MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.037—Structure and Interpretation of Computer Programs
IAP 2018

**Project 2**

Release date: 16 January, 2018
Due date: 18 January, 2018 at 1900h

## Purpose

The purpose of this project is to explore abstract data types and procedures with state. Part of this project deals with mutating objects, using `set-car!` and `set-cdr!`, which require some boilerplate to work in Racket. You can find this boilerplate, as well as an outline of the problems in the project, in the file `memoize.scm`, which can be obtained from the *Projects* link on the course web page. You can also find `table-tests.scm` there as well – more on that below.

Like the previous project, be sure to include test cases and comments in your code! When you are ready to submit your code, save the definitions and **email it to the course staff** at `6.037-psets@mit.edu`.

## Problem 1: A table for later

Let's define an abstraction for a simple key-value table (it will be needed for later questions in this project). It will include a constructor, `make-table`, a mutator `table-put!`, accessors `table-has-key?` and `table-get`, and operators `table?`. Use of the table will look like this:

```
(define my-table (make-table))
(table? my-table) ;; => #t
(table-put! my-table 'ben-bitdiddle 'chocolate) ;; => undefined
(table-put! my-table 'alyssa-p-hacker 'cake) ;; => undefined
(table-has-key? my-table 'ben-bitdiddle) ;; => #t
(table-has-key? my-table 'louis-reasoner) ;; => #f
(table-get my-table 'ben-bitdiddle) ;; => chocolate
(table-get my-table 'louis-reasoner) ;; => ERROR
```

Assume that there won't be a large number of key-value pairs. **What will you use to test for key-equality in `table-get` and `table-has-key`? Why?**

**Write the table abstraction.** You may find the `assoc` procedure useful, although you do not need to use it. It takes a value and a list of pairs and returns the first pair whose `car` is `equal?`-to the value. You may also find the `error` function useful – it takes a string, and produces a run-time error.

We have provided a complete formal test suite for the table abstraction, in the `table-tests.scm` file. We'll talk about these testing functions more next week – but if placed in the same directory as your `memoize.scm` file, running it should run a number of tests against your table implementation. A correct implementation will pass all tests!

These tests are more complete than we expect for your test cases for the rest of the project – mainly because they serve as the primary specification for the table abstraction. You should feel free to write your own test cases in the informal style that we used in the first two projects. As always, they should be well-chosen exemplars that exercise a number of edge cases of each procedure. Show both the input and the result, and if the result is not simply observably correct, mention how you determined the correct value.

Note for debugging: DrRacket will not let you, in its lower Interactions window, `set!` something you `define`d in the upper Definition window.

## Problem 2: lambda-net is monitoring you

Recall the recursive definition for `fib` from the first recitation. Here it is again as a reminder:

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

We saw that this procedure shows *exponential* growth, and worse, it computes the same thing over and over again.

In general, performance problems in computer programs are often caused by excessive numbers of calls to the same procedure. To help track down such programming errors, let's build a procedure that will allow us to track the number of times a procedure is called.

We will do this by building and returning a new procedure that has local state. Specifically, this new procedure will keep a counter which is incremented every time we call the original procedure. We will query and manipulate this counter by passing the new procedure special symbols.

**Write the make-monitored procedure.** It should take a procedure, `f`, of one argument and return a new procedure, `mf`, of one argument. When `mf` is passed the special symbol `how-many-calls?`, it should return the value of its internal counter. When `mf` is passed the special symbol `reset-call-count`, it should reset its internal counter to zero. If `mf` is passed any other value, it should increment its counter and call `f` with its input.

For example:

```
(fib 8) ;; => 21
(set! fib (make-monitored fib))
(fib 8) ;; => 21
(fib 'how-many-calls?) ;; => 67
(fib 8) ;; => 21
(fib 'how-many-calls?) ;; => 134
(fib 'reset-call-count)
(fib 'how-many-calls?) ;; => 0
```

Note that the following will not work as intended:

```
(define fib ...)
(define mon-fib (make-monitored fib))
(mon-fib 8)
(mon-fib 'how-many-calls?) ;; => 1
```

**Why doesn't this correctly record the number of times fib was called?** Think about how the recursive case is handled.

## Problem 3: Back to the table

Let's get a handle on how bad our implementation of `fib` actually is. **Write a procedure, make-num-calls-table,** which given a monitored procedure (a procedure that was returned from `make-monitored`) and a number `max`, returns a new table recording the number of calls the procedure makes for each input between 1 and `max`.

```
> (make-num-calls-table fib 10)
(table (1 1) (2 3) (3 5) (4 9) (5 15) (6 25) (7 41) (8 67) (9 109) (10 177))
```

**How many calls to `fib` are made when you evaluate `(fib 20)`? `(fib 30)`?**

## Problem 4: Remembering the past

Clearly, our `fib` procedure is very inefficient. Ben Bitdiddle comes up with idea of keeping a list of prior results in a table, so that if called upon to compute something that's already been computed, the prior result can simply be returned again without any further work. Ben starts to modify his `fib` procedure to include a key-value table, where the keys are the values of `n` passed in, and the values are the result of computing `(fib n)`.

Alyssa P. Hacker sees this and scolds Ben for not thinking bigger. "Surely this concept could apply to other procedures, not just `fib`, Ben!" She starts writing a procedure `memoize` which takes a procedure of one argument and returns a procedure of one argument. She shows Ben to use it like this:

```
(set! fib (memoize fib))
(fib 8) ;; => 21
```

**Write the `memoize` procedure.** It should take a procedure, `f`, of one argument and return a new procedure, `mf`. The first call to `mf` with a given argument should call `f`, but cache the result in a table. Subsequent calls to `mf` with the same argument should return the cached value instead of redoing the computation.

## Problem 5: (Optional) A word of advice

We can make our monitoring system more generic using **advice**. Advice is a general method of augmenting procedures with additional functionality. Some Lisp systems have facilities of adding advice to procedures globally, allowing add-in libraries to augment system procedures easily.

Our advice system will be relatively simple. It will consist of a pair of procedures of no arguments that are executed before and after a given procedure of one argument.

**Write the `advise` procedure**, which creates a procedure with advice. `advise` takes three procedures as arguments (`func`, `before`, and `after`) and returns a new procedure that, when invoked with an argument, calls `before`, `func` with the argument, and then `after`, finally returning the result of `func`.

```
> (define (add-1 x) (+ x 1))
> (define advised-add-1
    (advise add-1
      (lambda () (displayln "calling add-1"))
      (lambda () (displayln "add-1 done"))))
> (advised-add-1 5)
calling add-1
add-1 done
6
```

## Problem 6: (Optional) Yep, lambda-net is still monitoring you

Now we'll reimplement our monitoring procedure using advice. To mix it up a bit, we'll change the way we access the number of calls made. Instead of passing a symbol to the returned procedure, the new monitoring system should print out the number of calls made to the monitored procedure, but only when the top-most execution of the procedure finishes. That is, the monitored procedure should not print anything out during recursive calls, only the original call the user made.

```
> (set! fib (make-monitored-with-advice fib))
> (fib 10)
Num calls: 177
55
```

**Write** `make-monitored-with-advice`**.**