```
04    +2098    (load)
05    +3097    (add)
```

Fig. 15.27 Nonoptimized code from the program of Fig. 15.25.

```
06   +2196   (store)
07   +2096   (load)
08   +2198   (store)
```

Fig. 15.27 Nonoptimized code from the program of Fig. 15.25.

Simple program	SML location and instruction	Description
`5 rem sum 1 to x`	*none*	**`rem`** ignored
`10 input x`	`00  +1099`	read **`x`** into location **`99`**
`15 rem   check y == x`	*none*	**`rem`** ignored
`20 if y == x goto 60`	`01  +2098`	load **`y`** (**`98`**) into accumulator
	`02  +3199`	sub **`x`** (**`99`**) from accumulator
	`03  +4211`	branch to location **`11`** if zero
`25 rem   increment y`	*none*	**`rem`** ignored
`30 let y = y + 1`	`04  +2098`	load **`y`** into accumulator
	`05  +3097`	add **`1`** (**`97`**) to accumulator
	`06  +2198`	store accumulator in **`y`** (**`98`**)
`35 rem   add y to total`	*none*	**`rem`** ignored
`40 let t = t + y`	`07  +2096`	load **`t`** from location (**`96`**)
	`08  +3098`	add **`y`** (**`98`**) accumulator
	`09  +2196`	store accumulator in **`t`** (**`96`**)
`45 rem   loop y`	*none*	**`rem`** ignored
`50 goto 20`	`10  +4001`	branch to location **`01`**
`55 rem   output result`	*none*	**`rem`** ignored
`60 print t`	`11  +1196`	output **`t`** (**`96`**) to screen
`99 end`	`12  +4300`	terminate execution

Fig. 15.28 Optimized code for the program of Fig. 15.25.

c) Allow the compiler to recognize uppercase and lowercase letters in Simple statements (e.g., **`'A'`** is equivalent to **`'a'`**). No modifications to the Simulator are required.

d) Allow **`input`** statements to read values for multiple variables such as **`input x, y`**. No modifications to the Simpletron Simulator are required.

e) Allow the compiler to output multiple values in a single **`print`** statement such as **`print a, b, c`**. No modifications to the Simpletron Simulator are required.

f) Add syntax-checking capabilities to the compiler so error messages are output when syntax errors are encountered in a Simple program. No modifications to the Simpletron Simulator are required.

g) Allow arrays of integers. No modifications to the Simpletron Simulator are required.

h) Allow subroutines specified by the Simple commands **`gosub`** and **`return`**. Command **`gosub`** passes program control to a subroutine, and command **`return`** passes control back to the statement after the **`gosub`**. This is similar to a function call in C++. The same subroutine can be called from many **`gosub`** commands distributed throughout a program. No modifications to the Simpletron Simulator are required.

i) Allow repetition structures of the form

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

This **`for`** statement loops from **`2`** to **`10`** with an increment of **`2`**. The **`next`** line marks the end of the body of the **`for`** line. No modifications to the Simpletron Simulator are required.

j) Allow repetition structures of the form

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

This **for** statement loops from **2** to **10** with a default increment of **1**. No modifications to the Simpletron Simulator are required.

k) Allow the compiler to process string input and output. This requires the Simpletron Simulator to be modified to process and store string values. (Hint: Each Simpletron word can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the ASCII decimal equivalent of a character. Add a machine language instruction that will print a string beginning at a certain Simpletron memory location. The first half of the word at that location is a count of the number of characters in the string (i.e., the length of the string). Each succeeding half word contains one ASCII character expressed as two decimal digits. The machine language instruction checks the length and prints the string by translating each two-digit number into its equivalent character.)

l) Allow the compiler to process floating-point values in addition to integers. The Simpletron Simulator must also be modified to process floating-point values.

15.30 (*A Simple interpreter*) An interpreter is a program that reads a high-level language program statement, determines the operation to be performed by the statement and executes the operation immediately. The high-level language program is not converted into machine language first. Interpreters execute slowly because each statement encountered in the program must first be deciphered. If statements are contained in a loop, the statements are deciphered each time they are encountered in the loop. Early versions of the BASIC programming language were implemented as interpreters.

Write an interpreter for the Simple language discussed in Exercise 15.26. The program should use the infix-to-postfix converter developed in Exercise 15.12 and the postfix evaluator developed in Exercise 15.13 to evaluate expressions in a **let** statement. The same restrictions placed on the Simple language in Exercise 15.26 should be adhered to in this program. Test the interpreter with the Simple programs written in Exercise 15.26. Compare the results of running these programs in the interpreter with the results of compiling the Simple programs and running them in the Simpletron Simulator built in Exercise 5.19.

15.31 *(Insert/Delete Anywhere in a Linked List)* Our linked list class template allowed insertions and deletions at only the front and the back of the linked list. These capabilities were convenient for us when we used private inheritance and composition to produce a stack class template and a queue class template with a minimal amount of code by reusing the list class template. Actually, linked lists are more general that those we provided. Modify the linked list class template we developed in this chapter to handle insertions and deletions anywhere in the list.

```cpp
// LIST.H
// Template List class definition
// Added copy constructor to member functions (not included in chapter).
#ifndef LIST_H
#define LIST_H

#include <iostream>

using std::cout;

#include <cassert>
#include "listnd.h"

template< class NODETYPE >
class List {
public:
   List();                              // default constructor
   List( const List< NODETYPE > & );   // copy constructor
   ~List();                             // destructor
   void insertAtFront( const NODETYPE & );
   void insertAtBack( const NODETYPE & );
   bool removeFromFront( NODETYPE & );
   bool removeFromBack( NODETYPE & );

   void insertInOrder( const NODETYPE & );
   bool isEmpty() const;
   void print() const;
```

```
protected:
   ListNode< NODETYPE > *firstPtr;  // pointer to first node
   ListNode< NODETYPE > *lastPtr;   // pointer to last node

   // Utility function to allocate a new node
   ListNode< NODETYPE > *getNewNode( const NODETYPE & );
};

// Default constructor
template< class NODETYPE >
List< NODETYPE >::List() { firstPtr = lastPtr = 0; }

// Copy constructor
template< class NODETYPE >
List< NODETYPE >::List( const List<NODETYPE> &copy )
{
   firstPtr = lastPtr = 0;  // initialize pointers

   ListNode< NODETYPE > *currentPtr = copy.firstPtr;

   while ( currentPtr != 0 ) {
      insertAtBack( currentPtr -> data );
      currentPtr = currentPtr -> nextPtr;
   }
}

// Destructor
template< class NODETYPE >
List< NODETYPE >::~List()
{
   if ( !isEmpty() ) {    // List is not empty
      cout << "Destroying nodes ...\n";

      ListNode< NODETYPE > *currentPtr = firstPtr, *tempPtr;

      while ( currentPtr != 0 ) {  // delete remaining nodes
         tempPtr = currentPtr;
         cout << tempPtr -> data << ' ';
         currentPtr = currentPtr -> nextPtr;
         delete tempPtr;
      }
   }

   cout << "\nAll nodes destroyed\n\n";
}

// Insert a node at the front of the list
template< class NODETYPE >
void List< NODETYPE >::insertAtFront( const NODETYPE &value )
{
   ListNode<NODETYPE> *newPtr = getNewNode( value );

   if ( isEmpty() )  // List is empty
      firstPtr = lastPtr = newPtr;
   else {           // List is not empty
      newPtr -> nextPtr = firstPtr;
      firstPtr = newPtr;
   }
}

// Insert a node at the back of the list
template< class NODETYPE >
```

```
void List< NODETYPE >::insertAtBack( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value );

   if ( isEmpty() )  // List is empty
      firstPtr = lastPtr = newPtr;
   else {           // List is not empty
      lastPtr -> nextPtr = newPtr;
      lastPtr = newPtr;
   }
}

// Delete a node from the front of the list
template< class NODETYPE >
bool List< NODETYPE >::removeFromFront( NODETYPE &value )
{
   if ( isEmpty() )             // List is empty
      return false;             // delete unsuccessful
   else {
      ListNode< NODETYPE > *tempPtr = firstPtr;

      if ( firstPtr == lastPtr )
         firstPtr = lastPtr = 0;
      else
         firstPtr = firstPtr -> nextPtr;

      value = tempPtr -> data;  // data being removed
      delete tempPtr;
      return true;              // delete successful
   }
}

// Delete a node from the back of the list
template< class NODETYPE >
bool List< NODETYPE >::removeFromBack( NODETYPE &value )
{
   if ( isEmpty() )
      return false;   // delete unsuccessful
   else {
      ListNode< NODETYPE > *tempPtr = lastPtr;

      if ( firstPtr == lastPtr )
         firstPtr = lastPtr = 0;
      else {
         ListNode< NODETYPE > *currentPtr = firstPtr;

         while ( currentPtr -> nextPtr != lastPtr )
            currentPtr = currentPtr -> nextPtr;

         lastPtr = currentPtr;
         currentPtr -> nextPtr = 0;
      }

      value = tempPtr -> data;
      delete tempPtr;
      return true;   // delete successful
   }
}


// Is the List empty?
template< class NODETYPE >
bool List< NODETYPE >::isEmpty() const { return firstPtr == 0; }
```

```
// Return a pointer to a newly allocated node
template< class NODETYPE >
ListNode< NODETYPE > *List< NODETYPE >::getNewNode( const NODETYPE &value )
{
   ListNode< NODETYPE > *ptr = new ListNode< NODETYPE >( value );
   assert( ptr != 0 );
   return ptr;
}

// Display the contents of the List
template< class NODETYPE >
void List< NODETYPE >::print() const
{
   if ( isEmpty() ) {
      cout << "The list is empty\n\n";
      return;
   }

   ListNode< NODETYPE > *currentPtr = firstPtr;

   cout << "The list is: ";

   while ( currentPtr != 0 ) {
      cout << currentPtr -> data << ' ';
      currentPtr = currentPtr -> nextPtr;
   }

   cout << "\n\n";
}

template< class NODETYPE >
void List< NODETYPE >::insertInOrder( const NODETYPE &value )
{
   if ( isEmpty() ) {
      ListNode< NODETYPE > *newPtr = getNewNode( value );
      firstPtr = lastPtr = newPtr;
   }
   else {
      if ( firstPtr -> data > value )
         insertAtFront( value );
      else if ( lastPtr -> data < value )
         insertAtBack( value );
      else {
         ListNode< NODETYPE > *currentPtr = firstPtr -> nextPtr,
                              *previousPtr = firstPtr,
                              *newPtr = getNewNode( value );

         while ( currentPtr != lastPtr && currentPtr -> data < value ) {
            previousPtr = currentPtr;
            currentPtr = currentPtr -> nextPtr;
         }

         previousPtr -> nextPtr = newPtr;
         newPtr -> nextPtr = currentPtr;
      }
   }
}

#endif
```

```
// LISTND.H
// ListNode template definition
#ifndef LISTND_H
#define LISTND_H

template< class T > class List;  // forward declaration

template< class NODETYPE >
class ListNode {
   friend class List< NODETYPE >; // make List a friend
public:
   ListNode( const NODETYPE & );  // constructor
   NODETYPE getData() const;     // return the data in the node
   void setNextPtr( ListNode *nPtr ) { nextPtr = nPtr; }
   ListNode *getNextPtr() const { return nextPtr; }
private:
   NODETYPE data;                // data
   ListNode *nextPtr;            // next node in the list
};

// Constructor
template< class NODETYPE >
ListNode< NODETYPE >::ListNode( const NODETYPE &info )
{
   data = info;
   nextPtr = 0;
}

// Return a copy of the data in the node
template< class NODETYPE >
NODETYPE ListNode< NODETYPE >::getData() const { return data; }

#endif
```

```
// LIST2.H
// Template List class definition
// NOTE: This solution only provides the delete anywhere operation.

#ifndef LIST2_H
#define LIST2_H

#include <cassert>
#include "listnd.h"
#include "list.h"

template< class NODETYPE >
class List2 : public List< NODETYPE > {
public:
   bool deleteNode( const NODETYPE &, NODETYPE & );
};

// Delete a node from anywhere in the list
template< class NODETYPE >
bool List2< NODETYPE >::deleteNode( const NODETYPE &val, NODETYPE &deletedVal )
{
   if ( isEmpty() )
      return false;   // delete unsuccessful

   else {
      if ( firstPtr -> getData() == val ) {
```

```
         removeFromFront( deletedVal );
         return true;  // delete successful
      }
      else if ( lastPtr -> getData() == val ) {
         removeFromBack( deletedVal );
         return true;  // delete successful
      }
      else {
         ListNode< NODETYPE > *currentPtr = firstPtr -> getNextPtr(),
                              *previousPtr = firstPtr;

         while ( currentPtr != lastPtr && currentPtr -> getData() < val ) {
            previousPtr = currentPtr;
            currentPtr = currentPtr -> getNextPtr();
         }

         if ( currentPtr -> getData() == val ) {
            ListNode< NODETYPE > *tempPtr = currentPtr;
            deletedVal = currentPtr -> getData();
            previousPtr -> setNextPtr( currentPtr -> getNextPtr() );
            delete tempPtr;
            return true;  // delete successful
         }
         else
            return false;  // delete unsuccessful
      }
   }
}

#endif
```

```
// Exercise 15.31 solution
#include <iostream>

using std::cout;
using std::cin;

#include <cstdlib>
#include <ctime>
#include "list2.h"

int main()
{
   srand( time( 0 ) );  // randomize the random number generator

   List2< int > intList;

   for ( int i = 1; i <= 10; ++i )
      intList.insertInOrder( rand() % 101 );

   intList.print();

   int value, deletedValue;

   cout << "Enter an integer to delete (-1 to end): ";
   cin >> value;

   while ( value != -1 ) {
      if ( intList.deleteNode( value, deletedValue ) ) {
         cout << deletedValue << " was deleted from the list\n";
         intList.print();
```

```
      }
      else
         cout << "Element was not found";

      cout << "Enter an integer to delete (-1 to end): ";
      cin >> value;
   }

   return 0;
}
```

```
The list is: 0 3 20 35 39 51 61 62 64 82

Enter an integer to delete (-1 to end): 8
Element was not foundEnter an integer to delete (-1 to end): 82
82 was deleted from the list
The list is: 0 3 20 35 39 51 61 62 64

Enter an integer to delete (-1 to end): -1
Destroying nodes ...
0 3 20 35 39 51 61 62 64
All nodes destroyed
```