CS61A Summer 2010

George Wang, Jonathan Kotker, Seshadri Mahalingam, Steven Tang, Eric Tzeng

Homework 7

Due: Monday, August 9, 2010, at 7AM

*N*ote: The number of stars (★) besides a question represents the ***estimated*** relative difficulty of the problem: the more the number of stars, the harder the question.

1 How to Start and Submit

First, download `http://inst.eecs.berkeley.edu/~cs61a/su10/hw/hw7.scm` and use it as a template while filling out your procedures. See `http://www-inst.eecs.berkeley.edu/~cs61a/su10/hw-faq.pdf` for submission instructions.

2 Lazy Evaluator

A version of the lazy evaluator is available online in `~cs61a/lib/lazy.scm`.

1. SICP Exercises.
   (a) (★) 4.25
   (b) (★★) 4.27
   (c) (★★) 4.28
2. (★) Consider the following series of definitions:

```
(define (square x) (* x x))

(define (sum-squares x y)
  (+ (square x) (square y)))

(sum-squares (square 3) (square 4))
```

   How many times would the `square` procedure be called on the last invocation of the `sum-squares` procedure if Scheme ran in (a) applicative order, or in (b) normal order?
3. (★★) Consider the following series of definitions:

```
(define (foo x y)
  (if (> x y)
      (/ x y)
      x))

(define (bar x y)
  (if (> x 1)
      (/ x y)
      x))
```

   For each of the following interactions, determine if the lazy evaluator is more efficient than the regular metacircular evaluator.

(a) `(foo (* 2 3) (* 5 2))`

(b) `(bar (foo 10 5) 1)`

(c) `(bar 1 (foo 10 5))`

3 Nondeterministic Evaluator

A version of the nondeterministic evaluator is available online in `~cs61a/lib/vambeval.scm`. (For this week, do *not* use the version located at `ambeval.scm`, where the name is without the `v`.) A version that includes the `and` and `or` special forms is available at `http://inst.eecs.berkeley.edu/~cs61a/su10/hw/vambtables.scm`.

1. (★) SICP 4.35 (The lower and upper bounds are inclusive.)
2. (★★) SICP 4.37
3. (★) SICP 4.42

4 Logic Programming

A version of the logic evaluator is available online in `~cs61a/lib/query.scm`. For all problems that involve writing queries or rules, test your solutions. For questions 1 and 2, you need to load in the data from the Microshaft database. You can do so with the following commands:

```
> (load "~cs61a/lib/query.scm")
> (initialize-data-base microshaft-data-base)
> (query-driver-loop)
```

1. (★) SICP 4.56
2. (★★) SICP 4.58
3. (★) Write a relation `length` that returns the length of a list. The relation should work as follows:

   ```
   > (length (foo bar baz) ?z)
   (length (foo bar baz) (a a a))
   ```

   Note that in our version of logic programming, we represent numbers by a list of `a`s. As a result, `1` is denoted by `(a)`, `2` is denoted by `(a a)`, `5` is denoted by `(a a a a a)`, `0` is denoted by `()`, and so on.
4. Write a relation `max` that returns the maximum between two numbers. For example:

   ```
   > (max (a a a a a) (a a a) ?z)
   (max (a a a a a) (a a a) (a a a a a))

   > (max () (a a a) ?z)
   (max () (a a a) (a a a))
   ```
5. (★★ **Optional**, but good practice) Write a relation `depth` that returns the depth of a deep list. For example:

   ```
   > (depth (foo bar baz) ?z)
   (depth (foo bar baz) (a))

   > (depth (foo (bar) baz) ?z)
   (depth (foo (bar) baz) (a a))
   ```

```
> (depth (foo ((bar (baz))) foo) ?z)
(depth (foo ((bar (baz))) foo) (a a a a))
```

You may find the `max` relation that you defined in question 4 useful.

5 Case Study (Optional): Sudoku Solver

5.1 Introduction

Sudoku is a puzzle that has been gaining a steady (and large!) following in the past few years. The basic premise of the game is this: we have a 9×9 grid with some numbers filled in. The objective is to fill in each column and row with numbers from 1 to 9, such that no column or row has duplicate numbers. We also have the additional constraint that each 3×3 sub-grid, starting from the top left, should also be filled with numbers from 1 to 9, sans duplicates. If you have not played the game before, you can try your hand at it on `http://www.websudoku.com`.

We will implement a Sudoku puzzle solver using the nondeterministic evaluator. In the interest of time and debugging, we will focus on smaller 4×4 Sudoku puzzles.

Download the solver template from `http://inst.eecs.berkeley.edu/~cs61a/su10/hw/sudoku.scm` and the modified nondeterministic evaluator from `http://inst.eecs.berkeley.edu/~cs61a/su10/hw/vambtables.scm`.

5.2 Modifications to the Evaluator

In the `vambtables` Scheme file, we have provided you with a nondeterministic evaluator based on the regular metacircular evaluator, but we have augmented it with the ability to work with two-dimensional tables, as well as with the ability to understand `and` and `or` statements.

5.3 Tables, Puzzles, and Procedures

We have provided the `make-table` procedure in the `vambtables` Scheme file, which takes one argument `n` and returns an $n \times n$ two-dimensional table. It implements a table as a vector of equal-length vectors. We have also provided the `insert!` procedure to insert an element into the table, and the `lookup` procedure to look up an element from the table. The table coordinates start counting from (0, 0).

In the `sudoku` Scheme file, we have initialized an empty table called `sudoku-grid`. This is the grid that we will need to fill in with the correct answers.

A Sudoku puzzle is represented as a list of lists, one for each row in the puzzle. Each row is an association list, where each key is a position in the row, and the corresponding value is the number at that position in the Sudoku puzzle. We have provided four 4×4 puzzles for you to work with, as well as one empty 4×4 puzzle for you to debug your code with.

The main solver procedure is the `solve-sudoku` procedure, which takes in the puzzle to be solved and the grid to place the final answer in. In turn, it calls the `fill-row` workhorse procedure to fill in each row of the grid. We will be implementing most of our changes in the `fill-row` procedure.

5.4 Running the Solver

In order to run the solver, first load the `vambtables` Scheme file. Run the nondeterministic evaluator with the command `(mce)`. Then, copy and paste all of the code from the `sudoku` Scheme file into the evaluator, and call the `solve-sudoku` procedure with the appropriate arguments: the puzzle you are trying to solve and `sudoku-grid`. The procedure should return the `sudoku-grid` vector, but fully solved, with all entries in the grid filled.

5.5 Exercises

1. (★) Copy your code from section 3 question 1, to implement the `an-integer-between` procedure.
2. (★★) Fill in the `generate-subgrid-coords` procedure, whose arguments are the coordinates of the bottom-right corner of the subgrid, and which returns a list of the coordinates of all of the cells in the subgrid. An example interaction is:

   ```
   > (generate-subgrid-coords 1 1)
   ((1 . 1) (1 . 0) (0 . 1) (0 . 0))
   ```

   Assume that the Sudoku grid, and thus its subgrids, is square. You may use the `SUBGRID-WIDTH` constant that we have defined for you.
3. (★) Currently, the `fill-row` procedure fills a grid from left to right across a row. However, it does not enforce the conditions required for a correct Sudoku grid. Add a condition to ensure row consistency: in other words, add a condition to ensure that all of the elements in the row that have already been added are distinct from each other. You may find the `distinct?` procedure, defined near the top of the file, useful. Test your code with the empty 4×4 puzzle, and ensure that each row only has one occurrence of each of the numbers from `1` to `4`.
4. (★★) We have ensured row consistency; now, ensure column consistency. In other words, add a condition to ensure that all of the elements in the jth column (up until the ith row) are distinct from each other. Test your code with the empty 4×4 puzzle.
5. (★★) Finally, also ensure subgrid consistency. In other words, add a condition to ensure that all of the elements in a subgrid are distinct from each other. However, you only need to check for this condition when you are filling in cells that represent the bottom-right cells of subgrids: how can you tell if a cell is the bottom-right cell of a subgrid? (*Hint.* Try it out: draw out a grid and determine the coordinates of the bottom-right cells of each subgrid, but as if you were to start counting coordinates from $(1, 1)$, instead of from $(0, 0)$. Do you see a pattern, especially with the divisibility of the coordinates by `SUBGRID-WIDTH`?)

   Test your code with the empty 4×4 puzzle. You may find the `generate-subgrid-coords` procedure useful.
6. (★) Run your solver on the easy and hard 4×4 puzzles. Bring your solution to the second hard 4×4 puzzle to the face-to-face grading session.

5.6 Extra for Experts

As of now, the Sudoku solver is quite slow for 4×4 puzzles. Explore different ways that you can speed the Sudoku solver up, and then augment your solver code to work with 9×9 puzzles. We have provided an easy 9×9 puzzle for you to work with. Currently, a naïve solution at the end of the above exercises does not finish solving the puzzle, even after chugging away for 12 hours.

6 Feedback

Now that you are done, please leave me some feedback at the following link regarding how the course is going. This is not worth points, but will give us valuable feedback that in turn improves your course experience. Thanks! `https://spreadsheets0.google.com/viewform?formkey=dFJEN3R2RzZXQkRlXzd6N0NTYl9NMUE6MQ#gid=0`