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

Pset 3

Issued: 28 February 2024 Due: 8 March 2024

Reading: SICP sections 2.4 and 2.5 (Tagged data, Data-directed programming, Generic Operations)

SDF in Chapter 3, Section 3.2 (Extensible generic procedures), and Section 3.4 (Efficient generic procedures)

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

To Do

First `(manage 'new 'generic-procedures)` to work with generic procdures.

This will be needed for calling `make-generic-arithmetic` when making a generic arithmetic to install, as in:

```
(define full-generic-arithmetic
  (let ((g (make-generic-arithmetic make-simple-dispatch-store)))
    (add-to-generic-arithmetic! g numeric-arithmetic)
    (extend-generic-arithmetic! g function-extender)
    (add-to-generic-arithmetic! g
            (symbolic-extender numeric-arithmetic))
    ;; YOU MAY ADD STUFF HERE
    g))

;;; To allow use of assign-handler! do the following:

(install-arithmetic! full-generic-arithmetic)

;;; This makes the default top-level arithmetic
;;;  the full-generic-arithmetic

;;; You can use assign-handler! to add handlers.
;;; For example, to enable + to append strings you may write:
;;; (assign-handler! + string-append string? string?)

;;; And now we see that
;;; (+ "foo" "bar")
;;; ;Value: "foobar"
```

Exercise 3.4: Functional Values (SDF p. 101)

Exercise 3.6: Matrices (SDF pp. 102–103)

Now we want to deal with automatic diffferentiation so we bring that into the system, so we must: `(manage 'add 'automatic-differentiation)`

Exercise 3.8: Partial Derivatives (SDF p. 113)

Next, let's do some experiments with efficiency.

We want an arithmetic based on a simple stupid dispatch store as a baseline, so we need:

```
(define full-generic-arithmetic
  (let ((g (make-generic-arithmetic make-simple-dispatch-store)))
    (add-to-generic-arithmetic! g numeric-arithmetic)
    (extend-generic-arithmetic! g function-extender)
    (add-to-generic-arithmetic! g
            (symbolic-extender numeric-arithmetic))
    g))
```

To work with the trie version you must incant:

```
(define trie-full-generic-arithmetic
  (let ((g (make-generic-arithmetic make-trie-dispatch-store)))
    (add-to-generic-arithmetic! g numeric-arithmetic)
    (extend-generic-arithmetic! g function-extender)
    (add-to-generic-arithmetic! g
       (symbolic-extender numeric-arithmetic))
    g))
```

And to work with a cached dispatch store we need to define one:

```
(define (make-cached-trie-dispatch-store)
  (cache-wrapped-dispatch-store (make-trie-dispatch-store)
                                implementation-type-name))

(define cached-trie-full-generic-arithmetic
  (let ((g (make-generic-arithmetic make-cached-trie-dispatch-store)))
    (add-to-generic-arithmetic! g numeric-arithmetic)
    (extend-generic-arithmetic! g function-extender)
    (add-to-generic-arithmetic! g
       (symbolic-extender numeric-arithmetic))
    g))
```

After defining these we can switch between them by installing any of these arithmetics with `install-arithmetic!`.

This is what it takes to setup for the following exercises.

Exercise 3.14: Dispatch efficiency: Gotcha! (SDF p. 130)

Exercise 3.15: Cache Performance (SDF p. 132)