CS61A Summer 2010

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

Homework 4

Due: Monday, July 19 2010, at 7AM

Note: 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/hw4.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 Scheme OOP

1. (★★) Define a `random-generator` class, whose constructor takes one argument called `range`. Implement a method called `number` that returns a random number in the generator's range. Also, implement a method called `count` that returns how many instances of the random-generator class have been constructed. There is a primitive procedure called `random` that works as follows: the call `(random x)`, where `x` is a non-negative integer, returns a non-negative integer between 0 and one less than `x`.

   Examples:

   ```
   (define generator1 (instantiate random-generator 5))
   (define generator2 (instantiate random-generator 100))
   (define generator3 (instantiate random-generator 250))

   (ask generator1 'number) should return a random integer between 0 and 4,
   (ask generator3 'number) should return a random integer between 0 and 249,
   (ask generator1 'count) should return 3.
   ```

2. (★) Create a `CalSodaMachine` class, each of whose objects is instantiated with an initial number of cans. A `CalSodaMachine` should accept two messages, `refill` and `buy`. `Refill` takes one argument, a number of cans, and adds that many cans to the `CalSodaMachine`'s number of cans. `Buy` takes in a number and removes that number of cans from the `CalSodaMachine`, and then returns `"here you go"`. However, if there are no cans in the inventory when buy is given, or the number of cans in the inventory is less than the number asked for, `"not enough cans"` should be returned.

   Examples:

   ```
   (define calsodamachine1 (instantiate calsodamachine 10))
   (define calsodamachine2 (instantiate calsodamachine 30))

   (ask calsodamachine1 'buy 30) should return "not enough cans"
   (ask calsodamachine2 'buy 30) should return "here you go"
   (ask calsodamachine1 'refill 50)
   (ask calsodamachine1 'buy 30) should return "here you go"
   (ask calsodamachine2 'buy 1) should return "not enough cans"
   ```

3. (★★) A devious Stanford engineer decides to steal a `CalSodaMachine` and recreate it as a `StanfordSodaMachine`. However (of course), the engineer makes poor design decisions, and `StanfordSodaMachine`s have a security vulnerability. Write a `StanfordSodaMachine` class that accepts a starting number of cans, has the same `buy` and `refill` methods as a `CalSodaMachine` (do not explicitly write these methods; they should be inherited), and also an additional method `shake` that removes a can from the machines inventory. If there are zero cans left when shake is given, then return `"go bears"`. Use inheritance in your solution. Your answer should not use `set!` explicitly. Be sure to test your code!

   Examples

   ```
   (define stanfordsodamachine1 (instantiate stanfordsodamachine 10))
   (ask stanfordsodamachine1 'shake) should return "here you go"
   (ask stanfordsodamachine1 'buy 10) should return "not enough cans"
   (ask stanfordsodamachine1 'buy 9) should return "here you go"
   (ask stanfordsodamachine1 'shake) should return "go bears"
   ```

4. (★★) Define a class `ubiquitous`, each of whose objects is instantiated with an initial number, called `count`. Implement a method `SimplePlus` that takes in a number `n` and increments the object's count by `n`. Then, implement a method `AllPlus` that accepts a number argument, and increments the count of *all* `ubiquitous` *instances* by that number. Do not use a global variable to implement this counter. Hint: You may want more than one variable.

   Note: When testing, remember to reset your STk, since old objects do not change if the class definition is changed. They need to be reinitialized.

   Examples:

   ```
   (define counter1 (instantiate ubiquitous 5))
   (define counter2 (instantiate ubiquitous 5))
   (ask counter1 'simpleplus 3)
   (ask counter1 'allplus 2)
   ```

   At this point,

   ```
   (ask counter1 'count) should return 10, and
   (ask counter2 'count) should return 7.
   ```

5. (★★) Create a `button` class. It has no instantiation variables and has two methods. One is `check`, which returns the button's state. The other is `switch`, which changes the state of the button. Buttons can either be on or off. You may represent these however you wish.

   Now, define a `machine` class, whose objects will each maintain their own list of buttons. Implement the following methods in the `machine` class:

   - `add` - adds a button onto the list of buttons
   - `remove` - removes the most recently created button from the list of buttons
   - `check` - takes a number `n`, and returns `#t` if button `n` has been pressed, `#f` if button `n` has not been pressed, and returns `"no such button"` if the button does not exist. Buttons start at $n = 0$, which would correspond to the first element in the button list. You can either make button 0 the most recent button, or the very first button created; it is up to you.
   - `press` - takes a number `n`, and changes the state of button `n`. If the button does not exist, return "no such button".

3 Below the Line

1. SICP exercises (★) 3.3, (★) 3.4, (★★) 3.8. Do not use define-class or OOP-Scheme.

2. (★★★) Recreate `ubiquitous` from question 4, but do not use define-class or OOP-Scheme. You will want to think about what it means to have a class-variable or an instance-variable. You will also want to think about what the calls and answers to this procedure will look like. You have some latitude in terms of how to implement this procedure, but make sure that all the functionality of the original procedure was here.

4 Short Answer

1. (★★) Consider the following two procedures:

```
(define (foo x)
  (let ((t 1))
    (set! t (+ x 1))
    t))

(define foo
  (let ((t 1))
    (lambda (x) (set! t (+ x 1)) t)))
```

For each definition of `foo`, write what Scheme would return to the following successive calls:

```
(foo 4)
(foo 5)
(foo 6)
```

Explain what is different between the two procedures.

5 Environment Diagrams

For this section, you do not need to submit these online, although you certainly can. You can use paint, or just ascii art, or just draw them with pencil and paper and bring them to your face-to-face grading session.

1. (★) Draw an environment diagram for the following calls.

```
(define x 3)
(define (foo z)
  (+ z x))
(foo 7)
(define (awesome x)
  (foo x))
(awesome 100)
```

2. (★★) What does the following program do, and why? Draw the environment diagram. What if we typed `x` at the end? What would be returned?

```
(define x 4)
(define (changexto1 x)
 (set! x 1))
```

3. (★★★) Draw an environment diagram for the following calls.

```
(define x 3)
(define (foo z)
  (+ z x))
(define (feed d)
  (foo d))
(define (amazing h)
  (let ((x 4)
        (+ (lambda (s t) (+ s t x))))
       (feed (+ x h))))
(amazing 7)
```

6 Extra for Experts (Optional)

Discuss multiple inheritance patterns. Why might it be better to inherit from a second-choice parent instead of a first-choice grandparent? When might it be better to go inherit from first grandparents before second parents? Come up with an example for each.

7 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://spreadsheets.google.com/viewform?formkey=dFNWdVZJREhxdWhBendJY2hfb2RiLUE6MQ`