MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.5150/6.5151—Large-Scale Symbolic Systems
Spring 2024

# Pset 2

Issued: 21 Feb. 2024 Due: 1 Mar. 2024

Reading:

SICP sections 2.4 and 2.5

SDF Chapter 3, Section 3.1

Documentation:

The MIT/GNU Scheme documentation, available online at
`http://www.gnu.org/software/mit-scheme/`

The Software Manager documentation, available online on the class website

Note: This and subsequent problem sets require you to read and understand a substantial amount of code. Please be sure to start early and ask for help if you need guidance.

The code for this problem set is written in a professional style. It is broken up into many files, some of which are quite small. There are many parts that are setups for future use in a larger system (and in future problem sets!). Also, we use MIT/GNU Scheme record structures at the lowest level, so you may need to read about them in the documentation.

The code base for 6.5150/6.5151 is available as a tarball `sdf.tgz` on the class website. You should download it and expand it under an appropriate directory (I suggest `~/6.5150/`). For my laptop the appropriate shell command is:

```
gjs@gjs-x2:~/6.5150$ tar xzf sdf.tgz
```

This will make a subdirectory `~/6.5150/sdf/` that contains the code directories for the entire term. There is a special directory `~/6.5150/sdf/manager` this containing the software manager code, that we will use to manipulate and load software for this class (RTFM!, it is short!). After you start the MIT/GNU Scheme system, tell it to load the software manager with:

```
(load "~/6.5150/sdf/manager/load" )
```

For example, to load the files that are used in this problem set you will say:

```
(manage 'new 'combining-arithmetics)
```

You will see that many files are loaded into the Scheme system from various subdirectories of `sdf`. Files from `common` are used for lots of stuff, whereas files from `combining-arithmetics` are just for this work. The instructions for which files to load are in the `sdf/combining-arithmetics/load-spec` file, which contains the following declaration:

```
(define-load-spec combining-arithmetics
  ("arith" from "common")
  ("numeric-arith" from "common")
  "standard-arith"
  "function-variants"
  ("stormer2" from "common" test-only? #t))
```

So, besides a bunch of stuff that is default loaded from the `common` directory there are two files that are explicitly loaded from `common` and two files that are loaded from the `combining-arithmetics` directory.

You can modify any files in your copy of the `sdf` directory structure, and you can make new ones, which you may add to the appropriate `load-spec` file. Note that any files that you modify in `common` may affect what happens in future problem sets!

In any case, please hand in only your modifications... We do not want to see huge amounts of code that repeats what we have provided.

When creating new arithmetics, you do not need to write n-ary procedures—n-ary procedures should be defined as binary procedures which are then extended to n-ary when they are installed. See `sdf/common/arith.scm` for details on how this works.

One special note: after you execute an `install-arithmetic!` you will have modified the Scheme arithmetic, so ordinary arithmetic may no longer work. You can fix this by `(install-arithmetic! numeric-arithmetic)`. If you are really confused, you can always do a reload by executing `(manage 'new 'combining-arithmetics)`, which will get you an entirely new environment, in the state described by your `load-spec`. This will lose the broken environment and all definitions made in that environment. If you are using EDWIN this has no effect on your EDWIN buffers. In general, it is almost never necessary to get a new Scheme system: it is a very tough system!

A note about workflow: If you are using Edwin (or running under Emacs) it is almost never to your advantage to leave the Emacs or Edwin and start another one. Scheme and Emacs are interactive systems. We do not need to go through the edit-compile-runprogram-debug loop that is common with languages like C. When your Scheme program has a bug you can examine the stack and evironment in Scheme. You can edit your code to fix the bug. You can then clear the bug state with ctrl-C ctrl-C, and re-execute the parts of your code to effect the patch and retry your example. This is the most efficient way to work in Scheme under Emacs or Edwin.

## To Do

Exercise 3.1: Warmup with Boolean Arithmetic (SDF p. 84)

Exercise 3.2: Vectors (SDF p. 85–86)

Exercise 3.3: Ordering of Extensions (SDF p. 86–87)

Exercise 3.a: Again, to what extent is this kind of expansion of the power of arithmetic possible in your favorite non-Lisp language? Can you overload simple operators, like + and *? Can you overload arbitrary functions, like sin and sqrt? What is hard and what is easy? Pick your favorite language and show examples of how to do what can be done easily and what is hard. For example, in python there are "dunder methods" for some operators. Is there a moral to this story?