CS 61A Summer 2009 Homework 4

Topic: Environment Model and Mutable Data

Lectures: Monday 7/13, Tuesday 7/14, Wednesday 7/15, Thursday 7/16

Reading: Abelson and Sussman: 3.1, 3.2, 3.3

1. (Recommended but not Required) Abelson and Sussman 3.3, 3.4

2. Draw the environmental diagram for the following Scheme expressions and use it to determine what the `(foo 50)` call will return.

```
(define num 1)

(define foo
   (let ((fn (lambda (x) (set! num x))))
      (lambda (num)
          (fn num)
           num)))

(foo 50)
```

Note: You can just turn in an English description of what the final diagram looks like. For example, if we asked you to draw the environmental diagram for

```
(define num 10)

(define (bar fn num)
   (fn num))

(bar (lambda (x) (set! num x)) 1000)
```

You could say:

```
GLOBAL ENV contains:
- num --> 1000
- bar --> *lambda1*
- [*lambda 1*, params: fn num, body: (fn num)]
- [*lambda 2*, params: x, body: (set! num x)]
- E1 ENV
- E2 ENV

E1 ENV contains:
- fn --> *lambda2*
- num --> 1000

E2 ENV contains:
- x --> 1000
```

3. In the following program, we translated an Object-oriented class definition into its simplified regular Scheme equivalent:

```
(define make-student
  (let ((total-students 0))

    (lambda (name)
      (let ((age 12))

        (define (change-name new-name)
           (set! name new-name))

        (define (dispatch m)
           (cond ((eq? m 'name) name)
                 ((eq? m 'change-name) change-name)
                 ((eq? m 'age) age)
                 ((eq? m 'total-students) total-students)
                 ((eq? m 'greet) 'hello)
                 (else ``UNKNOW MESSAGE'')))
        dispatch))))
```

Part A: Draw the environment diagram for the `(define make-student ...)` code above as well as the following expressions:
(The diagram is complicated, but don't give up in frustration! Use the slides from lecture as a guide)

```
> (define jo (make-student 'johann-sebestian))
> ((jo 'change-name) 'john-fitzgerald)
> (define mi (make-student 'michaelangelo))
```

Part B: answer the following questions.

1. Which function is `make-student` bound to? What argument(s) does `make-student` take?
2. What does calling `make-student` return?
3. We know that an Object is actually a procedure, which procedure is it? Why?
4. Is `total-students` a class, instance, or instantiation variable? Why?
5. Is `name` a class, instance, or instantiation variable? Why?
6. Is `age` a class, instance, or instantiation variable? Why?

Part C: Re-write the `make-student` procedure in the syntax of Object Oriented Programming.

Part D: Modify the `make-student` procedure so that every time you make a new student, the `num` class variable would be incremented by 1.

4. Abelson and Sussman, 3.13, 3.14

5. Abelson and Sussman, 3.16, 3.17

6. (Recommended but not Required) 3.21

7. Write a procedure `easy-flatten!` that takes a **list of flat-lists** and uses mutation to turn it into a completely flat list.

For example:

```
> (define a-list '((you just) (keep on trying) (til) (you run out of cake)))
> (easy-flatten! a)
 (you just keep on trying til you run out of cake)
```

Your just need to have your procedure should return the flattened list, you do not need to preserve the original pointer meaning you do not need to worry about what `a-list` looks like after you call your procedure. **DO NOT create any new pairs!** That means, do not use `cons` or any procedure that uses `cons`

8. Write a procedure `flatten!` that takes a deep-list and uses mutation to turn it into a completely flat list. `flatten!` should return the flattened list. You do not need to preserve the original pointer. **DO NOT create any new pairs!** That means, do not use `cons` or any procedure that uses `cons`

```
> (define a-list '((this cake is ((great))) so (delicious (and moist))))
> (flatten! a-list)
 (this cake is great so delicious and moist)
```

Extra for experts:

Meta-programs are programs that reason with other programs. Meta-programs can be incredibly useful; for example, it would be great if we have a program that takes another program as input and determines whether the input program has bugs.

Unfortunately, meta-programs are usually very difficult to write; in fact, Alan Turing proved that it is logically impossible to create powerful meta-programs with his celebrated Turing Halting paradox.

One interesting implication of this is that if computers can eventually be as smart as humans, then the Turing Halting paradox says that humans' abilities to reason with logic is inherently and theoretically limited.

To get a sense of how difficult it is to write meta-programs, write the procedure `cxr-name`. Its argument will be a function made by composing `car`s and `cdr`s. It should return the appropriate name for that function:

```
> (cxr-name (lambda (x) (cadr (cddar (cadar x)))))
CADDDAADAR
```

(credit: this problem is first conceived by Brian Harvey)