CSE 413 – Homework #4

DUE: Thursday October 28, 2004, 11:59 p.m.

As before, there are three steps to turning in your homework:
a.) Use the turnin link from the calendar page; follow the instructions and submit the file solver.scm.
b.) Then complete the web survey that you are directed to.
c.) Print a copy of your solver.scm and turn it in during class on Friday October 29. Write your name and UW Net ID prominently on the first page.

General advice: This homework is more open-ended than previous homework. The final solution does not require a lot of code (it can be done in less than 50 lines), but may require more thought to decide what to do. **Start early!**

Symbolic algebra was one of the original application areas for Lisp, the ancestor of Scheme. In this assignment, you will write a procedure that solves simple symbolic equations. Your procedure should be defined in the file solver.scm.

The procedure solver has two arguments: a symbol `var` and an equality expression `eqn`. The symbol should appear once somewhere in the equality expression, either as part of the left operand or the right operand. The result of `(solver var eqn)` should be another equality expression where `var` appears alone as the left operand and the right operand is the original equation, solved for `var`.

The top level of the input equation `eqn` is a list of the form `(= exp1 exp2)`. The two expressions `exp1` and `exp2` are legal Scheme expressions formed from atoms and numbers using +, -, *, and / only. You should assume that each of these operators is binary (ie, has exactly two arguments). You don't need to worry about division by zero.

If there is exactly one occurrence of `var` in `eqn`, then solve the equation and return the solution. Otherwise, return the string "Too Hard". Except for checking for the right number of occurrences of `var`, you can assume that the input is well-formed.

The key to designing this procedure is to unwind the equation, one operator at a time, building a new and equivalent equation at each step.

For example, the equation x*7=2+b will be provided to your solver in the following form:

```
(solver 'x '(= (* x 7) (+ 2 b)))
```

You solve this by noticing first that there is exactly one x in the equation. Then you transform the existing left hand side by reducing it to the left operand of the left hand side, and transform the right hand side by using the inverse of the left hand side operator, the existing right hand side, and the right operand of the left hand side. In other words,

$$x * 7 = 2 + b$$
$$x = (2 + b) / 7$$

or (= (* x 7) (+ 2 b)) becomes (= x (/ (+ 2 b) 7))

The file expression.scm is provided to you, and contains definitions for constructor and accessor functions for binary expressions.

1. You will probably want to have helper functions to streamline your code a little. One useful function is (count-x x exp) that counts the number of times the symbol x appears in the expression tree exp. Use the accessor functions provided in expression.scm to traverse the tree and count the number of times you find the symbol x in either a left expression or a right expression.

2. Another useful function is (inverse-op o). Given an operator symbol like '+, it returns the inverse operation '-. This is useful when building the new transformed expression from the old expression.

3. The third useful function is an iterative procedure that takes existing left hand side and right hand side expressions and ultimately (after iterating as needed) returns a right hand side that is solved for the desired symbol.

4. Write the solver procedure. The file hw3-test-it.scm contains several test cases with expected solutions. **You do not have to simplify the right hand side (e.g. to notice that multiplication by 1 can be eliminated). See the extra credit.**

5. Once your solver is running without major problems, you can run it again with run-solver-from-file.scm. This file contains a procedure that reads equations from a file, passes them to your solver, and then writes the results to a text file. The file automatically runs with the equations in x.txt, producing the text file x-solved.txt, then does it again with y.txt and y-solved.txt.

Extra Credit: Write a new function `simplify-solver` that behaves like `solver` but tries to simplify the result using standard mathematical identifies. In your comments, describe what simplifications you do and list some sample equations that show what your solver can do (list both the equations and the output of your solver). Include this function in the same file solver.scm. This will be worth a modest amount of extra credit.