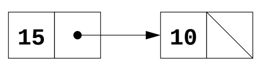

Fig. 22.1 Two self-referential class objects linked together.

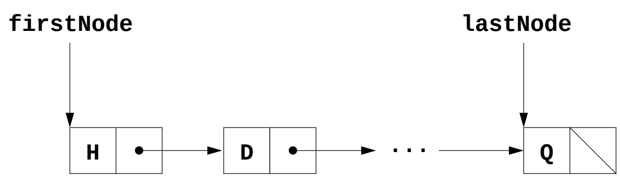

Fig. 22.2 A graphical representation of a linked list.

```java
// Fig. 22.3: List.java
// Class ListNode and class List definitions
package com.deitel.jhtp3.ch22;

class ListNode {
   // package access data so class List can access it directly
   Object data;
   ListNode next;

   // Constructor: Create a ListNode that refers to Object o.
   ListNode( Object o ) { this( o, null ); }

   // Constructor: Create a ListNode that refers to Object o and
   // to the next ListNode in the List.
   ListNode( Object o, ListNode nextNode )
   {
      data = o;          // this node refers to Object o
      next = nextNode;   // set next to refer to next
   }

   // Return a reference to the Object in this node
   Object getObject() { return data; }

   // Return the next node
   ListNode getNext() { return next; }
}

// Class List definition
public class List {
   private ListNode firstNode;
   private ListNode lastNode;
   private String name;  // String like "list" used in printing

   // Constructor: Construct an empty List with s as the name
   public List( String s )
   {
      name = s;
      firstNode = lastNode = null;
   }

   // Constructor: Construct an empty List with
   // "list" as the name
   public List() { this( "list" ); }

   // Insert an Object at the front of the List
   // If List is empty, firstNode and lastNode will refer to
   // the same object. Otherwise, firstNode refers to new node.
   public synchronized void insertAtFront( Object insertItem )
   {
      if ( isEmpty() )
         firstNode = lastNode = new ListNode( insertItem );
```

Fig. 22.3 Manipulating a linked list (part 1 of 5).

```
      else
         firstNode = new ListNode( insertItem, firstNode );
   }

   // Insert an Object at the end of the List
   // If List is empty, firstNode and lastNode will refer to
   // the same Object. Otherwise, lastNode's next instance
   // variable refers to new node.
   public synchronized void insertAtBack( Object insertItem )
   {
      if ( isEmpty() )
         firstNode = lastNode = new ListNode( insertItem );
      else
         lastNode = lastNode.next = new ListNode( insertItem );
   }

   // Remove the first node from the List.
   public synchronized Object removeFromFront()
          throws EmptyListException
   {
      Object removeItem = null;

      if ( isEmpty() )
         throw new EmptyListException( name );

      removeItem = firstNode.data;  // retrieve the data

      // reset the firstNode and lastNode references
      if ( firstNode.equals( lastNode ) )
         firstNode = lastNode = null;
      else
         firstNode = firstNode.next;

      return removeItem;
   }

   // Remove the last node from the List.
   public synchronized Object removeFromBack()
          throws EmptyListException
   {
      Object removeItem = null;

      if ( isEmpty() )
         throw new EmptyListException( name );

      removeItem = lastNode.data;  // retrieve the data

      // reset the firstNode and lastNode references
      if ( firstNode.equals( lastNode ) )
         firstNode = lastNode = null;
      else {
         ListNode current = firstNode;

```

Fig. 22.3 Manipulating a linked list (part 2 of 5).

```
         while ( current.next != lastNode )  // not last node
            current = current.next;      // move to next node

         lastNode = current;
         current.next = null;
      }

      return removeItem;
   }

   // Return true if the List is empty
   public synchronized boolean isEmpty()
      { return firstNode == null; }

   // Output the List contents
   public synchronized void print()
   {
      if ( isEmpty() ) {
         System.out.println( "Empty " + name );
         return;
      }

      System.out.print( "The " + name + " is: " );

      ListNode current = firstNode;

      while ( current != null ) {
         System.out.print( current.data.toString() + " " );
         current = current.next;
      }

      System.out.println( "\n" );
   }
}
```

Fig. 22.3 Manipulating a linked list (part 3 of 5).

```
// Fig. 22.3: EmptyListException.java
// Class EmptyListException definition
package com.deitel.jhtp3.ch22;

public class EmptyListException extends RuntimeException {
   public EmptyListException( String name )
   {
      super( "The " + name + " is empty" );
   }
}
```

Fig. 22.3 Manipulating a linked list (part 4 of 5).

```java
// Fig. 22.3: ListTest.java
// Class ListTest
import com.deitel.jhtp3.ch22.List;
import com.deitel.jhtp3.ch22.EmptyListException;

public class ListTest {
   public static void main( String args[] )
   {
      List objList = new List();  // create the List container

      // Create objects to store in the List
      Boolean b = Boolean.TRUE;
      Character c = new Character( '$' );
      Integer i = new Integer( 34567 );
      String s = "hello";

      // Use the List insert methods
      objList.insertAtFront( b );
      objList.print();
      objList.insertAtFront( c );
      objList.print();
      objList.insertAtBack( i );
      objList.print();
      objList.insertAtBack( s );
      objList.print();

      // Use the List remove methods
      Object removedObj;

      try {
         removedObj = objList.removeFromFront();
         System.out.println(
            removedObj.toString() + " removed" );
         objList.print();
         removedObj = objList.removeFromFront();
         System.out.println(
            removedObj.toString() + " removed" );
         objList.print();
         removedObj = objList.removeFromBack();
         System.out.println(
            removedObj.toString() + " removed" );
         objList.print();
         removedObj = objList.removeFromBack();
         System.out.println(
            removedObj.toString() + " removed" );
         objList.print();
      }
      catch ( EmptyListException e ) {
         System.err.println( "\n" + e.toString() );
      }
   }
}
```

Fig. 22.3 Manipulating a linked list (part 5 of 5).

```
The list is: true

The list is: $ true

The list is: $ true 34567

The list is: $ true 34567 hello

$ removed
The list is: true 34567 hello

true removed
The list is: 34567 hello

hello removed
The list is: 34567

34567 removed
Empty list
```

Fig. 22.4 Sample output for the program of Fig. 22.3.

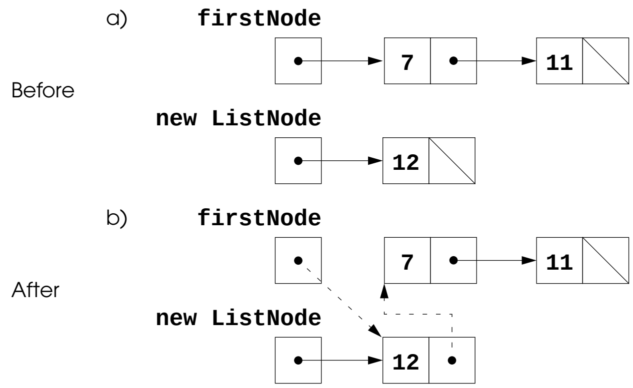

Fig. 22.5 The **`insertAtFront`** operation.

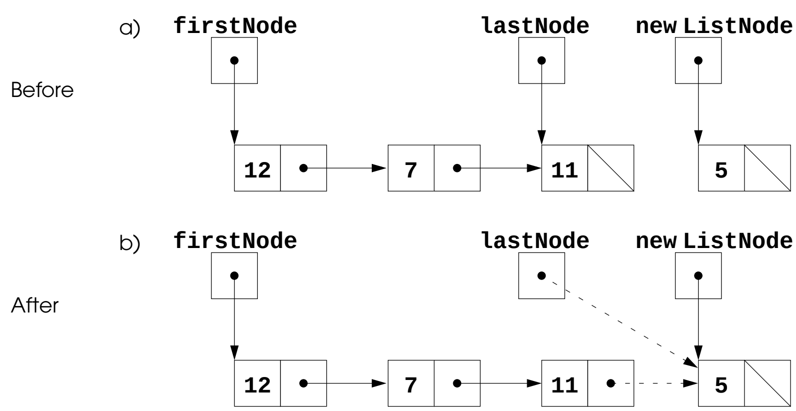

Fig. 22.6 A graphical representation of the **`insertAtBack`** operation.

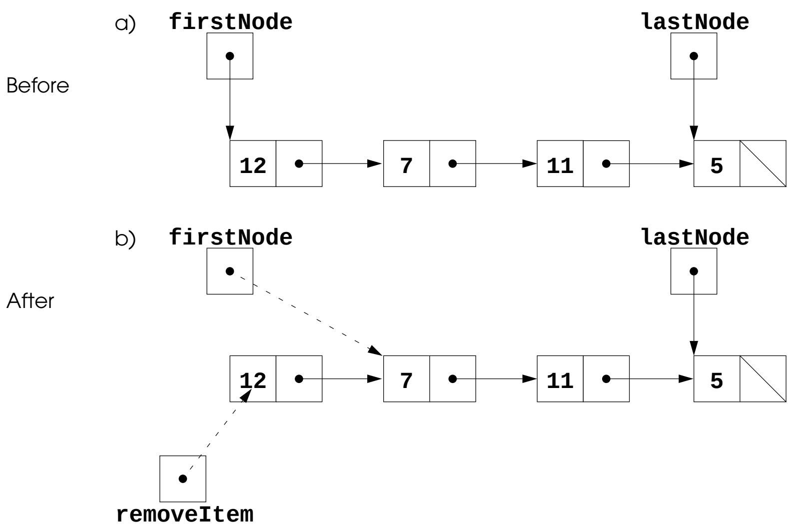

Fig. 22.7 A graphical representation of the `removeFromFront` operation.

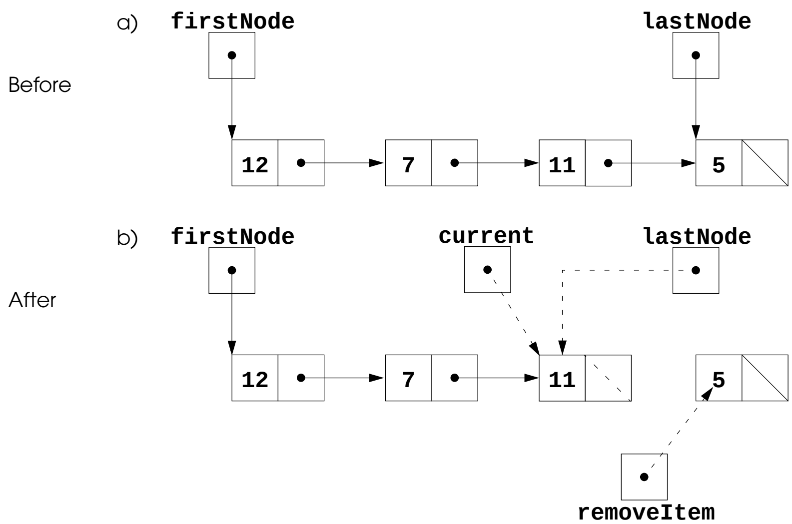

Fig. 22.8 A graphical representation of the `removeFromBack` operation.

```java
// Fig. 22.9: StackInheritance.java
// Derived from class List
package com.deitel.jhtp3.ch22;

public class StackInheritance extends List {
   public StackInheritance() { super( "stack" ); }
   public void push( Object o )
      { insertAtFront( o ); }
   public Object pop() throws EmptyListException
      { return removeFromFront(); }
   public boolean isEmpty() { return super.isEmpty(); }
   public void print() { super.print(); }
}
```

Fig. 22.9 A simple stack program (part 1 of 3).

```java
// Fig. 22.9: StackInheritanceTest.java
// Class StackInheritanceTest
import com.deitel.jhtp3.ch22.StackInheritance;
import com.deitel.jhtp3.ch22.EmptyListException;

public class StackInheritanceTest {
   public static void main( String args[] )
   {
      StackInheritance objStack = new StackInheritance();

      // Create objects to store in the stack
      Boolean b = Boolean.TRUE;
      Character c = new Character( '$' );
```

Fig. 22.9 A simple stack program (part 2 of 3).

```
      Integer i = new Integer( 34567 );
      String s = "hello";

      // Use the push method
      objStack.push( b );
      objStack.print();
      objStack.push( c );
      objStack.print();
      objStack.push( i );
      objStack.print();
      objStack.push( s );
      objStack.print();

      // Use the pop method
      Object removedObj = null;

      try {
         while ( true ) {
            removedObj = objStack.pop();
            System.out.println( removedObj.toString() +
                                " popped" );
            objStack.print();
         }
      }
      catch ( EmptyListException e ) {
         System.err.println( "\n" + e.toString() );
      }
   }
}
```

Fig. 22.9 A simple stack program (part 3 of 3).

```
The stack is: true

The stack is: $ true

The stack is: 34567 $ true

The stack is: hello 34567 $ true

hello popped
The stack is: 34567 $ true

34567 popped
The stack is: $ true

$ popped
The stack is: true

true popped
Empty stack

com.deitel.jhtp3.ch22.EmptyListException:
   The stack is empty
```

Fig. 22.10 Sample output from the program of Fig. 22.9.

```
// Fig. 22.11: StackComposition.java
// Class StackComposition definition with composed List object
package com.deitel.jhtp3.ch22;

public class StackComposition {
   private List s;

   public StackComposition() { s = new List( "stack" ); }
   public void push( Object o )
      { s.insertAtFront( o ); }
   public Object pop() throws EmptyListException
      { return s.removeFromFront(); }
   public boolean isEmpty() { return s.isEmpty(); }
   public void print() { s.print(); }
}
```

Fig. 22.11 A simple stack class using composition.

```java
// Fig. 22.12: QueueInheritance.java
// Class QueueInheritance definition
// Derived from List
package com.deitel.jhtp3.ch22;

public class QueueInheritance extends List {
   public QueueInheritance() { super( "queue" ); }
   public void enqueue( Object o )
      { insertAtBack( o ); }
   public Object dequeue()
      throws EmptyListException { return removeFromFront(); }
   public boolean isEmpty() { return super.isEmpty(); }
   public void print() { super.print(); }
}
```

Fig. 22.12 Processing a queue (part 1 of 2).

```java
// Fig. 22.12: QueueInheritanceTest.java
// Class QueueInheritanceTest
import com.deitel.jhtp3.ch22.QueueInheritance;
import com.deitel.jhtp3.ch22.EmptyListException;

public class QueueInheritanceTest {
   public static void main( String args[] )
   {
      QueueInheritance objQueue = new QueueInheritance();

      // Create objects to store in the queue
      Boolean b = Boolean.TRUE;
      Character c = new Character( '$' );
      Integer i = new Integer( 34567 );
      String s = "hello";

      // Use the enqueue method
      objQueue.enqueue( b );
      objQueue.print();
      objQueue.enqueue( c );
      objQueue.print();
      objQueue.enqueue( i );
      objQueue.print();
      objQueue.enqueue( s );
      objQueue.print();

      // Use the dequeue method
      Object removedObj = null;

      try {
         while ( true ) {
            removedObj = objQueue.dequeue();
            System.out.println( removedObj.toString() +
                                " dequeued" );
            objQueue.print();
         }
      }
      catch ( EmptyListException e ) {
         System.err.println( "\n" + e.toString() );
      }
   }
}
```

Fig. 22.12 Processing a queue (part 2 of 2).

```
The queue is: true

The queue is: true $

The queue is: true $ 34567

The queue is: true $ 34567 hello

true dequeued
The queue is: $ 34567 hello

$ dequeued
The queue is: 34567 hello

34567 dequeued
The queue is: hello

hello dequeued
Empty queue

com.deitel.jhtp3.ch22.EmptyListException:
   The queue is empty
```

Fig. 22.13 Sample output from the program in Fig. 22.12.

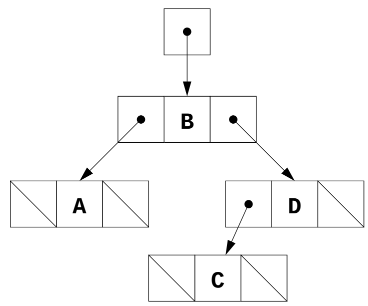

Fig. 22.14 A graphical representation of a binary tree.

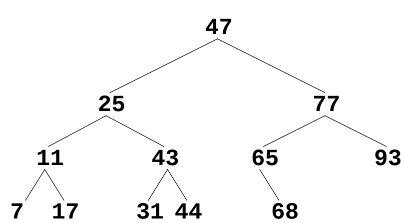

Fig. 22.15 A binary search tree.

```java
// Fig. 22.16: Tree.java
package com.deitel.jhtp3.ch22;

// Class TreeNode definition
class TreeNode {
   // package access members
   TreeNode left;   // left node
   int data;        // data item
   TreeNode right;  // right node

   // Constructor: initialize data to d and make this a leaf node
   public TreeNode( int d )
   {
      data = d;
      left = right = null;  // this node has no children
   }

   // Insert a TreeNode into a Tree that contains nodes.
   // Ignore duplicate values.
   public synchronized void insert( int d )
   {
      if ( d < data ) {
         if ( left == null )
            left = new TreeNode( d );
         else
            left.insert( d );
      }
      else if ( d > data ) {
         if ( right == null )
            right = new TreeNode( d );
         else
            right.insert( d );
      }
   }
}

// Class Tree definition
public class Tree {
   private TreeNode root;

   // Construct an empty Tree of integers
   public Tree() { root = null; }

   // Insert a new node in the binary search tree.
   // If the root node is null, create the root node here.
   // Otherwise, call the insert method of class TreeNode.
   public synchronized void insertNode( int d )
   {
      if ( root == null )
         root = new TreeNode( d );
```

Fig. 22.16 Creating and traversing a binary tree (part 1 of 3).

```
      else
         root.insert( d );
   }

   // Preorder Traversal
   public synchronized void preorderTraversal()
      { preorderHelper( root ); }

   // Recursive method to perform preorder traversal
   private void preorderHelper( TreeNode node )
   {
      if ( node == null )
         return;

      System.out.print( node.data + " " );
      preorderHelper( node.left );
      preorderHelper( node.right );
   }

   // Inorder Traversal
   public synchronized void inorderTraversal()
      { inorderHelper( root ); }

   // Recursive method to perform inorder traversal
   private void inorderHelper( TreeNode node )
   {
      if ( node == null )
         return;

      inorderHelper( node.left );
      System.out.print( node.data + " " );
      inorderHelper( node.right );
   }

   // Postorder Traversal
   public synchronized void postorderTraversal()
      { postorderHelper( root ); }

   // Recursive method to perform postorder traversal
   private void postorderHelper( TreeNode node )
   {
      if ( node == null )
         return;

      postorderHelper( node.left );
      postorderHelper( node.right );
      System.out.print( node.data + " " );
   }
}
```

Fig. 22.16 Creating and traversing a binary tree (part 2 of 3).

from Java: How To Program, 3e
by Deitel & Deitel

```java
// Fig. 22.16: TreeTest.java
// This program tests the Tree class.
import com.deitel.jhtp3.ch22.Tree;

// Class TreeTest definition
public class TreeTest {
   public static void main( String args[] )
   {
      Tree tree = new Tree();
      int intVal;

      System.out.println( "Inserting the following values: " );

      for ( int i = 1; i <= 10; i++ ) {
         intVal = ( int ) ( Math.random() * 100 );
         System.out.print( intVal + " " );
         tree.insertNode( intVal );
      }

      System.out.println ( "\n\nPreorder traversal" );
      tree.preorderTraversal();

      System.out.println ( "\n\nInorder traversal" );
      tree.inorderTraversal();

      System.out.println ( "\n\nPostorder traversal" );
      tree.postorderTraversal();
      System.out.println();
   }
}
```

Fig. 22.16 Creating and traversing a binary tree (part 3 of 3).

```
Inserting the following values:
39 69 94 47 50 72 55 41 97 73

Preorder traversal
39 69 47 41 50 55 94 72 73 97

Inorder traversal
39 41 47 50 55 69 72 73 94 97

Postorder traversal
41 55 50 47 73 72 97 94 69 39
```

Fig. 22.17 Sample output from the program of Fig. 22.16.

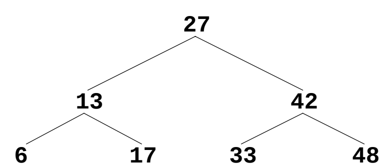

Fig. 22.18 A binary search tree.

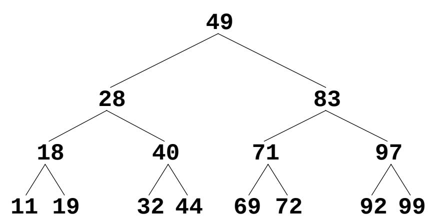

Fig. 22.19 A 15-node binary search tree.

```
               99
          97
               92
     83
               72
          71
               69
49
               44
          40
               32
     28
               19
          18
               11
```

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)`**. Note that 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. 22.20 Simple commands.

```
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. 22.21 Simple program that determines the sum of two integers.

```
10 rem   determine and print 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. 22.22 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. 22.23 Calculate the squares of several integers.

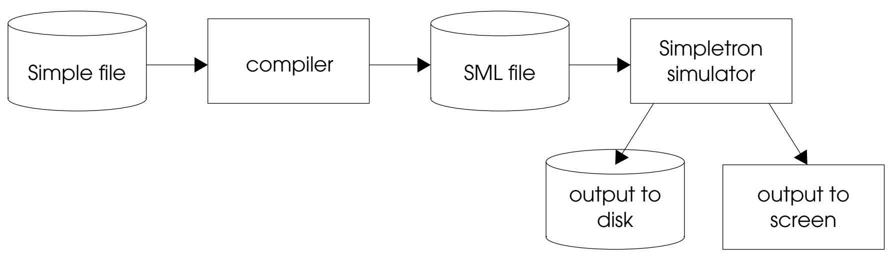

Fig. 22.24 Writing, compiling and executing a Simple language program.

from Java: How To Program, 3e
by Deitel & Deitel

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

Fig. 22.25 SML instructions produced after the compiler's first pass (part 1 of 2).

Simple program	SML location and instruction	Description
	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. 22.25 SML instructions produced after the compiler's first pass (part 2 of 2).

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**

Fig. 22.26 Symbol table for program of Fig. 22.25 (part 1 of 2).

Symbol	Type	Location
`55`	`L`	`15`
`60`	`L`	`15`
`99`	`L`	`16`

Fig. 22.26 Symbol table for program of Fig. 22.25 (part 2 of 2).

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

Fig. 22.27 Unoptimized code from the program of Fig. 22.25.

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. 22.28 Optimized code for the program of Fig. 22.25.