Answers to Machine Representation Problem Set

1. Assembly- to machine-language translation

Assembly-language version / *Machine-language version*

```
start:   INPUT   n              (01)  +120
         LOAD    #0             (02)  +323
         STORE   total          (03)  +421
         LOAD    #1             (04)  +322
loop:    STORE   i              (05)  +419
         LOAD    n              (06)  +320
         SUB     i              (07)  +619
         JNEG    done           (08)  +917
         LOAD    total          (09)  +321
         ADD     i              (10)  +519
         ADD     i              (11)  +519
         SUB     #1             (12)  +622
         STORE   total          (13)  +421
         LOAD    i              (14)  +319
         ADD     #1             (15)  +522
         JUMP    loop           (16)  +705
done:    OUTPUT  total          (17)  +221
         HALT                   (18)  +700

i:       0                      (19)  +000
n:       0                      (20)  +000
total:   0                      (21)  +000

                                (22)  +001
                                (23)  +000
```

} *These two values can be reversed*

What output does this program produce if you enter 5 as the input value?	25
What value does this program compute in general?	the square of the input number

2. MiniSim coding

Write a MiniSim program **`remainder.asm`** that requests two numbers from the user (which you may assume are both positive) and then computes the remainder of the first divided by the second. The problem, of course, is that MiniSim has only **`ADD`** and **`SUB`** instructions, and doesn't support multiplication and division. On the other hand, you can easily simulate the process of division by repeatedly subtracting the second number from the first until the result is negative. The remainder is the value immediately before the last subtraction.

Answer to problem 2

```
/*
 * File: remainder.asm
 * -------------------
 * This program computes the remainder of two input numbers.
 * The implementation simulates the following Java program:
 *
 *    public void run() {
 *        int n1 = readInt(" ? ");
 *        int n2 = readInt(" ? ");
 *        while (n1 - n2 >= 0) {
 *            n1 -= n2;
 *        }
 *        println(n1);
 *    }
 */

start:  INPUT    n1
        INPUT    n2
loop:   LOAD     n1
        SUB      n2
        JNEG     done
        STORE    n1
        JUMP     loop
done:   OUTPUT   n1
        HALT

/* Variables */

n1:     0
n2:     0
```

3a)

Suppose that the class `IndexList` has been defined as follows:

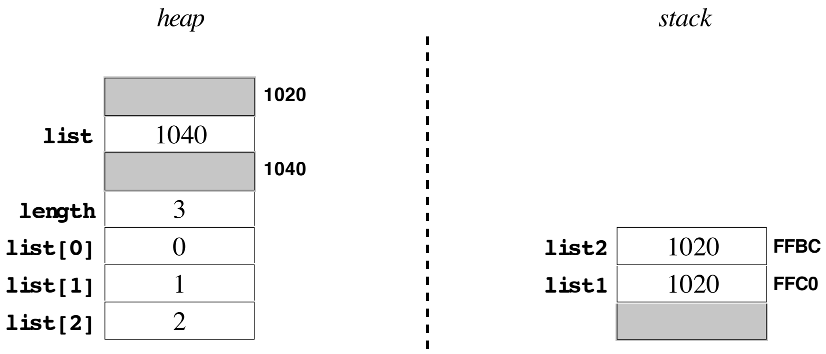
```
public class IndexList {

   public IndexList(int n) {
      list = new int[n];
      for (int i = 0; i < n; i++) {
         list[i] = i;
      }
   }

   private int[] list;

}
```

and that the method `testIndexList` looks like this:

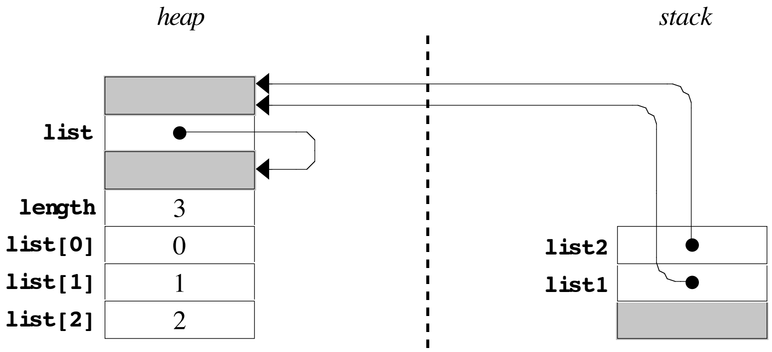
```
public void testIndexList() {
   IndexList list1 = new IndexList(3);
   IndexList list2 = list1;
}
```

←*Diagram at this point*

Using the heap-stack diagrams in Chapter 7 as a model, draw a diagram showing how memory is allocated just before `testIndexList` returns. You need not include explicit addresses in your diagram, but must indicate—either through addresses or arrows—where reference values point in memory. Your diagram should also include the names of any variables or fields.



If you use arrows for this problem, the diagram would appear as follows:



3b)

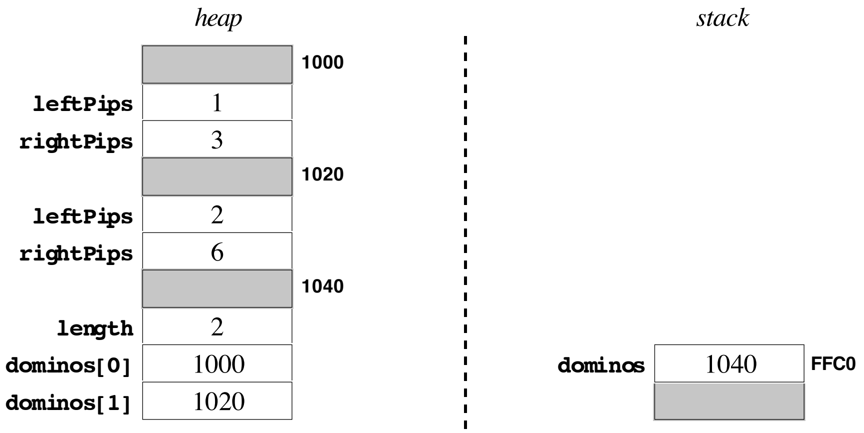
Suppose that the class `Domino` has been defined as follows:

```
public class Domino {

   public Domino(int p1, int p2) {
      leftPips = p1;
      rightPips = p2;
   }

   private int leftPips, rightPips;
}
```

and that the method `testDominos` looks like this:

```
public void testDominos() {
   Dominos[] dominos = new Dominos[2];
   dominos[0] = new Domino(1, 3);
   dominos[1] = new Domino(2, 6);
}
```

←*Diagram at this point*



If you use arrows for this problem, the diagram would appear as follows: