CS3L Summer 2011 Final Exam, Part 1

25 points

* You have **up to 60 minutes** to code answers to these problems. If you finish early, you may begin work on the written Part 2 of the final, but once you stop working on Part I, you will no longer be able to use your computer. We recommend that you spend no more than 45 minutes on Part I, and after 60 minutes, everyone must stop using their computers and move on to Part 2 of the final.

* You may use STk, the Unix terminal, your favorite text editor, and your "cheat sheet". You may not refer to the Internet or to your own previous work. You may not communicate with other students in any way. **Violations of these rules will result in a 0 on this part of the exam.**

* You may use either your own laptop or one of the computers in the room. It's OK to ssh into a different computer; you might want to stake one out in advance!

* Put all your code in a file called `finalexam.scm`. Make sure your procedures have exactly the names and behavior that we've specified. Use `submit finalexam` when you are ready, or once time has run out. You may resubmit up until the deadline, but **only your last submission will be graded**.

You have two cups, the first of which holds a maximum volume of `v1` ounces of water, and the second of which holds a maximum volume of `v2` ounces of water. (`v1` and `v2` are different positive integers; `v1` may be larger than `v2`, or vice versa.) Both cups are initially empty. You also have access to a faucet that can produce as much water as you need. You have been challenged to produce exactly `target` ounces of water in as few actions as possible. (`target` is a positive integer smaller than the maximum of `v1` and `v2`.) Each time you act, you must choose one of the following actions:

1. Choose one of your cups and pour any water in that cup out, making that cup empty

2. Choose one of your cups (which might be empty, or might already have some water in it) and fill it all the way up with water

3. Choose one of your cups (call it "from") and pour water from that cup into the other cup (call it "to") until one of the following occurs, at which point you stop immediately: either there is no more water left in the "from" cup, or the "to" cup is full.

If, after taking an action, there is a cup that has exactly `target` ounces of water in it, you win the game.

You have no measuring devices, and you can't estimate volumes precisely, so you can't do something like fill a cup partway up from the faucet. The only actions you may perform are the ones above, and you may not mix actions or perform them incompletely.

For example, let's say you have a 12 ounce cup and a 20 ounce cup, and the goal is to produce 16 ounces. (That is, v1 = 12, v2 = 20, and target = 16.) The most efficient way to do this wins the game after 6 actions:

Action #	**Action**	**Water in 12 oz cup after action**	**Water in 20 oz cup after action**
Initial state	n/a	0	0
1	Fill 20 oz cup	0	20
2	Pour from 20 oz cup to 12 oz cup	12	8
3	Empty 12 oz cup	0	8
4	Pour from 20 oz cup to 12 oz cup	8	0
5	Fill 20 oz cup	8	20
6	Pour from 20 oz cup to 12 oz cup	12	**16**

Your ultimate goal is to write a procedure `(solve-game v1 v2 target)` that finds a solution requiring the smallest number of actions. `solve-game` should return a list consisting of all of the states of the game leading up to the solution, **in the correct order**; each state is a two-item list containing the volume of water in cup 1, followed by the volume of water in cup 2. In the case of the above example, `solve-game` would return:

```
((0 0) (0 20) (12 8) (0 8) (8 0) (8 20) (12 16))
```

If there are multiple solutions requiring the same number of actions, `solve-game` may return any one of them.

Throughout these problems, do not use mutation of any sort. (You also really wouldn't find it very helpful!)

1. (5 points) Write a procedure `(pour wfrom wto vto)` that takes three arguments:

* `wfrom`, the amount of water in the cup that you're pouring from (the "from" cup) before the pouring begins.
* `wto`, the amount of water in the "to" cup before the pouring begins.
* `vto`, the maximum volume that the cup that you're pouring into (the "to" cup) can hold.

and returns a two-item list containing the volumes of water in the “from” cup and the “to” cup after the pouring is complete, in that order.

Once pouring starts, it continues until either the “from” cup is empty, or the “to” cup is full. Pouring ceases immediately whenever at least one of these conditions is met. (There is never any overflowing or spillage.)

Examples:

```
STk> (pour 20 8 12)
(16 12) ; pouring stops because the "to" cup becomes full

STk> (pour 15 3 20)
(0 18) ; pouring stops because the "from" cup becomes empty

STk> (pour 13 9 9)
(13 9) ; no water moves, because the "to" cup starts off full

STk> (pour 0 6 6)
(0 6) ; no water moves, because the "from" cup starts off empty
```

Hint: You may find it easiest to write `pour` as a recursive procedure that transfers 1 ounce of water at a time.

2. (5 points) You will eventually need a procedure `(shortest-solution sollist)` that takes a list `sollist` of solutions (each of which is a list of two-item lists) and returns the shortest one, or, if multiple lists are tied for the shortest, any of the shortest ones. If `sollist` is initially empty, shortest-solution should return the word `no-solution`.

Example: (these are the solutions to the problem with v1 = 2, v2 = 3, target = 1)

```
STk> (shortest-solution
'( ((0 0) (2 0) (2 3) (0 3) (2 1))
   ((0 0) (2 0) (0 2) (2 2) (2 3) (0 3) (2 1))
   ((0 0) (2 0) (0 2) (2 2) (1 3))
   ((0 0) (2 0) (0 2) (0 3) (2 1))
   ((0 0) (0 3) (2 3) (2 0) (0 2) (2 2) (1 3))
   ((0 0) (0 3) (2 1)) ))
((0 0) (0 3) (2 1))
STK> (shortest-solution '())
no-solution
```

3. (15 points) Write `(solve-game v1 v2 target)`. Generally speaking, your code will not be judged on efficiency, but we reserve the right to give only partial credit or no credit if your code does something egregiously slow and inefficient that would not be a viable solution in real life (e.g., generates *all* possible lists of *all* possible two-item lists of lengths up to 9999 of *any* integers up to the maximum volumes in the problem, and then sees which of those constitute valid games, and then...)

Examples:

```
STk> (solve-game 3 5 4)
((0 0) (0 5) (3 2) (0 2) (2 0) (2 5) (3 4))
STk> (solve-game 5 3 2)
((0 0) (5 0) (2 3))
STk> (solve-game 3 4 5)
no-solution
STk> (solve-game 3 6 5)
no-solution
```

This is a hard problem, and some people may not finish it or get it working successfully. There will definitely be partial credit!

Hint: Any problem that has at least one solution also has infinitely many solutions (because you can always add in arbitrarily many useless actions that don't make progress toward the goal), so your program should keep track of which states have already been seen by a particular solution path, and stop exploring any path that returns to a previous state. Your program only needs to output a most efficient solution, and we guarantee that throwing out any solution that revisits a state will not cause you to throw out the right answer!