10/10 WELL DONE


```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 6.891 PS5
;;;
;;; Dan R. K. Ports <drkp@mit.edu>
;;;
;;; $Date: 2007-03-14 12:07:31 -0400 (Wed, 14 Mar 2007) $ $Rev: 2292 $
;;;
;;; Collaborators: '(iyzhang amdragon)
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(load "load.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
;;;
;;; Here's an impementation of the choice extension.
;;;

;; Match combinator
(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)


;; Compiler
(define (compile-match-choice pattern use-env loop)
  `(match:choice (list ,@(map (lambda (x) (compile-pattern x use-env))
                              (match:choices pattern)))))

(eq-put! '?:choice 'pattern-keyword compile-match-choice)


;; Test cases
(let ((test
       (rule-simplifier
        (list
         (rule (?:choice foo bar)
               none
               matched)))))
  (assert-equal (test 'foo) 'matched)
  (assert-equal (test 'bar) 'matched)
  (assert-equal (test 'baz) 'baz))

(let ((test
       (rule-simplifier
        (list
         (rule (?:choice (? x number?) (? x string?))
               none
               matched)))))
  (assert-equal (test '(4 "foo" bar)) '(matched matched bar)))
```

```
;;;
;;; Problem 2
;;;
;;; We need to close the match restriction in the usage syntactic
;;; environment since the match restriction could be a name that
;;; hasn't been bound at the time of transformation. For example, if
;;; we did something like
;;;
;;;   (let ((frobble? number?))
;;;     (define foo
;;;       (rule-simplifier
;;;        (list
;;;         (rule (+ (? a frobble?)) none (: a))))))
;;;
;;; then without the close-syntax, frobble? would be looked up in the
;;; syntactic environment of the transformer, where it has no
;;; meaning. Similarly, if we redefined an existing name, the old
;;; value would be used instead.
```

— 2/2

Excellent!

3/3

```
;;;
;;; Problem 3
;;;

;; Compiler
(define (compile-match-restrict pattern use-env loop)
  (let ((inside (loop (match:restriction-pattern pattern))))
    `(match:restrict ,inside
                     ,(close-in-dictionary
                       (match:restriction-restriction pattern)
                       use-env))))

(eq-put! '?r 'pattern-keyword compile-match-restrict)

(define (match:restriction-pattern pattern)
  (cadr pattern))

(define (match:restriction-restriction pattern)
  (caddr pattern))

;; Combinator
(define (match:restrict match-combinator restriction?)
  (define (restrict-match data dictionary succeed)
    (match-combinator data dictionary
                      (lambda (dictionary n)
                        (if (restriction? dictionary)
                            (succeed dictionary n)
                            #f))))
  restrict-match)

;; Test cases
(let ((test
       (rule-simplifier
        (list
         (rule (?r (? x number?)
                   (> (: x) 0))
               none
               matched)))))
  (assert-equal (test 5) 'matched)
  (assert-equal (test -5) '-5))

(let ((test
       (rule-simplifier
        (list
         (rule (?r (+ (* (? a number?) (expt (? x) 2))
                      (* (? b number?) (? x))
                      (? c number?))
                   (= (expt (: b) 2) (* 4 (: a) (: c))))
               none
               matched)))))
  (assert-equal (test '(+ (* 1 (expt x 2))
                          (* 2 x)
                          1)) 'matched)
  (assert-equal (test '(+ (* 1 (expt x 2))
                          (* 2 x)
                          0))
                '(+ (* 1 (expt x 2))
                    (* 2 x) 0)))
```

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

(define (skel:transform? skeleton)
  (and (pair? skeleton)
       (eq? (car skeleton) ':t)))

(define (skel:transform-transformer skeleton)
  (cadr skeleton))

(define (skel:transform-expression skeleton)
  (caddr skeleton))

(define (compile-instantiate-transform skel use-env loop)
  `(instantiate:transform
    ,(close-syntax (skel:transform-transformer skel) use-env)
    ,(loop (skel:transform-expression skel))))

(eq-put! ':t 'template-keyword compile-instantiate-transform)

(define (instantiate:transform transformer expression)
  (lambda (dictionary continue)
    (expression dictionary
                (lambda (elements number)
                  (continue (transformer elements) #f)))))


;; Test cases
(let ((test
       (rule-simplifier
        (list
         ;; Replace numbers with 'number. No idea why you'd want to do
         ;; this at all, let alone with a transformer.
         (rule (? x number?)
               none
               (:t (lambda (x) 'number) (: x)))))))
  (assert-equal (test '(+ -1 2)) '(+ number number)))

(let ((test
       (rule-simplifier
        (list
         ;; Associative law with sorting for multiplication
         (rule (* (?? a) (* (?? b)) (?? c))
               none
               (:t (lambda (x) (cons '* x))
                   (:t (lambda (x) (sort x expr<?))
                       (,@(: a) ,@(: b) ,@(: c)))))))))
  (assert-equal (test '(* 4 (* 5 1)))
                '(* 1 4 5))
  (assert-equal (test '(* 4 (* 5 1) (* 2 7 (* 9 3 8 6))))
                '(* 1 2 3 4 5 6 7 8 9)))

(let ((test
       (rule-simplifier
        (list
         ;; Replace negative numbers with their absolute value
         (rule (?r (? x number?)
                   (< (: x) 0))
               none
               (:t - (: x)))))))
  (assert-equal (test 5) 5)
  (assert-equal (test -5) 5))
```

Good!