MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.5150/6.5151 Spring 2024
Problem Set 1

Issued: Wed. 14 Feb. 2024 **Due: Fri. 23 Feb. 2024**

Readings:
- **Review SICP Chapter 1 (especially section 1.3)**
- **Software Design for Flexibility (SDF)**
  - **Chapter 2 (Domain-specific Languages)**
- **MIT/GNU Scheme Reference Manual -- as needed**

Code: function-combinators.scm

Documentation:
The MIT/GNU Scheme installation and documentation can be found online at http://www.gnu.org/software/mit-scheme/ The reference manual and the user's manual are both on that website. You will probably not need the user's manual, but the reference manual is quite essential.

The reading material for this week is foundational. Read Chapter 2 in SDF. Section 2.1 is needed to do this problem set. Then download the code for the problem set from the class website.

Just to repeat what was said for problem set 0:

In general, you need to make a directory (folder, in modern usage) on your computer for the problem set (I would personally call it "~/6.5150/ps01/") and a subdirectory called "code" (so you would have the directory "~/6.5150/ps01/code/". Download the problem set text into "~/6.5150/ps01/" and the code into "~/6.5150/ps01/code/". There will be a file in the code directory named "load.scm". You can load the code you need into the Scheme system by pointing the Scheme at the code directory, with executing (cd "~/6.5150/ps01/code/") and then executing (load "load"). The code files you load will contain some extra material, not in the text, that support the problem set.

Note: Loading of the support files for the problem set by loading "load.scm" reinitializes the top-level environment of Scheme, so you will lose any definitions you have made in that environment. But this will not lose buffers in your EMACS or EDWIN, so your code is not lost. Of course, you should write out the files you are building regularly, probably in your directory "~/6.5150/ps01/".

To Do

Exercise 2.4: As Compositions? (SDF p.36)

Exercise 2.5: Useful Combinators (SDF p.37)

Exercise 2.a: (Not in SDF)
Most modern languages, such as Python and Javascript provide facilities to write and use combinators like COMPOSE. Pick your favorite language and write implementations, in your favorite language, of three of the combinators that that we show in section 2.1 of SDF. Can you deal with multiple arguments? To what extent can you make them work with multiple returned values? To what extent can you put in checking for correct arity? Do these requirements conflict. Demonstrate that your combinators work correctly with a few examples.

Exercise 2.b: (Not in SDF)
The implementation of arities in the combinator library we show in the book and give in the code file function-combinators.scm is not complete. It is not compatible with the MIT/GNU Scheme more general arity structure that allows procedures with unspecified numbers of arguments, like addition. For example, perfectly sensible compositions fail to work:

```
((compose cos +) 3 4)
;Assertion failed:
;(eqv? (procedure-arity-min a) (procedure-arity-max a))
```

This should, of course, be equivalent to:

```
(cos (+ 3 4))
;Value: .7539022543433046
```

Of course, everything works fine if you just remove the arity checking in the given combinators, except for combinators like SPREAD-COMBINE, which needs the arities of the given functions F and G to select argument to spread.

MIT/GNU Scheme arity specifications are more general than what we provide at the end of the file function-combinators.scm. You can find a clear description of the arity abstraction provided by MIT/GNU Scheme in section 12.2 of the Reference Manual:

www.gnu.org/software/mit-scheme/documentation/stable/mit-scheme-ref.pdf

YOUR JOB is to reengineer the arity interface (RESTRICT-ARITY and GET-ARITY in function-combinators.scm) to make everything work as expected. You may use any code specified in the reference manual.

You may have to slightly modify the combinator code to account for the more general arity structure you will need.

Have fun!

BTW, you can always ask the Scheme system how it implements stuff. The PP (Pretty-Print) procedure will try to show you an understandable representation of code in the system. (Some real primitives, such as CAR, CDR, INDEX-FIXNUM?, and LESS-THAN-FIXNUM? are not explorable this way, (they are implemented in a few machine instructions), but you can get the basic algorithms in terms of these real primitives.) For example:

```
(pp procedure-arity-max)
(named-lambda (procedure-arity-max arity)
  (cond ((index-fixnum? arity) arity)
        ((and (pair? arity)
              (index-fixnum? (car arity))
              (if (cdr arity)
                  (and (index-fixnum? (cdr arity))
                       (not (less-than-fixnum? (cdr arity)
                                               (car arity))))
                  #t))
         (cdr arity))
        (else
         (error:not-a procedure-arity?
                      arity
                      'procedure-arity-max))))


(pp procedure-arity-valid?)
(named-lambda (procedure-arity-valid? procedure arity)
  (procedure-arity<= arity (procedure-arity procedure)))

(pp procedure-arity<=)
(named-lambda (procedure-arity<= arity1 arity2)
  (and (not
        (greater-than-fixnum? (procedure-arity-min arity2)
                              (procedure-arity-min arity1)))
       (or (not (procedure-arity-max arity2))
           (and (procedure-arity-max arity1)
                (not
                 (greater-than-fixnum? (procedure-arity-max arity1)
                                       (procedure-arity-max arity2)))))))
```