MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.945/6.905—Large-Scale Symbolic Systems
Spring 2017

Pset 3

Issued: 1 March 2017 Due: 10 March 2017

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

Code: `load.scm`, `utils.scm`, `collections.scm`, `predicates.scm`, `predicate-metadata.scm`, `predicate-counte`
`applicability.scm`, `generic-procedures.scm`, `operators.scm`, `operations.scm`, `package.scm`,
`arith.scm`, `numeric-arith.scm`, `standard-arith.scm`, `function-variants.scm`, `generics.scm`,
`generic-arith.scm` `stormer2.scm`, `examples.scm`

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

Note: This problem set builds on the previous problem set. Please be sure to start early and ask for help if you need guidance. The software has been a bit modified since the previous problem set, but it is basically the same, except for the new files `generic-procedures.scm`, `generics.scm`, and `generic-arith.scm`.

The text that follows is a summary of the material that supports this problem set.

Extensible Generic Procedures

Systems built with combinators result in beautiful diamond-like systems. This is sometimes the right idea, and we will see it arise again, but it is very hard to add a discordant feature to a diamond. But if a system is built as a ball of mud it is easy to add more mud.

One organization for a ball of mud is a system erected on a substrate of extensible generic operators. Modern dynamically typed programming languages, such as Lisp, Scheme, and Python, usually have built-in arithmetic that is generic over a variety of types of numerical quantities, such as integers, floats, rationals, and complex numbers.[8, 4, 6] But systems built on these languages are usually not easily extensible after the fact.

Let's explore how to erect a general arithmetic using the parts we have already defined, but using user-extensible generic procedures as a substrate.[1]

The previously displayed mechanism extends arithmetic by accreting layers. The final layer that we install shadows the sublayers. By contrast, we may want to describe a generic procedure as a set of rules, each of which describes a handler that is appropriate for a given set of arguments. Such a rule is an applicability predicate and a handler.

Let's examine how this might work, by defining a generic operator `foo` which works like addition with numeric and symbolic quantities:

[1] A mechanism of this sort is implicit in most "object-oriented languages," but it is usually tightly bound to ontological mechanisms such as inheritance. The essential idea of extensible generics appears in SICP [1] and is usefully provided in tinyCLOS [5] and SOS [3]. A system of extensible generics, based on predicate dispatching, is used to implement the mathematical representation system in SICM [9]. A nice exposition of predicate dispatching is given by Ernst [2].

```
(define foo (simple-generic-procedure 'foo 2))

(define-generic-procedure-handler foo
  (all-args 2 number?)
  (lambda (a b) (+ a b)))

(define-generic-procedure-handler foo
  (any-arg 2 symbol? number?)
  (lambda (a b) (list '+ a b)))


(foo 1 2)
3


(foo 1 'a)
(+ 1 a)


(foo 'a 2)
(+ a 2)


(foo 'a 'b)
(+ a b)
```

The procedure `simple-generic-procedure` takes two arguments: The first is an arbitrary name to identify the operator when debugging; the second is the operator's arity. It returns a new generic procedure that can be assigned to the operator symbol. The returned generic procedure is implemented as a Scheme procedure that can be called with the specified number of arguments. The procedure `define-generic-procedure-handler` is used to add a rule to an existing generic procedure. Its first argument is the generic procedure to be extended; the second argument is an applicability specification for the rule being added; and the third argument is the handler for arguments that satisfy that applicability.

In this example we specify the applicability of a handler by using patterns of predicates each of which is applied to an argument. One such pattern is `(any-arg 2 symbol? number?)`. It is often necessary to specify a rule where different arguments are of different classes. For example, to make a vector arithmetic package we need to specify the interpretation of the `*` operator. If both arguments are vectors, the appropriate handler computes the dot product. If one argument is a scalar and the other is a vector then the appropriate handler scales the vector elements by the scalar.

Here is one way to implement a generic procedure:

```
(define (make-generic-procedure name arity dispatcher)
  (let* ((metadata
          (make-generic-metadata name
                                 arity
                                 (dispatcher)
                                 (get-default-handler name)))
         (procedure
          (lambda args
            (generic-procedure-dispatch metadata args))))
    (set-generic-procedure-metadata! procedure metadata)
    procedure))
```

This implementation uses an ordinary Scheme procedure to represent the generic procedure, and a metadata record (for stored rules, etc.) that determines the procedure's behavior. This record is associated with the generic procedure by storing it in a metadata table, so that `define-generic-procedure-handler` for example, can find it given the generic procedure.

The `dispatcher` argument to `make-generic-procedure` is a procedure that creates a structure for saving and retrieving handlers, and is called a "dispatch strategy". By passing this as an argument, the strategy can be chosen when the generic procedure is constructed, rather than hard-coding it into the implementation. For now we will use a simpler constructor:[2]

```
(define (simple-generic-procedure name arity)
  (make-generic-procedure name arity simple-generic-dispatcher))
```

The `define-generic-procedure-handler` procedure is also fairly simple, using the metadata table to get the metadata record for the generic procedure, creating a rule from the applicability predicate and handler, and adding that rule to the metadata.

```
(define (define-generic-procedure-handler proc
                                          applicability
                                          handler)
  (((generic-metadata-dispatcher
     (generic-procedure-metadata proc))
    'add-handler!)
   applicability
   handler))
```

Finally, the heart of the mechanism is the dispatch, which finds an appropriate handler and applies it:

```
(define (generic-procedure-dispatch metadata args)
  (let ((handler
         (get-generic-procedure-handler metadata args)))
    (if trace-generic-dispatch? ; Turn on for debugging
        (trace-generic-dispatch metadata args handler))
    (apply handler args)))
```

[2] The `simple-generic-dispatcher`, not shown here—see code file `generic-procedures.scm`, just maintains a list of the rules each of which pairs an applicability with a handler. The handler definition procedure puts new rules on the front of the list. The dispatcher scans the list sequentially for a handler whose applicability is satisfied by the arguments tendered. There are many ways to construct a dispatcher, from simple ones such as this to ones that run all acceptable handlers in parallel and compare the results!

```
(define (get-generic-procedure-handler metadata args)
  (or ((generic-metadata-getter metadata) args)
      ((generic-metadata-default-getter metadata))))
```

The default handler, which is called if the `generic-metadata-getter` cannot find a handler appropriate for the arguments, just signals an error. The default handler can be replaced, if desired. (See code file `generic-procedures.scm`.)

```
(define (get-default-handler name)
  (lambda args
    (error "Inapplicable generic procedure:" name args)))
```

Let's use this new generic-procedure mechanism in place of the heirarchical combinations used earlier:

```
(define (make-generic-arithmetic dispatcher)
  (make-arithmetic 'generic any-object? '()
    (lambda (name)
      (constant-union name))
    (lambda (operator)
      (simple-operation operator
                        any-object?
                        (make-generic-procedure
                          operator
                          (operator-arity operator)
                          dispatcher)))))
```

The `make-generic-arithmetic` procedure creates a new arithmetic. For each operator in the generic arithmetic an operation must be constructed. But here for each operator we supply a generic procedure as the implementation of an operation that is applicable to any arguments. We can install this arithmetic in the usual way.

But first, we need to define some handlers for the generic procedures. It's pretty simple to do now that we have the generic arithmetic object:

```
(define (add-to-generic-arithmetic! generic-arithmetic
                                    arithmetic)
  (for-each
   (lambda (name)
     (let ((binding
            (arithmetic-constant-binding
                name generic-arithmetic))
           (element
            (find-arithmetic-constant name arithmetic)))
       (set-cdr! binding
                 (constant-union name
                                 (cdr binding)
                                 element))))
   (arithmetic-constant-names generic-arithmetic))
  (for-each
   (lambda (operator)
     (let ((generic-procedure
            (simple-operation-procedure
             (arithmetic-operation operator
                                   generic-arithmetic))))
       (for-each (lambda (operation)
                   (define-generic-procedure-handler
```

```
                    generic-procedure
                    (operation-applicability operation)
                    (operation-procedure operation)))
                (operation-components
                 (arithmetic-operation operator
                                       arithmetic)))))
  (arithmetic-operators arithmetic)))
```

This takes a generic arithmetic package and an ordinary arithmetic package with the same operators. It adds a rule from each operator of the given arithmetic to the corresponding generic procedure. The code for making generic arithmetics can be found in code file `generic-arith.scm`.

We can add many arithmetics to a generic arithmetic to give it interesting behavior:

```
(let ((g (make-generic-arithmetic simple-generic-dispatcher)))
  (add-to-generic-arithmetic! g numeric-arithmetic)
  (add-to-generic-arithmetic! g
    (function-arithmetic numeric-arithmetic))
  (add-to-generic-arithmetic! g
    (symbolic-arithmetic numeric-arithmetic))
  (install-arithmetic! g))
```

This produces a generic arithmetic that combines numeric arithmetic with symbolic arithmetic over numeric arithmetic and function arithmetic over numeric arithmetic:

```
(+ 1 'a 3)
(+ (+ 1 a) 3)
```

And we can even run some more complex problems:

```
(pp (x 0 ((evolver F 'h stormer-2) numeric-s0 1)))
(+ 9.999833334166664e-3
   (* (/ (expt h 2) 12)
      -9.999750002487318e-7))
```

We can mix symbols and functions:

```
(+ 'a ((+ cos sin) 3))
(+ a -.8488724885405782)
```

but we can't apply functions to symbolic quantities because the symbolic quantities are not in the codomain arithmetic for the function arithmetic, so the following will not work:

```
(+ 'a ((+ cos sin) 'b))
```

But there is magic in this formulation: generic arithmetic can be closed in that all extensions can be made over the generic arithmetic!

```
(let ((g (make-generic-arithmetic simple-generic-dispatcher)))
  (add-to-generic-arithmetic! g numeric-arithmetic)
  (extend-generic-arithmetic! g symbolic-extender)
  (extend-generic-arithmetic! g function-extender)
  (install-arithmetic! g))
```

This uses a new procedure `extend-generic-arithmetic!` that captures a common pattern.

```
(define (extend-generic-arithmetic! generic-arithmetic extension)
  (add-to-generic-arithmetic! generic-arithmetic
                              (extension generic-arithmetic)))
```

Now we can use complex mixed expressions

```
(+ 'a ((+ 'c cos sin) (* 2 'b)))
(+ a (+ (+ c (cos (* 2 b))) (sin (* 2 b))))
```

or even functions that return functions:

```
(((+ (lambda (x) (lambda (y) (cons x y)))
     (lambda (x) (lambda (y) (cons y x))))
  3)
 4)
(+ (3 . 4) (4 . 3))
```

So perhaps we have achieved nirvana?

Not so fast!

Unfortunately, there is a severe dependence on the order in which rules are added to the generic procedures. This is not surprising, because the construction of the generic procedure system is by assignment. We can see this by changing the order of construction:

```
(let ((g (make-generic-arithmetic simple-generic-dispatcher)))
  (add-to-generic-arithmetic! g numeric-arithmetic)
  (extend-generic-arithmetic! g function-arithmetic)   ;*
  (extend-generic-arithmetic! g symbolic-arithmetic)   ;*
  (install-arithmetic! g))
```

and then we will find that this example fails

```
(+ 'a ((+ 'c cos sin) (* 2 'b)))
```

because the symbolic arithmetic captures the `(+ 'c cos sin)` to produce the symbolic expression, which is not a function that can be applied to `(* 2 'b)`. This requires a coercion from the symbolic form to a functional form; for now we can use `literal-function` as a workaround.

One way to resolve this problem is to restrict the symbolic quantities to represent numbers. We can do this by building our generic arithmetic so that the symbolic arithmetic is over the numeric arithmetic, as we did on page 6, rather than over the entire generic arithmetic:

```
(let ((g (make-generic-arithmetic simple-generic-dispatcher)))
  (add-to-generic-arithmetic! g numeric-arithmetic)
  (extend-generic-arithmetic! g function-arithmetic)
  (add-to-generic-arithmetic! g
      (symbolic-arithmetic numeric-arithmetic))
  (install-arithmetic! g))
```

Now this works, independently of the ordering, because there are no ambiguous rules.

```
(+ 'a ((+ 'c cos sin) (* 2 'b)))
(+ a (+ (+ c (cos (* 2 b))) (sin (* 2 b))))
```

Unfortunately, we may want to have symbolic expressions over other quantities besides numbers. We cannot yet implement a general solution to this problem, but we can continue to use `literal-function` as we did earlier:

```
(+ 'a ((+ (literal-function 'c) cos sin) (* 2 'b)))
(+ a (+ (+ (c (* 2 b)) (cos (* 2 b))) (sin (* 2 b))))
```

This will work independent of the order of construction of the generic arithmetic.

With this mechanism we are now in a position to evaluate the Stormer integrator with a literal acceleration function:

```
(pp (x 0 ((evolver (literal-function 'F) 'h stormer-2)
          (make-initial-history 't 'h 'xt 'xt-h 'xt-2h)
          1))
(+ (+ (* 2 xt) (* -1 xt-h))
   (* (/ (expt h 2) 12)
      (+ (+ (* 13 (f t xt))
            (* -2 (f (- t h) xt-h)))
         (f (- t (* 2 h)) xt-2h))))
```

This is pretty ugly, and it would be worse if we looked at the output of two integration steps. But it is interesting to look at the result of simplifying a two-step integration. Using a magic symbolic-expression simplifier we get a pretty readable expression. This can be useful for debugging a numerical process.

```
(+ (* 2 (expt h 2) (f t xt))
   (* -1/4 (expt h 2) (f (+ (* -1 h) t) xt-h))
   (* 1/6 (expt h 2) (f (+ (* -2 h) t) xt-2h))
   (* 13/12
      (expt h 2)
      (f (+ h t)
         (+ (* 13/12 (expt h 2) (f t xt))
            (* -1/6 (expt h 2) (f (+ (* -1 h) t) xt-h))
            (* 1/12 (expt h 2) (f (+ (* -2 h) t) xt-2h))
            (* 2 xt)
            (* -1 xt-h))))
   (* 3 xt)
   (* -2 xt-h))
```

Notice, for example that there are only four top-level calls to the acceleration function `f`. The second argument to the fourth call top-level call uses three calls to `f` that have already been computed. If we eliminate common subexpressions we get:

```
(let* ((G84 (expt h 2)) (G85 (f t xt)) (G87 (* -1 h))
       (G88 (+ G87 t)) (G89 (f G88 xt-h)) (G91 (* -2 h))
       (G92 (+ G91 t)) (G93 (f G92 xt-2h)))
  (+ (* 2 G84 G85)
     (* -1/4 G84 G89)
     (* 1/6 G84 G93)
     (* 13/12 G84
        (f (+ h t)
           (+ (* 13/12 G84 G85)
              (* -1/6 G84 G89)
              (* 1/12 G84 G93)
              (* 2 xt)
              (* -1 xt-h))))
     (* 3 xt)
     (* -2 xt-h)))
```

Here we clearly see that there are only four calls to `f`. However, each integration step in the basic integrator makes three calls to `f` but the two steps overlap on two intermediate calls. While this is obvious for such a simple example, we see how symbolic evaluation might be helpful in understanding a numerical computation.

Problem 3.1: Functional Values

The generic arithmetic structure allows us to close the system so that functions that return functions can work, as in the example

```
(((* 3
     (lambda (x) (lambda (y) (+ x y)))
     (lambda (x) (lambda (y) (vector y x))))
  'a)
 4)
(* (* 3 (+ a 4)) #(4 a))
```

a. How hard is it to arrange for this to work in the purely combinator-based arithmetic introduced previously? Why?

b. In the last problem set we asked about the implications of ordering of symbolic and functional extensions. Is the generic system able to support both expressions discussed there (and copied below)? Explain.

```
((magnitude unit-circle) 'a)
((magnitude (vector sin cos)) 'a)
```

c. Is there any good way to make the following work at all?

```
((vector cos sin) 3)
#(-.9899924966004454 .1411200080598672)
```

Show code that makes this work or explain the difficulties.

Problem 3.2: A Weird Bug

Consider the `+-like` procedure in `arith.scm`. It seems that it is written to execute the identity procedure every time the operator is called with no arguments.

```
(define (+-like operator identity-name)
  (lambda (arithmetic)
    (let ((binary
           (operation-procedure
            (arithmetic-operation operator arithmetic)))
          (get-identity
           (identity-name->getter identity-name arithmetic)))
      (cons operator
            (lambda args
              (case (length args)
                ((0) (get-identity))
                ((1) (car args))
                (else (reduce-left binary #f args))))))))
```

It seems that the identity for that operator should be computed only once. As a consequence it is proposed that the code should be modified as follows:

```
(define (+-like operator identity-name)
  (lambda (arithmetic)
    (let ((binary
           (operation-procedure
            (arithmetic-operation operator arithmetic)))
          (identity
           ((identity-name->getter identity-name arithmetic))))
      (cons operator
            (lambda args
              (case (length args)
                ((0) identity)
                ((1) (car args))
                (else (reduce-left binary #f args))))))))
```

However, this has a subtle bug. Can you elicit the bug? Can you explain it!

Problem 3.3: Matrices

(Optional—This is a pretty big project. Peerhaps combined with Problem 3.4 below this would be part of a nice term project for a group interested in algebra.)

a. Make and install a package for arithmetic on matrices of numbers. This package needs to be able to know the number of rows and the number of columns in a matrix, since matrix multiplication is defined only if the number of columns in the first matrix is the number of rows in the second one. For matrices to play well with vectors you probably need to distinguish row vectors and column vectors. How does this affect the design of the vector package?

You may assume that the vectors and matrices are of small dimension, so you do not need to deal with sparse representations. A reasonable representation of a matrix is a Scheme vector with each element a Scheme vector representing a row.

b. Vectors and matrices may contain symbolic numerical quantities. Make this work.

c. Matrix inversion is appropriate for your arithmetic. For numerical matrices there is a huge choice of methods.

Note that there is not much choice in inverting matrices with symbolic entries. If a symbolic matrix is dense the inverse may take space that is factorial in the dimension. Why?

Problem 3.4: Literal Vectors and Matrices

(Optional—This, combined with Problem 3.3 above is more of a project than a problem-set problem!)

It is also possible to have arithmetic on literal matrices and literal vectors with an algebra of symbolic expressions of vectors and matrices. Can you make symbolic algebra of these compound structures play well with vectors and matrices with symbolic numerical expressions as elements? Caution: 3.4 is quite hard. Perhaps it is part of an appropriate term project.

Efficient generic operations

In section we dispatch to a handler by finding an applicable rule using the dispatcher provided in the metadata:

```
(define (generic-procedure-dispatch metadata args)
  (let ((handler (get-generic-procedure-handler metadata args)))
    (if trace-generic-dispatch?
        (trace-generic-dispatch metadata args handler))
    (apply handler args)))
```

The implementation we used was rather crude. The simple dispatcher stores the rule set as a list of rules. Each rule is represented as a pair of an applicability and a handler. The applicability is a set of lists of predicates to apply to tendered arguments. The way we find an appropriate handler is to iterate over the rules looking for an applicability that is satisfied by the arguments:

```
(define (simple-generic-dispatcher)
  (let ((rules '())
        (default-handler #f))
    (define (get-handler args)
      (let ((rule
             (find (lambda (rule)
                     (predicates-match? (car rule) args))
                   rules)))
        (and rule (cdr rule))))
    (define (add-handler! applicability handler)
      (for-each (lambda (predicates)
                  (let ((p (assoc predicates rules)))
                    (if p
                        (set-cdr! p handler)
                        (set! rules
                              (cons (cons predicates handler)
                                    rules)))))
                applicability))
    (define (get-default-handler)
      default-handler)
    (define (set-default-handler! handler)
      (set! default-handler handler))
    (lambda (operator)
      (case operator
        ((get-handler) get-handler)
        ((add-handler!) add-handler!)
        ((get-default-handler) get-default-handler)
        ((set-default-handler!) set-default-handler!)
        ((get-rules) (lambda () rules))
        (else (error "Unknown operator:" operator))))))
```

This is seriously inefficient, because the applicability of many rules may have the same predicate in a given operand position: For example, for multiplication in a system of numerical and symbolic arithmetic there may be many rules whose first predicate is `number?`. So the `number?` predicate may be applied many times before finding an applicable rule. What is needed is a way of organizing the rules so that finding an applicable one does not perform redundant tests. This is usually accomplished by the use of an index.

Tries

One simple index mechanism is based on the *trie*.[3]

A trie is traditionally a tree structure, but more generally it may be a directed graph. Each node in the trie has edges connecting to successor nodes. Each edge has an associated predicate. The data being tested is a linear sequence of features.[4]

Starting at the root of the trie, the first feature is taken from the sequence and is tested by each predicate on an edge emanating from the root node. The successful predicate's edge is followed to the next node, and the process repeats with the remainder of the sequence of features. When we run out of features, the current node will contain the appropriate handler.

It is possible that at any node, more than one predicate may succeed. If this happens, then all of the successful branches must be followed. Consequently there may be multiple applicable handlers and there must be a separate means of deciding what to do.

Here is how we can use a trie. We can make a trie:

```
(define a-trie (make-trie))
```

We can add an edge to this trie

```
(define s (add-edge-to-trie a-trie symbol?))
```

where `add-edge-to-trie` will return the node that is at the target end of the new edge. When matching, this node is reached by being matched against a symbol.

We can make chains of edges, which are referenced by lists of the corresponding edge predicates

```
(define sn (add-edge-to-trie s number?))
```

Now node `sn` is reached from the root via the path `(list symbol? number?)`. Using a path, there's a simpler way to make a chain of edges:

```
(define ss (intern-path-trie a-trie (list symbol? symbol?)))
```

We can add a value to any node (here we show symbolic values, but we will later store values that are procedural handlers):

```
(trie-has-value? sn)
#f

(set-trie-value! sn '(symbol number))

(trie-has-value? sn)
#t

(trie-value sn)
(symbol number)
```

We can also use a path-based interface to set values

[3] A trie is a clever data structure invented by Edward Fredkin in the early 1960's.

[4] The code for our trie implementation is in the file `trie.scm`.

```
(set-path-value! a-trie (list symbol? symbol?) '(symbol symbol))

(trie-value ss)
(symbol symbol)
```

Note that both `intern-path-trie` and `set-path-value!` reuse existing nodes and edges when possible.

Now we can match a feature sequence against the trie we have constructed:

```
(equal? (list ss) (get-matching-tries a-trie '(a b)))
#t


(equal? (list s) (get-matching-tries a-trie '(c)))
#t
```

We can also combine matching with value fetching. The procedure `get-a-value` finds all matching nodes, picks one that has a value, and returns that value.

```
(get-a-value a-trie '(a b))
(symbol symbol)
```

But not all feature sequences have an associated value

```
(get-a-value a-trie '(-4))
;Unable to match features: (-4)
```

We can incrementally add values to nodes in the trie:

```
(set-path-value! a-trie (list negative-number?)
                 '(negative-number))
(set-path-value! a-trie (list even-number?) '(even-number))

(get-all-values a-trie '(-4))
((even-number) (negative-number))
```

where `get-all-values` finds all the nodes matching a given feature sequence and returns their values.

Problem 3.5: Trie rules

Remember that `make-generic-procedure` and `make-generic-arithmetic` take the dispatcher as an argument. This makes it easy to experiment with different dispatchers. For example, we may build a full generic arithmetic using `trie-generic-dispatcher` as follows:

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

(install-arithmetic! trie-full-generic-arithmetic)
```

a. Does this make any change to the dependence on order that we wrestled with in section ?

b. What characteristics of the predicates will produce situations where there are more than one appropriate handler for a sequence of arguments?

c. Are there any such situations in the generic arithmetic code we've written?

Problem 3.6: Redundant paths

Since some predicates are produced by combining others, specifically `any-arg` disjoins predicates, we may have two predicates that compute the same function that are not `eqv?`. This can produce redundant equivalent edges in the trie. We can reduce this problem by making predicate combinators like `disjoin`, `conjoin`, and `negate` memoize their results thus maintaining more cases of `eqv?`. Explain the essence of this problem in a short paragraph. Implement the strategy. There is code that is useful for this problem in `memoizers.scm`.

Measurement

We have provided a crude tool to measure the effectiveness of our dispatch strategy. By wrapping any computation with `with-predicate-counts` we can find out how many times each dispatch predicate is called in an execution. For example, assuming you have implemented the memoization strategy of exercise 3.6, evaluating `(fib 20)` in a generic arithmetic with trie-based dispatch may yield something like this:

```
(with-predicate-counts (lambda () (fib 20)))
(109453 function?)
(109453 number?)
(109453 symbolic?)
(54727 (disjoin number? symbolic?))
(54727 (disjoin any-object? function?))
6765
```

Problem 3.7: Gotcha!

Given this performance tool it is instructive to look at executions of

```
(define (test-stormer-counts)
  (define (F t x) (- x))
  (define numeric-s0
    (make-initial-history 0 .01 (sin 0) (sin -.01) (sin -.02)))
  (with-predicate-counts
   (lambda ()
     (x 0 ((evolver F 'h stormer-2) numeric-s0 1)))))
```

for the original rule-list–based implementation of dispatch, in the arithmetic you get by:

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

```
    g))

(install-arithmetic! full-generic-arithmetic)
```

and the trie-based version, in the arithmetic you get by:

```
(install-arithmetic! trie-full-generic-arithmetic)
```

You will find that the trie-based dispatch performs slightly worse than the rule-based version. Why?

Understanding this is very important, because it appears counterintuitive. We explicitly introduced the trie to avoid redundant calls. Explain this phenomenon in a concise paragraph.

For an additional insight look at the performance of `(fib 20)` in the two implementations.

By the way, for other problems the trie produces better performance. The performance will be better with the trie if we have a large number of rules with the same initial segment.

References

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, 2nd edition, MIT Press, ISBN 0-262-01553-0, (1996).

[2] M. D. Ernst, C. Kaplan, and C. Chambers. "Predicate Dispatching: A Unified Theory of Dispatch," In ECOOP'98. LNCS, vol. 1445. Springer, Berlin, 186–211 (1998).

[3] Chris Hanson, `SOS` software: Scheme Object System, (1993).

[4] Richard Kelsey, William Clinger, and Jonathan Rees (editors), *Revised^5 Report on the Algorithmic Language Scheme*, (1998).

[5] Gregor Kiczales, `tinyCLOS` software: Kernelized CLOS, with a metaobject protocol, (1992).

[6] Guido van Rossum, and Fred L. Drake, Jr. (Editor); *The Python Language Reference Manual*, Network Theory Ltd, ISBN 0954161785, (September 2003).

[7] http://www.python.org/dev/peps/pep-0020/

[8] Guy L. Steele Jr.; Common Lisp the language, The Digital Equipment Corporation, (1990).

[9] Gerald Jay Sussman and Jack Wisdom with Meinhard E. Mayer, *Structure and Interpretation of Classical Mechanics*, MIT Press, ISBN 0-262-019455-4, (2001).