MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.945/6.905 Spring 2017
Problem Set 5

Issued: Wed. 15 March 2017 **Due: Wed. 24 March 2017**

Reading: SICP Section 4.4, especially 4.4.4.3 and 4.4.4.4
MIT Scheme Reference Manual, section 2.11: Macros. This is complicated stuff, so don't try to read it until you need to.

Code: load.scm, matcher.scm, rule-implementation.scm, rules.scm, ghelper.scm, utils.scm, pattern-operator.scm

Generic operators allow us to do miraculous things by modulating the meanings of the free variables in a program. We may think of the program that employes a generic operator, such as "+" as advertising for a handler that can do the job of "+ing" the arguments supplied. Handlers are attached to a generic operator with defhandler. A handler is applicable if the arguments supplied satisfy the predicates presented in the defhandler declaration. A handler may present several such applicability criteria describing combinations of arguments it can handle. This idea is quite general, but the language for advertising jobs that need to be done is rather limited -- all we have is a single symbol, in this case "+".

One way to generalize this idea is through the use of patterns, compound expressions that describe the work to be done. In a general version of this idea the handlers also present pattern, and the patterns are "unified" to determine the applicability. We may talk about this later, but first let's see what we can do with simple patterns.

Pattern Matching and Instantiation

One of the most important techniques in symbolic programming is the use of patterns. We can match patterns against data objects, and we can instantiate a pattern to make a data object. For example, the elementary laws of algebra are usually expressed as patterns and instantiations:

a * (b + c) <==> a * b + a * c

This is the distributive law of multiplication over addition. It says that we can replace one side with the other without changing the value of the expression. Each side of the law is a pattern, with particular pattern variables a, b, c, and pattern constants *, +. What the law says is that if we find an algebraic expression that is the product of something and a sum of terms, we can replace it with a sum of two products, and vice versa.

Let's see how to organize programs based on pattern-match. A key idea will be compilation of patterns into combinators that are the pieces of a matcher. Then we will demonstrate this in a term-rewriting system for elementary algebra.

A Language of Patterns

The first job is to make up our language of patterns. We will start with something simple. We will make our patterns out of Lisp (Scheme) lists. Unlike the mathematical example above, we will not have reserved symbols, such as * and +, so we will have to distinguish pattern variables from pattern constants. Pattern variables can be represented by lists beginning with the query symbol: "?". This is a traditional choice. So in this language the patterns that make up the distributive law may be represented as follows, assuming that we are manipulating Lisp prefix mathematical expressions:

```
(* (? a) (+ (? b) (? c)))

(+ (* (? a) (? b)) (* (? a) (? c)))
```

You might complain that we could have used distinguished symbols, such as "?a" instead of the long-winded (? a). That would be fine, but that choice will make it a bit harder to extend, say if we want a variable that is restricted to match only numbers. Of course, we can add syntax later if it is helpful, but remember Alan Perlis's maxim: "Syntactic sugar causes cancer of the semicolon."

One constraint on our design of the matcher is that the second pattern above should match

```
(+ (* (cos x) (exp y)) (* (cos x) (sin z)))
```

where a=(cos x), b=(exp y), and c=(sin z). It should not match

```
(+ (* (cos x) (exp y)) (* (cos (+ x y)) (sin z)))
```

because there is no consistent assignment possible for (? a) (unless, somehow x=x+y. We will learn about that sort of stuff later if it will be necessary to study unification matching. Here we will decide that there is no match possible.)

Another constraint, which will have important influence on the structure of the matcher, is the requirement for "segment variables." Suppose we want to find instances of particular patterns in a sequence of many terms. For example, suppose we want to make a rule to find combinations of squares of sines and cosines and replace them with 1:

```
(+ ... (expt (sin theta) 2) ... (expt (cos theta) 2) ...)
  ==> (+ 1 ... ... ...)
```

The "..." here may stand for many terms. We will need segment variables, with the prefix "??", that can match many terms. So the pattern we will write is:

```
(+ (?? t1) (expt (sin (? x)) 2) (?? t2) (expt (cos (? x)) 2) (?? t3))
  ==> (+ 1 (?? t1) (?? t2) (?? t3))
```

Segment variables have a profound effect, because we don't know how long a segment is until we find the next part that matches, and we may be able to match the same data item many ways. For example, there may be both squares of sines and cosines of angles theta and phi in the same sum. Even simpler, the pattern

```
(a (?? x) (?? y) (?? x) c) can match the datum (a b b b b b b c)
```

in four different ways. (Notice that the segment variable x must eat up the same number of "b"s in the two places it appears in the pattern.) So the matcher must do a search over the space of possible assignments to the segment variables.

Design of the Matcher

A matcher for a particular pattern is constructed from a family of mix-and-match combinators that can be combined to make combinators of the same type. Each primitive element of the pattern is represented by a primitive combinator and the only combination, list, is represented by a combinator that combines combinators to make a compound one.

The match combinators are procedures that take three arguments: a list containing data to be matched, a dictionary of bindings of pattern variables, and a procedure to be called if the match is successful. The arguments to the succeed procedure must be the new dictionary resulting from the match and the number of items that were consumed from the data. A match combinator returns #f if the match is unsuccessful.

There are four basic match combinators. Let's go through them.

The match procedure match:eqv takes a pattern constant and produces a match combinator. It succeeds if and only if the first data item is equal (using eqv?) to the pattern constant. If successful, it does not add to the dictionary. The second argument to succeed is the number of items matched, which is 1 for this match procedure.

```
(define (match:eqv pattern-constant)
  (define (eqv-match data dictionary succeed)
    (and (pair? data)
         (eqv? (car data) pattern-constant)
         (succeed dictionary 1)))
  eqv-match)
```

So, for example:

```
(define x-matcher (match:eqv 'x))

(define a-list-of-data '(x))

(define an-empty-dictionary '())

(define another-list-of-data '(y))

(define (result-receiver dict n-eaten) `(success ,dict ,n-eaten))

(x-matcher a-list-of-data an-empty-dictionary result-receiver)
;Value: (success () 1)

(x-matcher another-list-of-data an-empty-dictionary result-receiver)
;Value: #f
```

The match procedure match:element is used to make a match combinator for an ordinary pattern variable, such as (? x). There are two cases of match. If the variable already has a value in the dictionary the combinator succeeds only if the value is equal (using equal?) to the first data item. If the variable has no value, the combinator succeeds with the new dictionary resulting from binding the variable to the first data item in the given dictionary.

```
(define (match:element variable)
  (define (element-match data dictionary succeed)
    (and (pair? data)
         (let ((vcell (match:lookup variable dictionary)))
           (if vcell
               (and (equal? (match:value vcell) (car data))
                    (succeed dictionary 1))
               (succeed (match:bind variable
                                    (car data)
                                    dictionary)
                        1)))))
  element-match)
```

And, some examples:

```
;;; Most of the time I do not name simple stuff...

((match:element 'x) '(a) '() result-receiver)
;Value: (success ((x a)) 1)

((match:element 'x) '(a b) '() result-receiver)
;Value: (success ((x a)) 1)

((match:element 'x) '((a b) c) '() result-receiver)
;Value: (success ((x (a b))) 1)
```

A segment variable procedure makes a more interesting combinator because it can consume more than one data item. A segment matcher must inform its caller not only of the new dictionary, but also how many items from the data were eaten.

```
(define (match:segment variable)
  (define (segment-match data dictionary succeed)
    (and (list? data)
         (let ((vcell (match:lookup variable dictionary)))
           (if vcell
               (let lp ((data data)
                        (pattern (match:value vcell))
                        (n 0))
                 (cond ((pair? pattern)
                        (if (and (pair? data)
                                 (equal? (car data) (car pattern)))
                            (lp (cdr data) (cdr pattern) (+ n 1))
                            #f))
                       ((not (null? pattern)) #f)
                       (else (succeed dictionary n))))
               (let ((n (length data)))
                 (let lp ((i 0))
                   (if (<= i n)
                       (or (succeed (match:bind variable
                                                (list-head data i)
                                                dictionary)
                                    i)
                           (lp (+ i 1)))
                       #f)))))))
  segment-match)
```

For example:

```
((match:segment 'a) '(z z z) '() result-receiver)
;Value: (success ((a ())) 0)

;;; Of course, a zero-length segment is OK.


;;; But if we want to see all of the possible matches:

(define (print-all-results dict n-eaten)
  (pp `(success ,dict ,n-eaten))
  #f)                    ;force backtracking

((match:segment 'a) '(z z z) '() print-all-results)
(success ((a ())) 0)
(success ((a (z))) 1)
(success ((a (z z))) 2)
(success ((a (z z z))) 3)
;Value: #f
```

Because the input data is a list we can compute its length. Again, there are two possibilities, either the segment variable already has a value or it does not yet have a value. If it has a value, each item of the value must be the same as a corresponding item from the data. If this is true, the match succeeds, eating a number of items from the input data equal to the number in the stored value. If the segment variable does not yet have a value it must be given one. The segment-variable match combinator starts by assuming that the segment will eat no items from the data. However, if that success ultimately leads to a later failure in the match, the segment tries to eat one more element than it had already tried. (This is accomplished by executing (lp (+ i 1)).) If the segment variable runs out of data items, it fails to match.

Finally, there is the list matcher:

```
(define (match:list . match-combinators)
  (define (list-match data dictionary succeed)
    (and (pair? data)
         (let lp ((lst (car data))
                  (matchers match-combinators)
                  (dictionary dictionary))
           (cond ((pair? matchers)
                  ((car matchers)
                   lst
                   dictionary
                   (lambda (new-dictionary n)
                     (if (> n (length lst))
                         (error "Matcher ate too much."
                                n))
                     (lp (list-tail lst n)
                         (cdr matchers)
                         new-dictionary))))
                 ((pair? lst) #f)
                 ((null? lst)
                  (succeed dictionary 1))
                 (else #f)))))
  list-match)
```

The match:list procedure takes match combinators and makes a list combinator that matches a list if and only if the given match combinators eat up all of the elements in the data list. It applies the combinators in succession. Each combinator tells the list combinator how many items to jump over before passing the result to the next combinator.

Notice that the list combinator has exactly the same interface as a single element, allowing it to be incorporated into a combination.

The dictionary we will use is just an alist of variable-value pairs:

```
(define (match:bind variable data-object dictionary)
  (cons (list variable data-object) dictionary))

(define (match:lookup variable dictionary)
  (assq variable dictionary))

(define (match:value vcell)
  (cadr vcell))
```

Using Match Combinators

For example, we can make a combinator that matches a list of any number of elements, starting with the symbol a, ending with the symbol b, and with a segment variable (?? x) between them by the combination:

```
  (match:list (match:eqv 'a) (match:segment 'x) (match:eqv 'b))
```

We can apply it to data. The initial dictionary is the empty list.

```
((match:list (match:eqv 'a) (match:segment 'x) (match:eqv 'b))
 '((a 1 2 b))
 '()
 (lambda (dict n-eaten) (list dict n-eaten)))
;Value: (((x (1 2))) 1)
```

This was a successful match. The dictionary returned has exactly one entry: x=(1 2), and the match ate precisely one element (the sublist) from the list supplied.

```
((match:list (match:eqv 'a) (match:segment 'x) (match:eqv 'b))
 '((a 1 2 b 3))
 '()
 (lambda (dict n-eaten) (list dict n-eaten)))
;Value: #f
```

This was a failure, because there was nothing to match the 3 after the b in the input data.

We can automate the construction of pattern matchers from patterns with an elementary compiler. First, we need to define the syntax.

```
(define (match:element? pattern)
  (and (pair? pattern)
       (eq? (car pattern) '?)))

(define (match:segment? pattern)
  (and (pair? pattern)
       (eq? (car pattern) '??)))

(define (match:variable-name pattern)
  (cadr pattern))

(define (match:list? pattern)
  (and (list? pattern)
       (or (null? pattern)
           (not (memq (car pattern) '(? ??))))))
```

List syntax is any list that does not begin with a variable declaration.

The compiler itself is just a generic operator.

```
(define match:->combinators (make-generic-operator 1 'eqv match:eqv))

(defhandler match:->combinators
  (lambda (pattern) (match:element (match:variable-name pattern)))
  match:element?)

(defhandler match:->combinators
  (lambda (pattern) (match:segment (match:variable-name pattern)))
  match:segment?)

(defhandler match:->combinators
  (lambda (pattern)
    (apply match:list (map match:->combinators pattern)))
  match:list?)
```

By varying this compiler, we can change the syntax of patterns any way we like.

The compiler produces as its value a match combinator appropriate for the pattern it is given: it has exactly the same interface as the elementary combinators given. Some simple examples are:

```
((match:->combinators '(a ((? b) 2 3) (? b) c))
 '((a (1 2 3) 2 c))
 '()
 (lambda (dict n-eaten) (list 'succeed dict n-eaten)))
;Value: #f

((match:->combinators '(a ((? b) 2 3) (? b) c))
 '((a (1 2 3) 1 c))
 '()
 (lambda (dict n-eaten) (list 'succeed dict n-eaten)))
;Value: (succeed ((b 1)) 1)
```

Some patterns involving segment variables may match in many ways. We can elicit all of the matches by failing back into the matcher to select the next one, until they are all exhausted:

```
((match:->combinators '(a (?? x) (?? y) (?? x) c))
      '((a b b b b b b c))
      '()
      (lambda (dict n-eaten)
        (pp '(succeed ,dict ,n-eaten))
        #f))
(succeed ((y (b b b b b b)) (x ())) 1)
(succeed ((y (b b b b)) (x (b))) 1)
(succeed ((y (b b)) (x (b b))) 1)
(succeed ((y ()) (x (b b b))) 1)
;Value: #f
```

Problem 4.1:

In the example above we got multiple matches, by returning #f from a success procedure. This is probably pretty mysterious. How does it work? Explain, in a short but concise paragraph, how the sequence of matches is generated.

This interface is not very nice for using the matcher. It is convenient for building into other systems that need this flexibility. However, for playing with the matcher it is convenient to use:

```
(define (matcher pattern)
  (let ((match-combinator (match:->combinators pattern)))
    (lambda (datum)
      (match-combinator (list datum)
                        '()
                        (lambda (dictionary n)
                          (and (= n 1)
                               dictionary))))))
```

With this interface we explicitly give the pattern to get the matcher procedure. We give the matcher procedure the datum the pattern is to match, and we get either a dictionary or #f.

```
((matcher '(a ((? b) 2 3) (? b) c))
 '(a (1 2 3) 1 c))
;Value: ((b 1))
```

Restrictions

Quite often we want to restrict the kind of object that can be matched by a pattern variable. For example, we may want to make a pattern where a variable can only match a positive integer. For example, we may be interested in finding positive integer powers of sine functions. We could write the pattern we want as follows:

```
'(expt (sin (? x)) (? n ,exact-positive-integer?))
```

We need a matcher procedure that can be used to make a matcher combinator satisfying this requirement. One way to do this is to make match:element take an extra argument, perhaps optional, which is the predicate for testing the datum for acceptability. This change is as follows (including a replacement syntax handler):

```
(define (match:element variable restrictions)
  (define (ok? datum)
    (every (lambda (restriction)
             (restriction datum))
           restrictions))
  (define (element-match data dictionary succeed)
    (and (pair? data)
         (ok? (car data))
         (let ((vcell (match:lookup variable dictionary)))
           (if vcell
               (and (equal? (match:value vcell) (car data))
                    (succeed dictionary 1))
               (succeed (match:bind variable
                                    (car data)
                                    dictionary)
                        1)))))
  element-match)

(defhandler match:->combinators
  (lambda (pattern)
    (match:element (match:variable-name pattern)
                   (match:restrictions pattern)))
  match:element?)
```

There was a nasty problem of evaluation of the predicate expression with the compilation of the matcher syntax for the restriction predicate (exact-positive-integer?). The symbols expt and sin are pattern constants, but the symbol exact-positive-integer? must be evaluated to a procedure. We specify the input patterns using quasiquote (a.k.a. backquote) to solve this problem. For example,

```
‘(a b ,(+ 20 3) d) --> (a b 23 d)
```

Consult the MIT Scheme Reference Manual for details on QUASIQUOTE.

Choice is Good

An interesting way to extend our pattern language is to introduce a choice operator:

```
(?:choice <pattern>...)
```

This should compile into a combinator that tries to match each of the (possibly null) list of <pattern>s in order from left to right, returning the first successful match or #f if none match. (This should remind you of regular expression "alternation" but the choice of the name "choice" is more traditional in pattern matching.)

For example:

```
((match:->combinators '(?:choice a b (? x) c))
 '(z)
 '()
 (lambda (d n) '(succeed ,d)))
;Value: (succeed ((x z)))

((match:->combinators
 '((? y) (?:choice a b (? x ,string?) (? y ,symbol?) c)))
 '((z z))
 '()
 (lambda (d n) '(succeed ,d)))
;Value: (succeed ((y z)))

((match:->combinators '(?:choice b (? x ,symbol?)))
  '(b) '()
  (lambda (x n)
    (pp '(succeed ,x))
    #f))
(succeed ())
(succeed ((x b)))
;Value: #f
```

Problem 4.2:

Implement a new matcher procedure, match:choice, for this new pattern schema. Augment the pattern compiler appropriately.

As always, demonstrate your code on the examples provided and on a couple of your own, both positive and negative boundary cases.

Naming is Better

Another extension is to provide named patterns, analogous to Scheme's LETREC.

Naming allows shorter, more modular patterns while also supporting recursive sub-patterns, including mutually recursive sub-patterns.

For instance, the pattern:

```
(?:pletrec ((odd-even-etc (?:choice () (1 (?:ref even-odd-etc))))
            (even-odd-etc (?:choice () (2 (?:ref odd-even-etc)))))
   (?:ref odd-even-etc))
```

...should match all lists of the following form (including the empty list):

```
(1 (2 (1 (2 (1 (... ())...)))))
```

Here, ?:PLETREC introduces a block of mutually recursive pattern definitions while ?:REF dereferences a defined pattern in place (in order to distinguish them from literal symbols like "a" and from pattern variables like "(? x)").

In a proper environment-based LETREC-like implementation, nested ?:PLETREC instances would introduce distinct contour lines for scoping. You needn't worry about that here. Specifically, just as pattern variables all share a common global namespace, so too can your pattern definitions.

To wit, notice how the pattern combinators traverse the pattern and data in left-to-right depth-first order, binding the first textual appearance of each distinct pattern variable (like "(? x)") to its corresponding datum then treating each subsequent textual appearance in the pattern as a constraining instance. This is achieved by threading the dictionary through the depth-first control path. Pay particular attention to the appearance of NEW-DICTIONARY in the body of MATCH:LIST.

This, in essence, decrees the leftmost, deepest instance of each unique pattern variable to be a defining instance in an implicit flat global namespace with all subsequent downstream appearances being constraining instances.

Feel free to make PLETREC behave similarly rather than rewrite all the existing combinator interfaces to accept an extra PATTERN-ENVIRONMENT parameter, or whatever.

Problem 4.3

Implement these new PLETREC and REF pattern schemata. One approach is to implement new matcher procedures, match:pletrec and match:ref, then augment the pattern compiler appropriately. Other approaches may also work. Explain your approach briefly if it is subtle or non-obvious.

As always, demonstrate your code on the examples provided and on a couple of your own, both positive and negative boundary cases.

Term Rewriting

We extend our pattern matching system to build a primitive algebraic simplifier, based on pattern matching and instantiation.

In rules.scm there are two elementary rule systems. In our system a rule has two parts: a pattern to match a subexpression, and a consequent expression. If the pattern matches, the consequent is evaluated and its result replaces the matched subexpression.

The rules are assembled into a list and handed to the rule-simplifier procedure. The result is a simplifier procedure that can be applied to an algebraic expression.

The first rule system demonstrates only elementary features. It does not use segment variables or restricted variables. The first system has three rules: The first rule implements the associative law of addition, the second implements the commutative law of multiplication, and the third implements the distributive law of multiplication over addition.

The commutative law looks like:

```
(rule '(* (? b) (? a))
      (and (expr<? a b)
           '(* ,a ,b)))
```

Notice the restriction predicate in the consequent of the rule for the commutative law. If the consequent expression returns #f, that match is considered to have failed. The system backtracks into the matcher to look for an alternative match; if none are forthcoming, the rule is not applicable. In the commutative law the restriction predicate expr<? imposes an ordering on algebraic expressions.

**---
Problem 4.4:**

**Why is the (expr<? a b) restriction necessary in the commutative law? What would go wrong if the there was no restriction?
---**

The second system of rules is far more interesting. It is built with the assumption that addition and multiplication are n-ary operations: it needs segment variables to make this work. It also uses variable restrictions to allow rules for simplifying numerical terms and prefactors.

**---
Problem 4.5:**

In the second system how does the use of the ordering on expressions imposed by the commutative laws make the numerical simplification rules effective?

**Suppose that the commutative laws did not force an ordering, how would we have to write the numerical simplification rules? Explain why numerical simplification would become very expensive.
---**

**---
Problem 4.6:**

**The ordering in the commutative laws evolves an n^2 bubble sort on the terms of a sum and the factors of a product. This can get pretty bad if there are many terms, as in a serious algebra problem. Is there some way in this system to make a more efficient sort? If not, why not? If so, how would you arrange it?
---**

**---
Problem 4.7:**

The system we have described does not collect like terms. For example:

```
(algebra-2 '(+ (* 4 x) (* 3 x)))
;Value (+ (* 3 x) (* 4 x))
```

Add rules that cause the collection of like terms, leaving the result as a sum of terms. Demonstrate your solution. Your solution must be able to handle problems like:

```
(algebra-3
  '(+ y (* x -2 w) (* x 4 y) (* w x) z (* 5 z) (* x w) (* x y 3)))
;Value: (+ y (* 6 z) (* 7 x y))
```

Now that we have some experience with the use of such a rule system, let's dive in to see how it works. The center of the system is in rule-implementation.scm.

A rule is a procedure that matches a pattern against its argument. If the match succeeds, it executes its consequent in an environment where the pattern variables are bound to their matched data. Rule procedures normally take succeed and fail continuations that can be used to backtrack into the consequent or match of a rule. For ease of testing rules, these continuations are optional, and reasonable defaults are supplied.

Two interesting procedures use rules. The rule-simplifier procedure is a simple recursive simplifier constructor. It produces a procedure, simplify-expression, that takes an expression and uses the rules to simplify the expression. It recursively simplifies all the subexpressions of an expression, and then applies the rules to simplify the resulting expression. It does this repeatedly until the process converges and the expression returned is a fixed point of the simplification process.

```
(define (rule-simplifier the-rules)
  (define (simplify-expression expression)
    (let ((subexpressions-simplified
           (if (list? expression)
               (map simplify-expression expression)
               expression)))
      (try-rules subexpressions-simplified the-rules
       (lambda (result fail)
         (simplify-expression result))
       (lambda ()
         subexpressions-simplified))))
  simplify-expression)
```

The procedure try-rules just scans the list of rules, sequencing the scan through the succeed and fail continuations.

```
(define (try-rules data rules succeed fail)
  (let per-rule ((rules rules))
    (if (null? rules)
        (fail)
        ((car rules) data succeed
         (lambda ()
           (per-rule (cdr rules)))))))
```

Another mechanism that uses rules (and try-rules) is an extension of the generic operation idea to pattern-directed operators. The make-pattern-operator procedure is analagous to make-generic-operator; it is an entity to allow the operator to be extended with additional rules dynamically (by attach-rule!, which is analagous to defhandler).

```
(define (make-pattern-operator #!optional rules)
  (define (operator self . arguments)
    (define (succeed value fail) value)
    (define (fail)
      (error "No applicable operations" self arguments))
    (try-rules arguments (entity-extra self) succeed fail))
  (make-entity operator (if (default-object? rules) '() rules)))

(define (attach-rule! operator rule)
  (set-entity-extra! operator
   (cons rule (entity-extra operator))))
```

Generic operations systems usually have no name for the analogue to our rules. That analogue is the generic method, together with the set of types or predicates that determine the parameters to which the method is applicable. Since the applicability testing and destructuring portions of a generic operations system are usually very predictable, there is less value to having that coupled object be manipulable in its own right.

A rule is made by compiling its pattern into an appropriate combinator pile, coercing its handler into combinator of a compatible shape, and hooking them together in sequence.

```
(define (make-rule pattern consequent)
  (let ((match-procedure (match:->combinators pattern))
        (bound-variables (procedure-bound-variables consequent)))
    (define (the-rule data succeed fail)
      (or (match-procedure (list data) '()
            ;; A matcher-success continuation
            (lambda (dict n)
              (define (matched-value name)
                (match:value
                 (or (match:lookup name dict)
                     (error "Handler asked for unknown name"
                            name dict))))
              (and (= n 1)
                   (let ((result
                          (apply consequent
                                 (map matched-value
                                      bound-variables))))
                     (and result
                          (succeed result
                                   (lambda () #f)))))))
          (fail)))
    the-rule))
```

For example, the commutative law rule at the beginning of this problem set can be made directly with make-rule thus:

```
(make-rule '(* (? a) (? b))
  (lambda (a b)
    (and (expr<? a b)
         '(* ,a ,b))))
```

Note that the handler (lambda (a b) ...) needs to extract the variables named a and b from the dictionary produced by the matcher combinator built out of the pattern '(* (? a) (? b)). This is arranged by the procedure user-handler->system-handler in pattern-directed-invocation.scm.

Magic Macrology

Compare the rule definition given at the beginning of this probelm set:

```
(rule '(* (? b) (? a))
      (and (expr<? a b)
           '(* ,a ,b)))
```

with what make-rule demands of us:

```
(make-rule '(* (? a) (? b))
  (lambda (a b)
    (and (expr<? a b)
         '(* ,a ,b))))
```

The names a and b are repeated: the occur both in the pattern and in the parameter list of the handler. This is both obnoxious to write and error-prone (one sign of good taste is when those two things coincide!), because we must remember to repeat the names, and we can make a mistake if we repeat them wrong.

This is a case for syntactic abstraction, otherwise known as a macro. The following rather magical object does what we want:

```
(define-syntax rule
  (sc-macro-transformer
   (lambda (form use-env)
     (let ((pattern (cadr form))
           (handler-body (caddr form)))
       `(make-rule
         ,(close-syntax pattern use-env)
         ,(compile-handler handler-body use-env
                           (match:pattern-names pattern)))))))

(define (compile-handler form env names)
  ;; See magic in utils.scm
  (make-lambda names env
    (lambda (env*) (close-syntax form env*))))
```

We can at least partially check this macro with the following magic incantation:

```
(pp (syntax '(rule '(* (? a) (? b))
                   (and (expr<? a b)
                        `(* ,a ,b)))
            (the-environment)))
==>
(make-rule '(* (? a) (? b))
           (lambda (b a) (and (expr<? a b) (list '* a b))))
```

We see that the rule expands into a call to make-rule with the pattern and its handler procedure. This is the expression that is evaluated to make the rule. In more conventional languages macros expand directly into code that is substituted for the macro call. However this process is not referentially transparent, because the macro expansion may use symbols that conflict with the user's symbols. In Scheme we try to avoid this problem, allowing a user to write "hygienic macros" that cannot cause conflicts. However this is a bit more complicated than just substituting one expression for another. We will not try to explain the problems or the solutions here, but we will just use the solutions described in the MIT Scheme reference manual, section 2.11.

A Potential Project

The matcher we have is still not a complete language, in that it does not support namespace scoping and parametric patterns. For example, we cannot write the following pattern, even though it is obvious what it means:

```
(?:pletrec ((palindrome
             (?:pnew (x)
               (?:choice ()
                         ((? x ,symbol?)
                          (?:ref palindrome)
                          (? x))))))
    (?:ref palindrome))
```

This pattern is intended to match only lists of symbols that are palindromes. For this to work in any reasonable way ?:pnew creates fresh lexically-scoped pattern variables that can be referred to only in the body of the ?:pnew.

A fully-worked out pattern language is a neat subsystem to have, but it is not entirely trivial to build. There are also "semantic matchers" that know something about the constants in a data item. One possible nice project is to flesh out these ideas and produce a full pattern language.