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

Pset 2

Issued: 22 February 2017 Due: 3 March 2017

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

Code: `load.scm`, `applicability.scm`, `arith.scm`, `examples.scm`, `function-variants.scm`, `numeric-arith.scm`, `operations.scm`, `operators.scm`, `package.scm`, `predicate-counter.scm`, `predicate-metadata.scm`, `predicates.scm`, `standard-arith.scm`, `stormer2.scm`, `utils.scm`

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

Note: This problem set requires 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 text that follows is a summary of the material that supports this problem set.

Arithmetic combinators

Combinators are very powerful and useful, but a system built of combinators is not extremely flexible. The problem is that the shapes of the parts must be worked out ahead of time: all of the generality that will be available depends on the detailed plan for the shapes of the parts.

The behavior of any part of a combinator system must be independent of the context. Additionally, self reference is cumbersome to arrange. We will show how combinators can get us into trouble, and one way of avoiding these problems.

A powerful source of flexibility that is available to a designer is to build systems that *do* depend upon their context. By varying the context of a system we can obtain variation of the behavior. This is quite dangerous, because it may be hard to predict how a variation will behave. However carefully controlled variations can be useful.

Suppose we have a numerical program that computes some useful numerical results. It depends on the meanings of the arithmetic operators that are referenced by the program text. These operators can be extended to work on things other than the numbers that were expected by the program. With these extensions the program may do useful things that were not anticipated when the program was written. A common pattern is a program that takes numerical weights and other arguments and makes a linear combination by adding up the weighted arguments. If the addition and multiplication operators are extended to operate on tuples of numbers as well as the original numbers, the program can now make linear combinations of vectors. This kind of extension can work because the set of arithmetic operators is a well-specified and coherent entity. Extensions of numerical programs with more powerful arithmetic can work, unless the new quantities do not obey the constraints that were assumed by the author of the program. For example, multiplication of matrices does not commute, so extension of a numerical program that depended on the fact that multiplication of numbers is commutative will not work. We will ignore this problem for now.

A simple ODE integrator

Let's investigate the generality of numerical operations in an explicit numerical integrator for second-order differential equations.

$$D^2x(t) = F(t, x(t))$$

The essential idea is that a discrete approximation to the second derivative of the unknown function is a linear combination of second derivatives of some previous steps. The particular coefficients are chosen by numerical analysts and are not of interest here.

$$\frac{x(t+h) - 2x(t) + x(t-h)}{h^2} = \sum_{j=0}^{k} A(j)F(t - jh, x(t - jh))$$

where h is the step size and A is the array of magic coefficients.

For example, Stormer's integrator of second order is

$$\begin{aligned} & x(t+h) - 2x(t) + x(t-h) \\ = \quad & \frac{h^2}{12}(13F(t, x(t)) - 2F(t-h, x(t-h)) + F(t-2h, x(t-2h))) \end{aligned} \tag{1}$$

To use this to compute the future of x we write a program. The procedure returned by `stormer-2`, given a history of states of the dynamical system, produces the value of x at the next state. The procedures `t` and `x` extract previous times and values of x from the history. For example, `(x 0 history)` returns $x(t)$, `(x 1 history)` returns $x(t-h)$, and `(x 2 history)` returns $x(t-2h)$.

```
(define (stormer-2 F h)
  (lambda (history)
    (+ (* 2 (x 0 history))
       (* -1 (x 1 history))
       (* (/ (expt h 2) 12)
          (+ (* 13 (F (t 0 history) (x 0 history)))
             (* -2 (F (t 1 history) (x 1 history)))
             (F (t 2 history) (x 2 history)))))))
```

The `stepper` procedure gives a new history advanced by h for the given integrator.

```
(define (stepper h integrator)
  (lambda (history)
    (extend-history (+ (t 0 history) h)
                    (integrator history)
                    history)))
```

The procedure `evolver` produces a procedure `evolve` that advances the history by a given number of steps of size h. We explicitly use specialized integer arithmetic here (the procedures named `n:>` and `n:-`) for counting steps. This will allow us to use different types of arithmetic for everything else without affecting simple counting.

```
(define (evolver F h make-integrator)
  (let ((integrator (make-integrator F h)))
    (let ((step (stepper h integrator)))
      (define (evolve history n-steps)
        (if (n:> n-steps 0)
            (evolve (step history) (n:- n-steps 1))
            history))
      evolve)))
```

For example, we can integrate the differential equation $D^2x(t) + x(t) = 0$, which is represented as a function:

```
(define (F t x) (- x))
```

We need three initial history values for the second-order integrator (but the equation needs only two!) so we construct an initial history. Since we know that this equation evolves the sine function we use three initial values of the sine function:

```
(define s0
  (make-initial-history 0 .01 (sin 0) (sin -.01) (sin -.02)))
```

Using Scheme's built in arithmetic, after 100 steps of size $h = .01$ we get a good approximation to $\sin(1)$

```
(x 0 ((evolver F .01 stormer-2) s0 100))
.8414709493275624
```

The code that supports this integration example is in the code file `stormer2.scm`.

Symbolic arithmetic

Let's consider the possibility of redefining what is meant by addition, multiplication, etc., for new datatypes unimagined by our example's programmer. Suppose we change our arithmetic operators to operate on and produce symbolic expressions rather than numerical values. This can be very useful in debugging purely numerical calculations, because if they are given symbolic arguments we can examine the resulting symbolic expressions to make sure that the program is calculating what we intend it to. It also can be the basis of a partial evaluator for optimization of numerical programs.

Here is one way to accomplish this goal. We introduce the idea of an *arithmetic package*. An arithmetic package, or just *arithmetic*, is a map from operator names to implementations.[1] We can install an arithmetic in the user's Scheme environment. An installation replaces the default bindings of the operators named in the arithmetic with the arithmetic's implementations.[2]

The procedure `make-arithmetic-1` generates a new arithmetic package.[3] It takes a name for the new arithmetic, and an operation-generator procedure that given an operator name constructs an *operation*, here a handler procedure, for that operator. The procedure `make-arithmetic-1` calls the operation-generator procedure with each each arithmetic operator, accumulating the results

[1]The list of arithmetic operator names is in code file `operators.scm`.

[2]The code that supports arithmetics is in the code file `arith.scm`. The code that supports the general package abstraction is in code file `package.scm`.

[3]The procedure `make-arithmetic-1` is given at the end of code file `standard-arith.scm`.

into a new arithmetic package. To implement symbolic arithmetic the operation is implemented as a procedure that creates a symbolic expression by `cons`ing the operator name onto the list of its arguments.

```
(define symbolic-arithmetic-1
  (make-arithmetic-1 'symbolic
    (lambda (operator)
      (lambda args (cons operator args)))))
```

To use this newly-defined arithmetic, we install it, which redefines the arithmetic operators to use this arithmetic:

```
(install-arithmetic! symbolic-arithmetic-1)
```

`install-arithmetic!` changes the values of the user's global variables that are the names of the arithmetic operators defined in the arithmetic to their values in that arithmetic.

Now we can observe the result of taking one step of the evolution:

```
(pp (x 0
       ((evolver 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 (negate xt)) (* -2 (negate xt-h)))
         (negate xt-2h))))
```

Of course, we could easily produce simplified expressions by replacing the `cons` in `symbolic-arithmetic-1` with an algebraic simplifier, and then we have a symbolic manipulator. (We'll explore this later.)

This transformation was ridiculously easy, and yet our original design didn't make any provisions for symbolic computation. We could just as easily add support for vector arithmetic, matrix arithmetic, etc.

The ability to redefine operators *after the fact* gives both extreme flexibility and ways to make whole new classes of bugs! (We anticipated such a problem in the `evolver` procedure and avoided it by using the special arithmetic operators `n:>` and `n:-` for counting steps.)

There are more subtle problems. A program that depends on the commutativity of numerical multiplication will certainly not work correctly for matrices. (Of course, a program that depends on the exactness of operations on integers will not work correctly for inexact floating-point numbers either.) This is exactly the risk that comes with the evolution of biological or technological systems—some mutations will be fatal! On the other hand, some mutations will be extremely valuable. But that risk must be balanced against the cost of narrow and brittle construction.

Indeed, it is probably impossible to prove very much about a program when the primitive operations can be redefined, except that it will work when restricted to the types it was defined for. This is an easy but dangerous path for generalization.

Combining arithmetics

The symbolic arithmetic cannot do numerical calculation, so we have broken our integration example by replacing the operator definitions. We really want to have our operators map to the appropriate handler procedure for handling operands of different types. Thus the arithmetic packages must be able to determine if a handler is appropriate for the operands tendered.

By annotating each operation with specifications that determine its applicability we can combine different kinds of arithmetic. For example, we can combine symbolic and numerical arithmetic so that a combined operation can determine which implementation is appropriate for its arguments.[4]

We can introduce a procedure, `make-operation`, to make an operation that includes an applicability for the handler procedure, like this:

```
(define (make-operation operator applicability procedure)
  (list 'operation operator applicability procedure))
```

It is then possible to get the applicability specifications for an operation:

```
(define (operation-applicability operation)
  (caddr operation))
```

We introduce an abstraction for writing applicability information for an operation. `all-args` takes two arguments, the first being the number of arguments that the operation accepts (its *arity*), and the second being a predicate that must be true of each argument. It returns an applicability specification that can be used to determine if the operation is applicable to the arguments supplied to it. In a numeric arithmetic, each operation takes numbers for each of its arguments.

Using `all-args` we can implement an operation constructor for the simplest operations:

```
(define (simple-operation operator predicate procedure)
  (make-operation operator
                  (all-args (operator-arity operator)
                            predicate)
                  procedure))
```

The code that supports annotation is in the code file `operations.scm`.

We will also find it useful to have an "in-domain?" predicate that is true for the objects (such as functions or matrices) that a given arithmetic's operators take as arguments. To support this we'll create a new procedure `make-arithmetic` that is like `make-arithmetic-1` but has an additional `domain-predicate` argument. It also takes a list of arithmetics that the new arithmetic may be built on (We will need this soon). We'll also add a procedure `arithmetic-domain-predicate` that accepts an arithmetic package as an argument and returns its in-domain? predicate.

[4]A procedure is deemed applicable to arguments if the arguments satisfy a *specification.* A specification is a list of predicates, such as `number?` or `symbolic?`. The *applicability* of a procedure is a list of specifications, each of which is a list of predicates that the corresponding arguments must satisfy. If any one list of specifications of the applicability is satisfied the procedure is applicable.

Using these new capabilities, we can define a numeric arithmetic with applicability information:[5]

```
(define numeric-arithmetic
  (make-arithmetic 'numeric number? '()
    (lambda (name)                ;constant generator
      (case name
        ((additive-identity) 0)
        ((multiplicative-identity) 1)
        (else (default-object))))
    (lambda (operator)            ;operation generator
      (simple-operation operator number?
         (get-implementation-value
            (operator->procedure-name operator))))))
```

We can similarly rewrite the `symbolic-arithmetic` constructor:

```
(define (symbolic-extender base-arithmetic)
  (make-arithmetic 'symbolic symbolic? (list base-arithmetic)
    (lambda (name base-constant)        ;constant generator
      base-constant)
    (let ((base-predicate
           (arithmetic-domain-predicate base-arithmetic)))
      (lambda (operator base-operation) ;operation generator
        (make-operation operator
                        (any-arg (operator-arity operator)
                                 symbolic?
                                 base-predicate)
                        (lambda args
                          (cons operator args)))))))
```

One difference between this and the numeric arithmetic is that the symbolic arithmetic is applicable whenever *any* argument is a symbolic expression.[6] This is indicated by the use of `any-arg` rather than `all-args`. The third argument to `any-arg` is a predicate for whatever is acceptable in combination with a symbolic quantity.[7] Also notice that this symbolic arithmetic is based on a provided `base-arithmetic`, which in principle will allow us to build a variety of such arithmetics.

These parts are of the same shape, by construction. The `symbolic-extender` procedure produces an arithmetic with the same operators as the `base-arithmetic` it is given. This suggests that making a combinator language for building bigger arithmetics from parts might be a good approach.

The procedure `add-arithmetics`, below, is a combinator for arithmetics. It makes a new arithmetic whose domain predicate is the disjunction of the given arithmetics, and each of the operators is mapped to the union of the operations for the given arithmetics.[8]

[5]Since numeric arithmetic is built on the Scheme substrate the appropriate handler for the operator for Scheme number arguments is just the value of the operator symbol for the Scheme implementation. Also, certain symbols, such as the identities for addition and multiplication and the unary negation and inversion operators are specially mapped.

[6]Another difference you may have noticed is that the constant-generator and operation-generator procedures for the numeric arithmetic have only one formal parameter, while the generator procedures for the symbolic extender have two. The symbolic arithmetic is built on a base arithmetic, so the constant or operation for the base arithmetic is made available to the generator.

[7]The call `(any-arg 3 p1? p2?)` will produce applicability specifications with three parts, because there are three ways that this applicability can be satisfied: `((p1? p2? p2?) (p2? p1? p2?) (p2? p2? p1?))`

[8]`disjoin*` is a predicate combinator. It accepts a list of predicates and produces the predicate that is their disjunction. It is defined in the code file `predicates.scm`.

```
(define (add-arithmetics . arithmetics)
  (add-arithmetics* arithmetics))

(define (add-arithmetics* arithmetics)
  (if (n:null? (cdr arithmetics))
      (car arithmetics)
      (make-arithmetic 'add
                       (disjoin*
                        (map arithmetic-domain-predicate
                             arithmetics))
                       arithmetics
                       constant-union
                       operation-union)))
```

The third argument to `make-arithmetic` is a list of the arithmetic packages being combined. The arithmetic packages must be compatible in that they specify operations for the same named operators. The fourth argument is `constant-union`, which combines multiple constants. Here this selects one of the argument constants for use in the combined arithmetic; later we will elaborate on this.[9]

```
(define (constant-union name . constants)
  (let ((unique
         (remove default-object?
                 (delete-duplicates constants eqv?))))
    (if (n:pair? unique)
        (car unique)
        (default-object))))
```

The last argument is `operation-union`, the procedure that constructs the operation for the named operator in the resulting arithmetic.

```
(define (operation-union operator . operations)
  (operation-union* operator operations))

(define (operation-union* operator operations)
  (make-operation operator
                  (applicability-union*
                   (map operation-applicability operations))
                  (lambda args
                    (operation-union-dispatch operator
                                              operations
                                              args))))
```

The procedure `operation-union-dispatch` must determine which of the `operations`, to use based on the arguments supplied. it chooses the operation from the given arithmetics that is appropriate to the given arguments and applies it to the arguments. The first arithmetic has priority in that if the operations from multiple arithmetics are applicable then the first is chosen.

[9]The procedure `default-object` produces an object that is different from any possible constant. The procedure `default-object?` identifies that value.

```
(define (operation-union-dispatch operator operations args)
  (let ((operation
         (find (lambda (operation)
                 (is-operation-applicable? operation args))
               operations)))
    (if (not operation)
        (error "Inapplicable operation:" operator args))
    (apply-operation operation args)))
```

A common pattern is to combine a base arithmetic with an extension on that arithmetic. The combination of numeric arithmetic and a symbolic arithmetic built on numeric arithmetic is such a case. So we provide an abstraction for that pattern:

```
(define (extend-arithmetic extension base-arithmetic)
  (add-arithmetics base-arithmetic
                   (extension base-arithmetic)))
```

We can use `extend-arithmetic` to combine the numerical arithmetic and the symbolic arithmetic, with the numerical arithmetic having priority. Since these are disjoint the order is irrelevant except for possible performance issues.

```
(define combined-arithmetic
  (extend-arithmetic symbolic-extender numeric-arithmetic))

(install-arithmetic! combined-arithmetic)
```

We can use the composite arithmetic. It works numerically:

```
(x 0 ((evolver F .01 stormer-2) s0 100))
.8414709493275624
```

It works symbolically:

```
(pp (x 0
       ((evolver 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 (- xt))
         (* -2 (- xt-h))
         (- xt-2h)))))
```

And it works in combination:

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

Arithmetic on functions

Traditional mathematics extends arithmetic on numerical quantities to many other kinds of objects. Over the centuries "arithmetic" has been extended to complex numbers, vectors, linear transformations and their representations as matrices, etc. One particularly revealing extension is to functions. We can combine functions of the same type using arithmetic operators:

$$\begin{aligned}
(f+g)(x) &= f(x)+g(x) \\
(f-g)(x) &= f(x)-g(x) \\
(fg)(x) &= f(x)g(x) \\
(f/g)(x) &= f(x)/g(x) \\
&\cdots
\end{aligned}$$

The functions that are combined must have the same domain and codomain, and the arithmetic must be defined on the codomain.

This extension is pretty easy to accomplish. Given an arithmetic package for the codomain of the functions that we wish to combine, we can make an arithmetic package that implements the function arithmetic, assuming that functions are implemented as procedures.

```
(define (pure-function-extender codomain-arithmetic)
  (make-arithmetic 'pure-function function?
                   (list codomain-arithmetic)
    (lambda (name codomain-constant)
      (lambda args codomain-constant))
    (lambda (operator codomain-operation)
      (simple-operation operator function?
        (lambda functions
          (lambda args
            (apply-operation codomain-operation
                             (map (lambda (function)
                                    (apply function args))
                                  functions))))))))
```

This extension is not very useful by itself. We need to combine it with the arithmetic that operates on the codomains to make a useful package:

```
(install-arithmetic!
  (extend-arithmetic pure-function-extender combined-arithmetic))
```

Now we can add functions as well as numbers and symbolic numerical quantities:

```
(* 'b ((+ cos sin) (+ 3 'a)))
(* b (+ (cos (+ 3 a)) (sin (+ 3 a))))
```

The mathematical tradition also allows one to mix numerical quantities with functions by treating the numerical quantities as constant functions of the same type as the functions they will be combined with.

$$(f+1)(x) = f(x)+1$$

We can implement the coercion of numerical quantities to constant functions quite easily, by a minor modification of the procedure `pure-function-extender`:

```
(define (function-extender codomain-arithmetic)
  (let ((codomain-predicate
         (arithmetic-domain-predicate codomain-arithmetic)))
    (make-arithmetic 'function
                     (disjoin codomain-predicate function?)
                     (list codomain-arithmetic)
      (lambda (name codomain-constant)
        codomain-constant)
      (lambda (operator codomain-operation)
        (make-operation operator
                        (any-arg (operator-arity operator)
                                 function?
                                 codomain-predicate)
          (lambda things
            (lambda args
              (apply-operation codomain-operation
                 (map (lambda (thing)
                        (if (function? thing)
                            (apply thing args)
                            thing))
                      things)))))))))
```

The domain of the new function arithmetic is the union of the functions and the codomain arithmetic (the possible values of the functions). The operator implementation is applicable if any of the arguments is a function; and functions are applied to the arguments that are given. Note that the constant generator doesn't need to rewrite the codomain constants as functions since the constants can now be used directly.

With this version we can

```
(install-arithmetic!
  (extend-arithmetic function-extender combined-arithmetic))

(* 'b ((+ 4 cos sin) (+ 3 'a)))
```
(b (+ 4 (cos (+ 3 a)) (sin (+ 3 a))))*

This raises an interesting problem: we have symbols that represent numbers, but none to represent functions. For example, if we write

```
(* 'b ((+ 'c cos sin) (+ 3 'a)))
```

our arithmetic will treat `c` as if it were a number. But we might wish to have `c` be a literal function that combines as a function. It's difficult to do this with our current design, because `c` carries no type information, and the context is insufficient to distinguish usages.

But we can make symbolic functions whose values are numeric. We make a function that returns a symbolic expression, which will be interpreted as a numerical symbolic expression. This is easy:

```
(define (literal-function name)
  (lambda args
    (cons name args)))
```

With this definition we can now have a symbolic function `c` correctly combine with other functions:

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

This is a narrow solution that handles a particularly useful case; later we will look into more general solutions.

The moral of this story

The arithmetic structures we've been building up to now are an example of the use of combinators to build complex structures by combining simpler ones. But there are some serious drawbacks to building this system using combinators. First, some properties of the structure are determined by the means of combination. For example, we pointed out that `add-arithmetics` prioritized its arguments, such that their order can matter. Second, the layering implicit in this design, such that the codomain arithmetic must be constructed prior to the function arithmetic, means that it's impossible to augment the codomain arithmetic after the function arithmetic has been constructed. Finally, we might wish to define an arithmetic for functions that return functions. This cannot be done in a general way within this framework, without introducing another mechanism for self reference.

The underlying problem is that construction by combinators requires careful design and planning. There must be a localized plan for how all the pieces are combined. This is not a problem for a well-understood domain, such as arithmetic, but it is not appropriate for open-ended construction. We'd prefer to be able to add new kinds of arithmetic incrementally, without having to decide where it goes in a heirarchy, and without having to change the existing parts that already work.

Systems built by 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.[10]

Exercises

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 (perhaps in a future problem set!). Also, we use MIT/GNU Scheme record structures at the lowest level, so you may need to read about them in the documentation.

You should copy the code from the website into your directory (or "folder," as young people say!) for this problem set. You can point MIT/GNU Scheme to the directory with `(cd <a string naming your directory>)`. There will be a file in the directory with the name `load.scm` which contains a loader for the problem-set files. They are loaded in the order of the load list in the load file. You can then load the code into MIT/GNU Scheme with `(load "load")`.

[10]At the APL-79 conference Joel Moses is reported to have said: "APL is like a beautiful diamond—flawless, beautifully symmetrical. But you can't add anything to it. If you try to glue on another diamond, you don't get a bigger diamond. Lisp is like a ball of mud. Add more and it's still a ball of mud—it still looks like Lisp." But Joel denies that he said this.

You can modify files in your problem-set directory, and you can make new ones, which you may add to the load sequence. However, please hand in only your modifications... We do not want to see huge amounts of code that repeats what we 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 (look at `arith.scm` to find the definition of `fixed-arity-to-variable`) 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 (for example `load` will fail in a wierd way!). You can get an entirely new top level read-eval-print loop environment by executing the following mysterious incantation in the broken one:

```
(ge (make-top-level-environment))
```

This will lose the broken environment and all definitions made in that environment, but your EDWIN editor buffers will be OK.

A minor point. Do not try to copy code from the text in the LaTeX problem-set. We supplied you with complete sources in ASCII characters in the code files. The problem is that the single-quote mark in the LaTeX printed version is not the correct character for Scheme to correctly interpret, so there will be mysterious, apparently invisible bugs in code copied from the LaTeX version.

If you find bugs in this problem set please tell me. This is a new implementation. GJS

Problem 2.1: Warmup

In digital design the boolean operations *and*, *or*, and *not* are written with the operators `*`, `+`, and `-`, respectively. There is a Scheme predicate `boolean?` that is true only of `#t` and `#f`. Use this to make a boolean arithmetic package that can be combined with the packages we have. Note that all other arithmetic operators are undefined for booleans, so the appropriate result of applying something like `cos` to a boolean is to report an error.

The following template could help get you started:

```
(define boolean-arithmetic
  (make-arithmetic 'boolean boolean? '()
    (lambda (name)
      (case name
        ((additive-identity) #f)
        ((multiplicative-identity) #t)
        (else (default-object))))
    (lambda (operator)
      (let ((procedure
             (case operator
               ((+) <...>)
               ((-) <...>)
               ((*) <...>)
               ((negate) <...>)
               (else
                (lambda args
                  (error "Operator undefined in Boolean"
                         operator))))))
        (and procedure
             (simple-operation operator boolean? procedure))))))
```

Hint: To make "-" work you will need to define the unary boolean operation for the operator "negate."

Problem 2.2: Vectors

We will make and install a package for arithmetic on geometric vectors. This is a big assignment that will bring to the surface many of the difficulties and inadequacies of the system we have developed so far.

a. We will represent vectors as Scheme `vector`s of numerical quantities. The elements of the vectors are coordinates relative to some Cartesian axes. There are a few issues here. Addition (and subtraction) is defined only for vectors of the same dimension, so your package must know about dimensions. First, make a package that defines only addition, negation, and subtraction of vectors over a base package of operations applicable to the coordinates of vectors. Applying any other operation to a vector should report an error.

Note: When you introduce a new primitive predicate, you will need to register it with the system. For example, to register the `vector?` predicate you must execute `(register-predicate! vector? 'vector)`. See `predicate-metadata.scm` where `register-predicate!` is defined and used to register some common predicates.

Hint: The following procedures will be helpful:

```
(define (vector-element-wise element-procedure)
  (lambda vecs    ; Note: this takes multiple vectors
    (ensure-vector-lengths-match vecs)
    (apply vector-map element-procedure vecs)))

(define (ensure-vector-lengths-match vecs)
  (let ((first-vec-length (vector-length (car vecs))))
    (if (any (lambda (v)
               (not (n:= (vector-length v)
                         first-vec-length)))
             vecs)
        (error "Vector dimension mismatch:" vecs))))
```

Build the required package and show that it works for numerical vectors and for vectors with mixed numerical and symbolic coordinates.

b. Your vector addition required addition of the coordinates. The coordinate addition procedure could be the value of the `+` operator that will be installed in the user environment, or it could be the addition operation from the base arithmetic of your vector extension. Either of these would satisfy many tests, and using the installed addition may actually be more general. Which did you use? Show how to implement the other choice. How does this choice affect your ability to make future extensions to this system? Explain your reasoning.

Style Hint: A nice way to control the interpretation of operators in a procedure is to provide the procedure to use for each operator as arguments to a "maker procedure" that returns the procedure needed. For example, to control the arithmetic operations used in `vector-magnitude` one might write:

```
(define (vector-magnitude-maker + * sqrt)
  (let ((dot-product (dot-product-maker + *)))
    (define (vector-magnitude v)
      (sqrt (dot-product v v)))
    vector-magnitude))
```

c. What shall we do about multiplication? First, for two vectors it is reasonable to define multiplication to be their dot product. But there is a bit of a problem here. You need to be able to use both the addition and multiplication operations, perhaps from the base package of arithmetic on the coordinates. This is not hard to solve. Modify your vector package to define multiplication of two vectors as their dot product. Show that your dot product works.

d. Add vector magnitude to your vector package. The code given above is most of the work!

e. Multiplication of a vector by a scalar or multiplication of a scalar by a vector should produce the scalar product (the vector with each coordinate multiplied by the scalar). So multiplication can have two meanings, dot product or scalar product, depending on the types of its arguments. Modify your vector package to make this work. This will require some testing of the arguments to the operator `*`, in addition to the annotation mechanism. Show that your vector package can handle both dot product and scalar product.

Problem 2.3: Ordering of Extensions

Consider two possible orderings for combining your vector extension with the existing arithmetics:

```
(define vec-before-func
 (extend-arithmetic
  function-extender
  (extend-arithmetic vector-extender combined-arithmetic)))

(define func-before-vec
 (extend-arithmetic
  vector-extender
  (extend-arithmetic function-extender combined-arithmetic)))
```

How does this ordering of extensions affect the properties of the resulting arithmetic? The following procedure makes points on the unit circle:

```
(define (unit-circle x)
  (vector (sin x) (cos x)))
```

Now suppose we try to execute each of the following expressions in environments resulting from installing `vec-before-func` and `func-before-vec`:

```
((magnitude unit-circle) 'a)

((magnitude (vector sin cos)) 'a)
```

The result (unsimplified) should be:

```
(sqrt (+ (* (sin a) (sin a)) (* (cos a) (cos a))))
```

However, each of these expressions fails with one of the two orderings of the extensions. Is it possible to make an arithmetic for which both evaluate correctly? Explain.

Problem 2.4: Literal Vectors

(Optional) It is also possible to have arithmetic on literal vectors with an algebra of symbolic expressions of vectors. Can you make symbolic algebra of these compound structures play well with vectors with symbolic and numerical expressions as elements? This gets more fun when integrating this with matrices. Perhaps this would be part of an appropriate term project.