Computer Laboratory 1
CSCI 1913: Introduction to Algorithms, Data Structures, and Program Development
Tuesday–Wednesday, September 13–14, 2016

0. Introduction.

It's possible to solve an equation *numerically*, by substituting numbers for its variables. It's also possible to solve an equation *symbolically*, by using algebra. For example, to solve the equation $m \times x + b = y$ symbolically for x, you'd first subtract b from both sides, giving $m \times x = y - b$. Then you'd divide both sides by m, giving $x = (y - b) \ / \ m$. You may assume that no variable is equal to zero.

In this laboratory exercise, you'll write a Python program that uses algebra to solve simple equations symbolically. Your program will use Python tuples to represent equations, and Python strings to represent variables. To simplify the problem, the equations will use only the binary arithmetic operators '+', '−', '×', and '/'. Also, your program need only solve for a variable that appears exactly once in an equation. Your program will use ideas from the constant expression evaluator that I discussed in the lecture.

1. Theory.

Here's a mathematical description of how your program must work. First, $L \to R$ means that an equation L is algebraically transformed into a new equation R. For example:

$$A + B = C \quad \to \quad B = C - A$$

Second, a variable is said to be *inside* an expression if it appears in that expression at least once. For example, the variable x is inside the expression $m \times x + b$, but it isn't inside the expression $u - v$. Each variable is considered to be inside itself, so that x is inside x.

Now suppose that $A \circ B = C$ is an equation, where A, B, and C are expressions, and $\circ$ is one of the four binary arithmetic operators. Also suppose that the variable x is inside either A or B. Then the following rules show how this equation can be solved for x.

$$A + B = C \to \begin{cases} A = C - B & \text{if } x \text{ is inside } A \\ B = C - A & \text{if } x \text{ is inside } B \end{cases} \tag{1}$$

$$A - B = C \to \begin{cases} A = C + B & \text{if } x \text{ is inside } A \\ B = A - C & \text{if } x \text{ is inside } B \end{cases} \tag{2}$$

$$A \times B = C \to \begin{cases} A = C \ / \ B & \text{if } x \text{ is inside } A \\ B = C \ / \ A & \text{if } x \text{ is inside } B \end{cases} \tag{3}$$

$$A \ / \ B = C \to \begin{cases} A = C \times B & \text{if } x \text{ is inside } A \\ B = A \ / \ C & \text{if } x \text{ is inside } B \end{cases} \tag{4}$$

For example, I can use the rules to solve the equation $m \times x + b = y$ for x. In Rule 1, A is $m \times x$, and B is b. Since x is inside A, I can transform the equation to $m \times x = y - b$. Then in Rule 3, A is m, and B is x. Since x is inside B, I can transform the equation to $x = (y - b)/m$. Now x is alone on the left side of the equal sign, so the equation is solved. This solution used only two rules, but a more complex equation might use more rules, and it might use rules more than once.

2. Representation.

Your program must represent operators and variables as Python strings. For example, it must represent the variable x as the string `'x'`. It must also represent equations and expressions as Python tuples with three elements each. For example, it must represent the expression $a + b$ as the Python tuple `('a', '+', 'b')`. These tuples can be nested, so that the equation $m \times x + b = y$ is represented like this:

```
(((('m', '*', 'x'), '+', 'b'), '=', 'y')
```

I've used an asterisk `'*'` as the multiplication operator. If you ignore the parentheses, commas, and quotes, then this tuple looks much like how you'd write the original equation. It's helpful to define functions `left`, `op`, and `right` that return the parts of tuples that represent expressions.

```
def left(e):
  return e[0]

def op(e):
  return e[1]

def right(e):
  return e[2]
```

For example, if `e` is the tuple `('a', '+', 'b')`, then `left(e)` returns `'a'`, `op(e)` returns `'+'`, and `right(e)` returns `'b'`.

3. Implementation.

Your program must define the following Python functions, and those functions must behave as described here. You must use the same function names as I do—this will make your program easier to grade. However, you need not use the same parameter names as I do. Your functions cannot change elements of the tuples that are passed to them as arguments, because tuples are immutable. Each function is worth 5 points, so the whole program is worth 35 points.

- `isInside(v, e)`. Test if the variable `v` is inside the expression `e`. It's inside if (1) `v` equals `e`, or (2) `v` is inside the left side of `e`, or (3) `v` is inside the right side of `e`. This definition is recursive. Other functions in your program will need to call `isInside`. Hint: Don't use the Python operator `in` when you write this function: it doesn't do what you want.

- `solve(v, q)`. Solve the equation `q` for the variable `v`, and return a new equation in which `v` appears alone on the left side of the equal sign. For example, if you call `solve` like this:

  ```
  solve('x', ((('m', '*', 'x'), '+', 'b'), '=', 'y'))
  ```

  then it will return this:

  ```
  ('x', '=', (('y', '-', 'b'), '/', 'm'))
  ```

The function `solve` really just sets things up for the function `solving`, which does all the work. If `v` is inside the left side of `q`, then call `solving` with `v` and `q`. If `v` is inside the right side of `q`, then call `solving` with `v` and a new equation like `q`, but with its left and right sides reversed. In either case, return the result of calling `solving`. If `v` is inside neither side of `q`, then return `None`, which will probably cause an error.

- `solving(v, q)`. Whenever this function is called, the variable `v` must be inside the left side of `q`. If `v` is equal to the left side of `q`, then the equation is solved, so simply return `q`. Otherwise, decide which of the four transformation rules (from Section 1) must be used next to solve `q`. Call the function that implements that rule on `v` and `q`, then return the result.
- `solvingAdd(v, q)`. Use rule 1 to transform the equation `q`, then call `solving` on the variable `v` and the transformed `q`. Return the result of calling `solving`.
- `solvingSubtract(v, q)`. Use rule 2 to transform the equation `q`, then call `solving` on the variable `v` and the transformed `q`. Return the result of calling `solving`.
- `solvingMultiply(v, q)`. Use rule 3 to transform the equation `q`, then call `solving` on the variable `v` and the transformed `q`. Return the result of calling `solving`.
- `solvingDivide(v, q)`. Use rule 4 to transform the equation `q`, then call `solving` on the variable `v` and the transformed `q`. Return the result of calling `solving`.

The functions `solvingAdd`, `solvingSubtract`, `solvingMultiply`, and `solvingDivide` will have very similar definitions. If you can write one of these functions, then you can also write the other three. You need not write any kind of user interface that reads equations from the keyboard, or writes equations to the display. Instead, just call the function `solve` directly with a nested tuple that represents an equation.

4. Tests.

The file `tests.py` on Moodle contains a series of tests. Each test calls a function from your program, then prints what the function returns. It is followed by a comment that tells what must be printed if the function works correctly. While you're writing your program, you can use these tests as examples of what the functions are supposed to do.

5. Deliverables.

Run the tests in `tests.py`. Also test `solve` on at least one equation of your own design. Then turn in (1) your program, (2) the tests, and (3) the results of the tests. Your TA will tell you how and where to turn them in. If your lab is on Tuesday, then your work must be turned in by midnight on Tuesday, September 20, 2016. If your lab is on Wednesday, then your work must be turned in by midnight on Wednesday, September 21, 2016.