MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.090—Building Programming Experience
IAP 2007

Problem Set 5
Due Thursday January 18, 1pm

The Game of Nimrod

Long, long ago in places far far away, way before the advent of the computer game, neanderthals played games with sticks. In the pursuit of the wisdom of the ancients, we return to this humble setting in order to come to a deeper understanding of the universe and computer programming.

One such game was called Nimrod (after the man who invented it, Nimble Roderick). Nimrod is played with two piles of rods and two players. The objective is to be player who takes the last rod. A valid move is to take any number of rods from either pile, or an equal number from both, but never an unequal number from both. Play progresses until no rods remain.

A sample game:

```
Pile A: 5     Pile B: 9

Jacob takes 0 from pile A and 2 from pile B

Pile A: 5     Pile B: 7

Jessica takes 2 from pile A and 2 from pile B

Pile A: 3     Pile B: 5

Jacob takes 3 from pile A and 0 from pile B

Pile A: 0     Pile B: 5

Jessica takes 0 from pile A and 5 from pile B

Pile A: 0     Pile B: 0

Jessica wins! Jacob loses!
```

Problems

Download the file nimrod.scm from the course website and open it in scheme. You'll find a bunch of procedure definitions, some of them written and some not. You will fill in the unwritten definitions as part of this project. After writing each of the procedures, test the procedure(s) to see if they work. When you submit your work, submit the resulting nimrod.scm file with test-cases for every problem. The test cases should be commented out.

Problem 1

Write a data abstraction for the piles. The piles contains two values: the number of rods in each pile. The constructor `make-piles` builds a piles from a pair of numbers. The selectors `pile-a` and `pile-b` pick out the lesser and greater pile respectively.

Problem 2

Implement the `empty-piles?` and `valid-piles?` procedures. The piles are valid if they have a reasonable number of rods in them. You only need to check for piles with negative numbers of rods.

Problem 3

Implement the move abstraction.

Problem 4

Implement `valid-move?`, which returns true if the input move is a valid move according to the rules. Don't worry about what the state of the piles would be afterward (taking 600 from both piles is a valid move that probably results in invalid piles). Examples of invalid moves include (0,0), (0, -1), or (3, 4). Hint: Using `not`, `and`, `or` will make for cleaner code.

Problem 5

Implement `apply-move`, which takes piles and a move, and returns the piles that result from doing that move. Again, don't worry if the move or resulting piles are valid; that is checked elsewhere.

Problem 6

Play a game of nimrod using the `play-nimrod` procedure. For initial piles, use the `random-piles` procedure to generate some. (Values between 1 and 15 produces nice short games). For strategies, use the `human-strat` and play against someone (possibly yourself). When prompted for a number, enter the number and hit RETURN. Then play `human-strat` against `simple-strat` and try to beat the computer. Turn in one log of your games.

Problem 7

Write a more interesting strategy and test it out. Start with the definition of `simple-strat` and use that as a guide to write your own version.

Optimal Strategy

The playing optimal strategy requires knowing the "lose positions". A lose position is a piles state from which the player next to play is guaranteed to lose against an optimal player. The smallest lose position is (0 0) by definition. By experimentation, the lose position (1 2) is quickly discovered. It takes a little more work to see the rest:

```
(0 0)
(1 2)
(3 5)
(4 7)
(6 10)
(8 13)
(9 15)
...
```

Two things to notice: the difference between the two piles increases by one for each successive lose position and each positive integer appears as either the low or high value in a lose position. These two things combine to ensure that no matter the state of the piles, if they are not currently in a lose position, it takes exactly one move to move the piles into a lose position.

The generation strategy is relatively simple: To generate the ith lose position, pick the lowest unused integer from all the lose positions smaller than i – that unused integer is the smaller of the pair, we'll call n. The larger of the pair is $i + n$. Cross off the two integers that were just used.

For example, the 0th lose position is (0,0). The $i = 1$ position is (1,2): $n = 1$ is the smallest unused integer, and $1 + 1 = 2$. The $i = 2$ position is (3,5): $n = 3$ is the smallest unused integer, and $2 + 3 = 5$.

Problem 8

First we need to be able to generate a list of a bunch of integers. Write `list-of-ints` which returns a list of integers in a given range.

Problem 9

Now we need a way to "cross off" a number from a list of numbers. Do this by completing the procedure `without-n` that when given a list of numbers, returns a new list which contains all the numbers except the number n. You may assume that there are no duplicate numbers in the list.

Problem 10

Finish the implementation of `lose-position-helper` by filling in the two blanks. Blank 1 constructs the new lose position. Blank 2 crosses off the used numbers from the list.

Problem 11

Implement `move-to`, which tries to generate a valid move that will move from the initial piles to the objective piles. For example, if the initial piles are (4 6) and the objective piles are (3 5), it should generate (1 1) as the required move. For initial (1 2) and objective (3 5), there is no valid move, so it should return false. Also test your code with initial (5 12) and objective (3 5) to make sure it works with "backwards" pile counts. Hint: Problem 4 may come in handy.

Problem 12

Test out the optimal strategy! Play it against itself a bunch of times to make sure it's working. Bask in the glory of success! (no need to turn anything in for this question)