Special Section: Building Your Own Compiler

In Exercise 7.27 and Exercise 7.28, we introduced Simpletron Machine Language (SML) and created the 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. We'll write programs in this new high-level language, compile the programs on the compiler we build, and run the programs on the simulator we built in Exercise 7.28.

12.26 ***(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` or `end` (see Fig. 12.23). 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.

Command	Example statement	Description
`rem`	`50 rem this is a remark`	Any text following the command `rem` is for documentation purposes only 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)`. An arbitrarily complex expression can appear to the right of the equal 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 program control to line `80` if the condition is true; otherwise, continue execution with the next statement.
`end`	`99 end`	Terminate program execution.

Fig. 12.23 | Simple commands.

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 the 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. Our compiler will assume that Simple programs

are entered correctly. Exercise 12.29 asks the student to modify the compiler to perform syntax error checking.

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: `<`, `>`, `<=`, `>=`, `==` or `!=`. The precedence of these operators is the same as in C.

Let's now consider several Simple programs that demonstrate Simple's features. The first program (Fig. 12.24) reads two integers from the keyboard, stores the values in variables `a` and `b`, and computes and prints their sum (stored in variable `c`).

```
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
```

Fig. 12.24 | Determine the sum of two integers.

Figure 12.25 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.

```
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. 12.25 | Find the larger of two integers.

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 12.26 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   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. 12.26 | Calculate the squares of several integers.

Using the sample programs of Figs. 12.24–12.26 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 seven 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.

12.27 ***(Building A Compiler; Prerequisite: Complete Exercise 7.27, Exercise 7.28, Exercise 12.12, Exercise 12.13 and Exercise 12.26)*** Now that the Simple language has been presented (Exercise 12.26), we discuss how to build our Simple compiler. First, we consider the process by which a Simple program is converted to SML and executed by the Simpletron simulator (see Fig. 12.27). 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. The Simpletron program developed in Exercise 7.28 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.

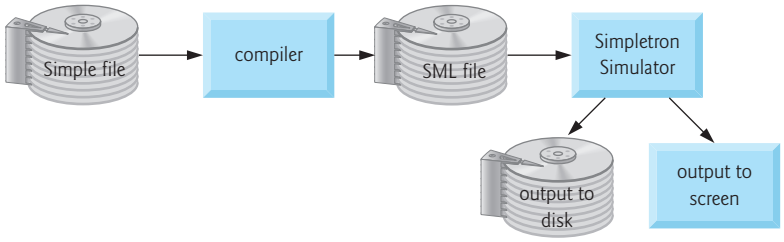

Fig. 12.27 | Writing, compiling and executing a Simple language program.

The compiler performs two passes of the Simple program to convert it to SML. The first pass constructs a symbol table in which every line number, variable name and constant 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(s) for each Simple statement. As we'll 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 incomplete 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's placed in the symbol table. A line number is placed in the symbol table only if it's the first token in a statement. The `symbolTable` is an array of `tableEntry` structures 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.

The `tableEntry` structure definition is as follows:

```
struct tableEntry {
   int symbol;
   char type; /* 'C', 'L' or 'V' */
   int location; /* 00 to 99 */
};
```

Each `tableEntry` structure 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, or `'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 7 that an SML instruction is a four-digit integer that comprises 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:*

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's 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 will be 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` statement, 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 possibly 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. Exercise 12.12 and Exercise 12.13 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, 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" that allows our compiler to 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 on 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:

1) Search the `flags` array for an unresolved reference (i.e., an element with a value other than `-1`).
2) Locate the structure in array `symbolTable` containing the symbol stored in the `flags` array (be sure that the type of the symbol is `'L'` for line number).
3) Insert the memory location from structure member `location` into the instruction with the unresolved reference (remember that an instruction containing an unresolved reference has operand `00`).
4) Repeat *Steps 1–3* 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).

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 are illustrated in Fig. 12.28. The symbol table constructed by the first pass is shown in Fig. 12.29.

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 +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. 12.28 | 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
45	L	14
50	L	14
55	L	15
60	L	15
99	L	16

Fig. 12.29 | Symbol table for program of Fig. 12.28.

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 (line numbers are placed in the symbol table only when they appear as the first token in a statement). Therefore, it's 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. The size of Simpletron's memory could present a problem for Simple programs with many statements, variables and constants. It's 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.

Step-by-Step View of the Compilation Process

Let's now walk through the compilation process for the Simple program in Fig. 12.28. 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 Chapter 8 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's inserted in the symbol table. Since we're 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 allocated for the line number (in case it's 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 simply has to determine the location of `x` in the SML array. Symbol `x` is not found in the symbol table. So, it's 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's 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 and pops the stack again into the left operand of the operator, 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'll 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. The result of t + y is assigned to temporary location 94 before being assigned to t (95). The instructions in memory locations 11 and 12 appear to be redundant. Again, we'll 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'll have to perform each of the following tasks:

a) Modify the Simpletron simulator program you wrote in Exercise 7.28 to take its input from a file specified by the user (see Chapter 11). Also, the simulator should output its results to a disk file in the same format as the screen output.

b) Modify the infix-to-postfix evaluation algorithm of Exercise 12.12 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 multi-digit integer operands and variable name operands. Also, the algorithm should now implement the previously discussed "hook" 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.

12.28 ***(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. 12.30 illustrates five of the SML instructions that were produced in the compilation of the program in Fig. 12.28. The first three instructions are the production that

adds 1 to y. Instructions 06 and 07 store the accumulator value in temporary location 96, 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. This optimization would enable the Simpletron to execute the program faster because there are fewer instructions in this version. Figure 12.31 illustrates the optimized SML for the program of Fig. 12.28. There are four fewer instructions in the optimized code—a memory-space savings of 25%.

```
04 +2098          (load)
05 +3097          (add)
06 +2196          (store)
07 +2096          (load)
08 +2198          (store)
```

Fig. 12.30 | Unoptimized code from the program of Fig. 12.28.

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. 12.31 | Optimized code for the program of Fig. 12.28.

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

12.29 ***(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 7.28.

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.
c) Allow the compiler to recognize uppercase and lowercase letters in Simple statements (e.g., `'A'` is equivalent to `'a'`). No modifications to the Simpletron 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`s 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
    rem Simple statements
next
```

j) 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.
k) Allow repetition structures of the form

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

l) This `for` statement loops from `2` to `10` with a default increment of `1`. No modifications to the Simpletron Simulator are required.
m) 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.
n) Allow the compiler to process floating-point values in addition to integers. The Simpletron Simulator must also be modified to process floating-point values.

12.30 ***(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 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 12.26. The program should use the infix-to-postfix converter developed in Exercise 12.12 and the postfix evaluator developed in Exercise 12.13 to evaluate expressions in a `let` statement. The same restrictions placed on the Simple language in Exercise 12.26 should be adhered to in this program. Test the interpreter with the Simple programs written in Exercise 12.26. 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 7.28.