10/10 Extremely well done! You are a super hero!

ps3.scm


```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 6.891 PS3
;;;
;;; Dan R. K. Ports <drkp@mit.edu>
;;;
;;; $Date: 2007-02-27 19:23:03 -0500 (Tue, 27 Feb 2007) $ $Rev: 2278 $
;;;
;;; Collaborators: '(iyzhang amdragon)
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(load "matcher.scm")

;;; Some helper functions

(define (assert x . msg)
  (if (not x) (apply error msg)))

(define (assert-equal a b)
  (assert (equal? a b)
          "Test failed"
          "Expected: "
          b
          " got: "
          a)
  #t)


;;;
;;; Problem 1
;;;
;;; The segment variable match combinator (and presumably any other
;;; hypothetical match combinators that could require backtracking)
;;; uses the return value of its success continuation to determine
;;; whether backtracking is necessary. The continuation can return #f
;;; to indicate that, even though the segment variable matched
;;; successfully, the rest of the pattern failed to match, so the
;;; matcher should try a different match for the segment variable. By
;;; always returning #f, we ensure that all possible matches are
;;; tried.
;;;
;;; (15 min)
;;;
```

1/1 Good!

Good!

2/2 Well done!

```
;;;
;;; Problem 2
;;;
;;; We allow match:element to take an optional argument; if a
;;; predicate is specified, the pattern variable can only match
;;; objects that satisfy the predicate.
;;;

(define (match:element variable pred)
  (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))
               (and (or (not pred)
                        (pred (car data)))
                    (succeed (match:bind variable (car data) dictionary)
                             1))))))
  element-match)

(define (match:variable-predicate pattern)
  (if (< (length pattern) 3)
      #f
      (third pattern)))

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


;;; Testing is done using the quasiquote mechanism, as described in
;;; Problem 3.

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

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

Really should test pred here too, so...

OK but can assume this is called only on restricted var patterns, right?

Oops: I see. Nice!

OK.

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

;; (30 min)
```

Good!

1/1

Good!

```
;;;
;;; Problem 3
;;;
;;; We'll use the quasiquote mechanism to solve this problem. The
;;; problem is that if we use a predicate pattern like '(? b odd?), we
;;; wind up with the *symbol* 'odd?, and we need the procedure. Using
;;; a quasiquote expression like '(? b ,odd?), the procedure odd? is
;;; stored in the pattern expression.
;;;
;;; One alternative is to store the symbol 'odd? in the pattern, and
;;; evaluate it at a later time. The problem is that it isn't clear
;;; in which environment to evaluate the predicate name. The suggested
;;; example, evaluating in the user-initial-environment, doesn't seem
;;; like a very good choice since in many usage scenarios the user
;;; will want to use a predicate defined in some other environment,
;;; (perhaps one created by a let, for example).
;;;
;;; For an example, see the test cases in problem 2.
;;;
;;; (15 min)
```

EXCELLENT! IN THE NEXT PROB SET WE RESOLVE THIS W/ MACROS, BUT THAT WAS TOO MUCH MECHANISM ALL AT ONCE FOR THIS ONE.

```
;;;
;;; Problem 4
;;;

(define (match:choice choices)
  (define (choice-match data dictionary succeed)
    (let lp ((choices choices))
      (if (null? choices)
          #f
          (or ((car choices) data dictionary succeed)
              (lp (cdr choices))))))
  choice-match)

;; Gratuitous abstraction
(define (match:choice? pattern)
  (and (pair? pattern)
       (eq? (car pattern) '?:choice)))

(define match:choices cdr)

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

(assert-equal
 ((match:->combinators '(?:choice a b (? x) c))
  '(z)
  '()
  (lambda (d n) `(succeed ,d ,n)))
 '(succeed ((x z)) 1))

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

(assert-equal
 ((match:->combinators '((? y) (?:choice a b (? x ,string?) (? y ,symbol?) c)))
  '((z d))
  '()
  (lambda (d n) `(succeed ,d ,n)))
 #f)

(assert-equal
 (let ((ret '()))
   ((match:->combinators '(?:choice b (? x ,symbol?)))
    '(b)
    '()
    (lambda (x y)
      (set! ret (append ret (list `(succeed ,x ,y))))
      #f))
   ret)
 '((succeed () 1) (succeed ((x b)) 1)))
```

;;; (30 min)

2/2 Good!

(AND (PAIR? data) ...) But OK.

Good!

```
;;;
;;; Problem 5
;;;
;;; It offends my inner theorist to call something letrec if it
;;; doesn't actually introduce scopes, so I'll implement a real letrec
;;; that behaves the way a letrec should.
;;;
;;; Unlike the other matchers, the magic of pletrec happens at compile
;;; time: the pletrec introduces some named patterns into the state,
;;; then compiles the body into a combinator (using the newly created
;;; patterns as appropriate). So there is no match:pletrec.
;;;
;;; We introduce into the compiler a pdictionary that contains a set
;;; of pbindings, each mapping a pvariable to a pvalue. The names
;;; start with 'p's in order to remain consistent with 'pletrec', and
;;; because they are euphonious and fun to say (try it! I'll
;;; wait). pletrec creates a new pdictionary, extending the existing
;;; enviroment with new pbindings for the new variables. The pvalues
;;; for the new pbindings are the compilation of the definitions *in
;;; the new environment*. The output of the pletrec is the compilation
;;; of the body in the new environment.
;;;
;;; The implementation of pletrec is similar to the usual
;;; mutation-based implementation of letrec. The pdictionary is stored
;;; as an alist mapping pvariable names to cells containing the
;;; associated compiled combinators. pletrec first creates a new
;;; pdictionary with the new pvariable names bound to cells containing
;;; unspecified data. It then compiles the definitions of the new
;;; pvariables within the new environment, and mutates the contents of
;;; each cell to the corresponding new combinator. Finally, it
;;; compiles the body in the new environment.
;;;
;;; At compile time, when a ref pattern is encountered, the compiler
;;; looks up the name in the pdictionary to find the appropriate
;;; pbinding. It outputs a combinator that contains a pointer to the
;;; cell containing the pvalue. At evaluation time, this combinator
;;; looks inside the cell, pulls out the combinator that was stored
;;; inside at compile time, and evaluates it.
;;;

(define (match:->combinators pattern)
  (define ((compile pdictionary) pattern)
    (cond ((match:element? pattern)
           (match:element (match:variable-name pattern)
                          (match:variable-predicate pattern)))
          ((match:choice? pattern)
           (match:choice (map (compile pdictionary) (match:choices pattern))))
          ((match:segment? pattern)
           (match:segment (match:variable-name pattern)))
          ((match:pletrec? pattern)
           (let ((new-pdictionary
                  (match:add-empty-pbindings
                   pdictionary
                   (match:pletrec-pbindings pattern))))
             (for-each (lambda (x) (match:update-pbinding!
                                    new-pdictionary
                                    (car x)
                                    ((compile new-pdictionary) (cadr x))))
                       (match:pletrec-pbindings pattern))
             ((compile new-pdictionary) (match:pletrec-pbody
                                         pattern))))
          ((match:ref? pattern)
           (match:ref (match:lookup-pbinding pdictionary
                                             (match:ref-pvariable pattern))))
          ((list? pattern)
           (apply match:list (map (compile pdictionary) pattern)))
          (else (match:eqv pattern))))
  ((compile '()) pattern))

(define (match:add-empty-pbindings pdictionary pbindings)
  (append (map (lambda (x) (cons (car x) (make-cell '()))) pbindings)
          (alist-copy pdictionary)))

(define (match:update-pbinding! pdictionary pvariable pvalue)
  (set-cell-contents! (cdr (assq pvariable pdictionary)) pvalue))

(define (match:lookup-pbinding pdictionary pvariable)
  (let ((pbinding (assq pvariable pdictionary)))
    (if pbinding
        (cdr pbinding)
        (error "Binding not found for match:ref" pvariable))))

(define (match:ref cell)
  (lambda (x y z)
    ((cell-contents cell) x y z)))

(define (match:pletrec? pattern)
  (and (pair? pattern)
       (eq? (car pattern) '?:pletrec)))
(define match:pletrec-pbindings second)
(define match:pletrec-pbody third)

(define (match:ref? pattern)
  (and (pair? pattern)
       (eq? (car pattern) '?:ref)))
(define match:ref-pvariable second)

(assert-equal
 ((match:->combinators
   '(?:pletrec ((odd-even-etc (?:choice () (1 (?:ref even-odd-etc))))
                (even-odd-etc (?:choice () (2 (?:ref odd-even-etc)))))
               (?:ref odd-even-etc)))
  '(())
  '()
  (lambda (d n) '(succeed ,d ,n)))
 '(succeed () 1))

(assert-equal
 ((match:->combinators
   '(?:pletrec ((odd-even-etc (?:choice () (1 (?:ref even-odd-etc))))
                (even-odd-etc (?:choice () (2 (?:ref odd-even-etc)))))
               (?:ref odd-even-etc)))
  '((1 (2 (1 ()))))
  '()
  (lambda (d n) '(succeed ,d ,n)))
 '(succeed () 1))

(assert-equal
 ((match:->combinators
   '(?:pletrec ((odd-even-etc (?:choice () (1 (?:ref even-odd-etc))))
                (even-odd-etc (?:choice () (2 (?:ref odd-even-etc)))))
               (?:ref odd-even-etc)))
  '((1 (2 (1 (2 (1 (2 ()))))))))
  '()
  (lambda (d n) '(succeed ,d ,n)))
 '(succeed () 1))

(assert-equal
 ((match:->combinators
```

```
  '(?:pletrec ((odd-even-etc (?:choice () (1 (?:ref even-odd-etc))))
               (even-odd-etc (?:choice () (2 (?:ref odd-even-etc)))))
              (?:ref odd-even-etc)))
 '((1 (2 (1 (2 (1 (5 ()))))))))
 '()
 (lambda (d n) '(succeed ,d ,n)))
#f)

(assert-equal
 ((match:->combinators
   '(?:pletrec ((seg (?? x)))
               ((?:ref seg) (?:ref seg))))
  '((a b a b))
  '()
  (lambda (d n) '(succeed ,d ,n)))
 '(succeed ((x (a b))) 1))

(assert-equal
 ((match:->combinators
   '(?:pletrec ((seg (?? x)))
               ((?:ref seg) (?:ref seg))))
  '((a b 1 2))
  '()
  (lambda (d n) '(succeed ,d ,n)))
 #f)

;; Scoping tests

(assert-equal
 ((match:->combinators
   '(?:pletrec ((var (? x)))
               (?:pletrec ((var (? x ,odd?)))
                   (?:ref var))))
  '(1)
  '()
  (lambda (d n) '(succeed ,d ,n)))
 '(succeed ((x 1)) 1))

(assert-equal
 ((match:->combinators
   '(?:pletrec ((var (? x)))
               (?:pletrec ((var (? x ,odd?)))
                   (?:ref var))))
  '(2)
  '()
  (lambda (d n) '(succeed ,d ,n)))
 #f)

(assert-equal
 ((match:->combinators
   '(?:pletrec ((var (? x))
                (foo
                 (?:pletrec ((var (? x ,odd?))) (? y))))
               (?:ref var)))
  '(1)
  '()
  (lambda (d n) '(succeed ,d ,n)))
 '(succeed ((x 1)) 1))

(assert-equal
 ((match:->combinators
   '(?:pletrec ((var (? x))
                (foo
                 (?:pletrec ((var (? x ,odd?))) (? y))))
               (?:ref var)))
  '(2)
  '()
  (lambda (d n) '(succeed ,d ,n)))
 '(succeed ((x 2)) 1))

;; (4 hours)
```

YIKES! BUT AWESOME SOLUTION, DUDE!

```
;;;
;;; Problem 6
;;;
;;; Just a straightforward new combinator here that shares a lot of
;;; its code with element matching.
;;;

(define (match:->combinators pattern)
  (define ((compile pdictionary) pattern)
    (cond ((match:element? pattern)
           (match:element (match:variable-name pattern)
                          (match:variable-predicate pattern)))
          ((match:restrict? pattern)
           (match:restrict (match:restrict-predicate pattern)))
          ((match:choice? pattern)
           (match:choice (map (compile pdictionary) (match:choices pattern))))
          ((match:segment? pattern)
           (match:segment (match:variable-name pattern)))
          ((match:pletrec? pattern)
           (let ((new-pdictionary
                  (match:add-empty-pbindings
                   pdictionary
                   (match:pletrec-pbindings pattern))))
             (for-each (lambda (x) (match:update-pbinding!
                                    new-pdictionary
                                    (car x)
                                    ((compile new-pdictionary) (cadr x))))
                       (match:pletrec-pbindings pattern))
             ((compile new-pdictionary) (match:pletrec-pbody
                                         pattern))))
          ((match:ref? pattern)
           (match:ref (match:lookup-pbinding pdictionary
                                             (match:ref-pvariable pattern))))
          ((list? pattern)
           (apply match:list (map (compile pdictionary) pattern)))
          (else (match:eqv pattern))))
  ((compile '()) pattern))

(define (match:restrict? pattern)
  (and (pair? pattern)
       (eq? (car pattern) '?:restrict)))
(define match:restrict-predicate second)

(define (match:restrict pred)
  (define (restrict-match data dictionary succeed)
    (and (pair? data)
         (and (pred (car data)))
         (succeed dictionary 1)))
  restrict-match)

(assert-equal
 ((match:->combinators
   '(?:restrict ,odd?))
  '(1)
  '()
  (lambda (d n) '(succeed ,d ,n)))
 '(succeed () 1))

(assert-equal
 ((match:->combinators
   '(?:restrict ,even?))
  '(1)
  '()
  (lambda (d n) '(succeed ,d ,n)))
 #f)

(assert-equal
 ((match:->combinators
 '(?:pletrec ((btos
               (?:choice ()
                         (?:restrict ,symbol?)
                         ((?:ref btos) (?:ref btos)))))
             (binary tree of symbols: (?:ref btos))))
 '((binary tree of symbols: ()))
 '()
 (lambda (d n) '(succeed ,d ,n)))
 '(succeed () 1))

(assert-equal
 ((match:->combinators
 '(?:pletrec ((btos
               (?:choice ()
                         (?:restrict ,symbol?)
                         ((?:ref btos) (?:ref btos)))))
             (binary tree of symbols: (?:ref btos))))
 '((binary tree of symbols: (a b)))
 '()
 (lambda (d n) '(succeed ,d ,n)))
 '(succeed () 1))

(assert-equal
 ((match:->combinators
 '(?:pletrec ((btos
               (?:choice ()
                         (?:restrict ,symbol?)
                         ((?:ref btos) (?:ref btos)))))
             (binary tree of symbols: (?:ref btos))))
 '((binary tree of symbols: ((a b) (c ()))))
 '()
 (lambda (d n) '(succeed ,d ,n)))
 '(succeed () 1))

(assert-equal
 ((match:->combinators
 '(?:pletrec ((btos
               (?:choice ()
                         (?:restrict ,symbol?)
                         ((?:ref btos) (?:ref btos)))))
             (binary tree of symbols: (?:ref btos))))
 '((binary tree of symbols: (a)))
 '()
 (lambda (d n) '(succeed ,d ,n)))
 #f)

(assert-equal
 ((match:->combinators
 '(?:pletrec ((btos
               (?:choice ()
                         (?:restrict ,symbol?)
                         ((?:ref btos) (?:ref btos)))))
             (binary tree of symbols: (?:ref btos))))
 '((binary tree of symbols: ((a b) (c))))
 '()
 (lambda (d n) '(succeed ,d ,n)))
 #f)

(assert-equal
 ((match:->combinators
 '(?:pletrec ((btos
```

2/2 Well done!

Yup. We addressed that too in the next prob set. Good eye!

≡ (pred (car data)) right?

Good!

```
                (?:choice ()
                          (?:restrict ,symbol?)
                          ((?:ref btos) (?:ref btos)))))
              (binary tree of symbols:  (?:ref btos))))
'((binary tree of symbols:  (a 2)))
'()
(lambda (d n) '(succeed ,d ,n)))
#f)

;; (15 min)
```

Good!