6.945 Adventures in Advanced Symbolic Programming

Spring 2009

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.945 Spring 2009
Problem Set 5

Issued: Wed. 4 March 2009 **Due: Wed. 11 March 2009**

Reading: SICP Section 4.4, especially 4.4.4.3 and 4.4.4.4

Code: load.scm, ghelper.scm (same as before), matcher.scm (attached)

Pattern Matching and Instantiation

One of the most important techniques in advanced 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 *, +. More precisely, 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.

In this problem set we will investigate 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. In the next problem set we will study building this into a simple rule system.

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 when we 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: 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 data items eaten from the input data by this matcher. A match combinator returns #f if the match is unsuccessful.

In general, a match combinator that is expected to match a single object, such as a symbol or a list, will eat the car of the data given to it and report that it ate one thing. However, a segment matcher may eat more than one thing, and it must report the number it ate. There are four basic match combinators. Let’s go through them one by one.

The match procedure match:eqv takes a pattern constant and produces a match combinator. It succeeds if and only if the data is a pair whose car is equal (using eqv?) to the pattern constant. If successful, it does not add to the dictionary and absorbs one item from the data.

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

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

A segment variable procedure makes a more interesting combinator.

```
(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)
```

Here the input data must be a list, so we can compute its length. Again, there are two possibilities, either the 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 ((data (car data))
                  (matchers match-combinators)
                  (dictionary dictionary))
           (cond ((pair? matchers)
                  ((car matchers) data dictionary
                      (lambda (new-dictionary n)
                        (lp (list-tail data n)
                            (cdr matchers)
                            new-dictionary))))
                 ((pair? data) #f)
                 (else (succeed dictionary 1))))))
  list-match)
```

The list match procedure takes match combinators and makes a list combinator that matches a list, which is the car of the given data, if and only if the given match combinators eat up all of the elements in the data list. It applies the combinators in succession. When a match combinator succeeds it 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 the other three, 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))
 '()
 list)
;Value: ((x (1 2)) 1)
```

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

```
((match:list (match:eqv 'a) (match:segment 'x) (match:eqv 'b))
 '((a 1 2 b 3))
 '()
 (lambda (d n) d))
;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)
       (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 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 (x y) `(succeed ,x ,y)))
;Value: #f

((match:->combinators '(a ((? b) 2 3) (? b) c))
 '((a (1 2 3) 1 c))
 '()
  (lambda (x y) `(succeed ,x ,y)))
;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 y)
   (pp `(succeed ,dict ,y))
   #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 5.1:

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 number-of-items-eaten)
         (and (= number-of-items-eaten 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))
```

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?))
```

But there are a number of complications here, not the least of which is that the symbols expt and sin are pattern constants, but the symbol exact-positive-integer? must be evaluated to a procedure. Let's worry about that later. First 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. Alternatively, we can make an entirely new matcher procedure, say match:element-restricted, which takes two arguments: the name of a variable and the restriction predicate. It should return an appropriate match combinator.

Problem 5.2:

Choose one of these strategies and make a new matcher procedure or modify an existing matcher procedure so as to implement pattern variables with restrictions.

Demonstrate your modification.

There is still this nasty problem of evaluation of the predicate expression with the compilation of the new matcher syntax, alluded to earlier (viz., exact-positive-integer? in the preceding problem example). One approach is to address the issue in the pattern specification. An alternative approach is to postpone addressing it when the patterns are compiled into the combinators.

Specifically, we could specify the patterns using quasiquote (a.k.a. backquote) when specifying input patterns. For example,

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

Consult the MIT Scheme Reference Manual for details on QUASIQUOTE.

Alternatively, one might invoke EVAL at pattern compile time (i.e., within match:->combinators), postponing resolving the predicate name to its associated procedure until the point where the combinator is generated. For example,

```
(eval ’integer? user-initial-environment)
```

returns the procedure named by the symbol "integer?".

One might also contemplate a point-of-use case where the predicate symbol is not resolved until it is actually needed within the invocation of each generated combinator. This does not lead to a pretty solution.

Problem 5.3:

How do you want to solve this problem? One way is to use the quasiquote mechanism built into Scheme (and every other Lisp). Another way is to do an explicit evaluation in the compiler. I (GJS) like the first way. Explain how this works, and demonstrate it on a real problem.

Explain the considerations you used to make your choice. Feel free to propose (and implement) other alternatives.

Choice is Good

Another 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 ,n)))
;Value: (succeed ((x z)) 1)

((match:->combinators '((? y) (?:choice a b (? x ,string?) (? y ,symbol?) c)))
 '((z z))
 '()
 (lambda (d n) '(succeed ,d ,n)))
;Value: (succeed ((y z)) 1)

((match:->combinators '(?:choice b (? x ,symbol?)))
  '(b)
  '()
  (lambda (x y)
    (pp '(succeed ,x ,y))
    #f))
(succeed () 1)
(succeed ((x b)) 1)
;Value: #f
```

Problem 5.4:

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 5.5

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.

Anonymity is Best

Finally, it is handy to be able to restrict a match sub-expression without having to name it.

For example:

```
  (?:pletrec ((btos
               (?:choice ()
                         (?:restrict ,symbol?)
                         ((?:ref btos) (?:ref btos)))))
    (binary tree of symbols:  (?:ref btos)))
```

...should be able to match each of:

```
    (binary tree of symbols:  ())
    (binary tree of symbols:  (a b))
    (binary tree of symbols:  ((a b) (c ())))
```

...but none of:

```
    (binary tree of symbols:  (a))
    (binary tree of symbols:  ((a b) (c)))
    (binary tree of symbols:  (a 2))
```

Problem 5.6

Implement a new matcher procedure, match:restrict, 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.

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. As you will see later, 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.

```
;;;; Matcher based on match combinators, CPH/GJS style.
;;;     Idea is in Hewitt's PhD thesis (1969).

(declare (usual-integrations))

;;; There are match procedures that can be applied to data items.  A
;;; match procedure either accepts or rejects the data it is applied
;;; to.  Match procedures can be combined to apply to compound data
;;; items.

;;; A match procedure takes a list containing a data item, a
;;; dictionary, and a success continuation.  The dictionary
;;; accumulates the assignments of match variables to values found in
;;; the data.  The success continuation takes two arguments: the new
;;; dictionary, and the number of items absorbed from the list by the
;;; match.  If a match procedure fails it returns #f.

;;; Primitive match procedures:

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

(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)


;;; Support for the dictionary.

(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))
```

```
(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)

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

```
;;; Syntax of matching is determined here.

(define (match:->combinators pattern)
  (define (compile pattern)
    (cond ((match:element? pattern)
           (match:element (match:variable-name pattern)))
          ((match:segment? pattern)
           (match:segment (match:variable-name pattern)))
          ((list? pattern)
           (apply match:list (map compile pattern)))
          (else (match:eqv pattern))))
  (compile pattern))



(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 (matcher pattern)
  (let ((match-combinator (match:->combinators pattern)))
    (lambda (datum)
      (match-combinator
       (list datum)
       '()
       (lambda (dictionary number-of-items-eaten)
         (and (= number-of-items-eaten 1)
              dictionary))))))
```

```
#|
((match:->combinators '(a ((? b) 2 3) 1 c))
 '((a (1 2 3) 1 c))
 '()
  (lambda (x y) `(succeed ,x ,y)))
;Value: (succeed ((b 1)) 1)

((match:->combinators '(a ((? b) 2 3) (? b) c))
 '((a (1 2 3) 2 c))
 '()
  (lambda (x y) `(succeed ,x ,y)))
;Value: #f

((match:->combinators '(a ((? b) 2 3) (? b) c))
 '((a (1 2 3) 1 c))
 '()
  (lambda (x y) `(succeed ,x ,y)))
;Value: (succeed ((b 1)) 1)


((match:->combinators '(a (?? x) (?? y) (?? x) c))
 '((a b b b b b b c))
 '()
 (lambda (x y)
   (pp `(succeed ,x ,y))
   #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

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