Quiz 3  
Symbols, Environment, Dispatch

Before starting, please write your name in the first blank below, and *optionally* a guess as to how well you think you will do in the second. Start the quiz only when instructed to do so. You may use any written resources you wish, but you may not consult another student, nor use a computer, nor a calculator. You will have two hours to finish this quiz, at which point please close the document, and *optionally* re-assess your anticipated grade in the third blank below. Please give your tests to the staff as you leave. Your actual grade will not be affected by your self-assessment, nor by opting out of self-assessment.

Name: ____________________________

Optional, expected grade (percent correct) before taking quiz: ____________________________

Optional, expected grade (percent correct) after taking quiz: ____________________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>Score</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>E.C.</td>
<td>+10</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100 (+10)</td>
<td></td>
</tr>
</tbody>
</table>
Introduction  One of the central fixtures in the arsDigita University facility is the soda fridge. The quiz will be centered around a program to control a robot that will manage the soda fridge for us, doing things like fetching sodas upon request, insuring that warm sodas (in the cabinets) gets shifted to the fridge when necessary, and the like.

Problem 1: 4 points  Here is the abstraction for soda that we will be using, and a sample call to create a soda. Each flavor of soda (not each can) will have a separate entry in the inventory using this abstraction, as we will see shortly.

(define (make-soda name num-in-fridge num-in-cabinet sugar caffeine)
  (let ((soda (list name num-in-fridge num-in-cabinet sugar caffeine))
        (lambda (m)
          (cond ((eq? m 'name) (car soda))
                ((eq? m 'num-in-fridge) (e1))
                ((eq? m 'num-in-cabinet) (e2))
                ((eq? m 'sugar) (e3))
                ((eq? m 'caffeine) (e4))
                (else (error "Unrecognized request" m))))))

(define example-soda-1 (make-soda 'coke 20 30 'yes 'yes))
(define example-soda-2 (make-soda 'poland-springs-lime 10 0 'no 'no))

Determine what the four missing fragments of code above, \(e_1\) through \(e_4\), should be. Make sure the interface to a soda object is uniform in usage.

\[
\begin{array}{c}
\langle e_1 \rangle: & (\text{cadr soda}) & \text{or} & (\text{list-ref 1 soda}) \\
\langle e_2 \rangle: & (\text{caddr soda}) & \text{or} & (\text{list-ref 2 soda}) \\
\langle e_3 \rangle: & (\text{caddr soda}) & \text{or} & (\text{list-ref 3 soda}) \\
\langle e_4 \rangle: & (\text{caddr (cdr soda)}) & \text{or} & (\text{list-ref 4 soda})
\end{array}
\]

Problem 2: 2 points  Does evaluating \(\text{make-soda 'coke 20 30 'yes 'yes}\) return a procedure object or a list? Write \(P\) for procedure or \(L\) for list.

\[
P
\]

It was very important to understand that the returned value here was a procedure.
Problem 3: 10 points  We want to have predicates on our soda objects. Finish the following code fragments. The procedure cold? should take a soda and return #t if and only if the number of cans in the fridge for that soda is positive (assume all of the cans in the fridge are cold, and all in the cabinet are warm). Similarly, caffeine? should return #t if and only if the caffeine field of soda is set to the symbol yes. Do not break the abstraction barrier for soda.

(define (cold? soda) (e₀))
(define (caffeine? soda) (e₀))
(define (caffeine-free? soda) (e₁))
(define (sugar? soda) (e₂))
(define (sugar-free? soda) (e₃))

Determine what the five missing fragments of code above, (e₀) through (e₃), should be.

\[
\begin{align*}
\langle e₀ \rangle & : \quad (< 0 \ (\text{soda 'num-in-fridge})) \\
\langle e₁ \rangle & : \quad (\text{eq? } \ (\text{soda 'caffeine}) \ \text{'yes}) \\
\langle e₂ \rangle & : \quad (\text{eq? } \ (\text{soda 'caffeine}) \ \text{'no}) \\
\langle e₃ \rangle & : \quad (\text{eq? } \ (\text{soda 'sugar}) \ \text{'yes}) \\
\langle e₄ \rangle & : \quad (\text{eq? } \ (\text{soda 'sugar}) \ \text{'no})
\end{align*}
\]

NOTE: the remainder of the quiz requires that you have full understanding of the correct answer to this question. If you do not feel completely confident with your answer, you may request an answer sheet for this question. However, in taking the solution, you will receive no points for this question.
Problem 4: 2 points  The set of soda objects (recall, one per flavor of soda) will be collected together to represent the total inventory. Here is the abstraction we will use for inventory and an example use.

(define (make-inventory . sodas)  sodas)
(define (inventory-add-entry soda inventory)  (cons soda inventory))
(define (inventory-first-entry inventory)  (car inventory))
(define (inventory-rest-entries inventory)  (cdr inventory))
(define (inventory-empty? inventory)  (null? inventory))

(define example-inventory
  (make-inventory
    (make-soda 'coke 20 50 'yes 'yes)
    (make-soda 'diet-coke 15 23 'no 'yes)
    (make-soda 'root-beer 25 10 'yes 'no)
    (make-soda 'poland-springs-lime 0 0 'no 'no)
    (make-soda 'dr-pepper 50 60 'yes 'yes)))

What is the underlying representation of the inventory data abstraction?

list

Problem 5: 2 points  What is the underlying representation of the objects we are placing in the example-inventory above?

procedure object
Problem 6: 20 points Assume a small inventory has been created containing Coke and Diet Coke. Complete the environment diagram that will result from executing the following three expressions (for convenience, these duplicate some of the code you have seen previously).

(defun (make-soda name num-in-fridge num-in-cabinet sugar caffeine)
  (let ((soda (list name num-in-fridge num-in-cabinet sugar caffeine)))
    (lambda (m)
      (cond ((eq? m 'name) ...)
            ((eq? m 'num-in-fridge) ...)
            ((eq? m 'num-in-cabinet) ...)
            ((eq? m 'sugar) ...)
            ((eq? m 'caffeine) ...)
            (else (error "Unrecognized request" m))))))

(defun make-inventory . sodas) ...)
(defun small-inventory
  (make-inventory
   (make-soda 'coke 20 30 'yes 'yes)
   (make-soda 'diet-coke 15 17 'no 'yes)))

On the last page of the quiz, you will find an incomplete environment diagram. Tear out that page and use it as a reference to provide values for all of the entries that appear in angle brackets (e.g. ⟨1⟩, ⟨2⟩, ...). The value for each entry should be taken from the object titles to the upper left of each object (e.g. GE, E1, E2, ..., P1, P2, ..., L1, L2, ...). Note that the sequencing of object titles has no relation to the order in which the objects were created.

⟨1⟩: \[ P3 \]  \hspace{2cm}  ⟨8⟩: \[ P1 \mid P2 \]  

⟨2⟩: \[ P4 \]  \hspace{2cm}  ⟨9⟩: \[ GE \]  

⟨3⟩: \[ L3 \]  \hspace{2cm}  ⟨10⟩: \[ GE \]  

⟨4⟩: \[ E5 \]  \hspace{2cm}  ⟨11⟩: \[ E1 \mid E2 \]  

⟨5⟩: \[ E4 \]  \hspace{2cm}  ⟨12⟩: \[ E1 \mid E2 \]  

⟨6⟩: \[ GE \]  \hspace{2cm}  ⟨13⟩: \[ GE \]  

⟨7⟩: \[ P1 \mid P2 \]  \hspace{2cm}  ⟨14⟩: \[ GE \]  

There were two consistent answers possible here; the one formed with the left-hand values (for entries with two values), and the one formed with the right-hand values.
Problem 7: 15 points Write `find-soda-by-name` that takes the name of a soda (a symbol) and an inventory, returning the soda object if it finds a match, and `nil` otherwise. Use the selector functions from inventory. Do not break the abstraction barriers.

```
(define (find-soda-by-name soda-name inventory)
  (cond ((inventory-empty? inventory)
         nil)
        ((eq? soda-name (inventory-first-entry inventory) 'name)
         (inventory-first-entry inventory))
        (else
         (find-soda-by-name soda-name (inventory-rest-entries inventory))))
```

Or, if you were concerned about the extra call to `inventory-first-entry`, you could insert a `let` as follows.

```
(define (find-soda-by-name soda-name inventory)
  (if (inventory-empty? inventory)
      nil
      (let ((entry (inventory-first-entry inventory)))
        (if (eq? soda-name (entry 'name))
            entry
            (find-soda-by-name soda-name (inventory-rest-entries inventory))))))
```

Problem 8: 5 points If you were to break the abstraction barrier for the inventory data structure while writing `find-soda-by-name`, there is an elegant solution available. On what higher-order function that we have seen in lecture and the text many times does this solution depend? Write the procedure name alone; do not give the full solution.

```
filter
```

The intended solution would look something like this:

```
(define (find-soda-by-name soda-name inventory)
  (filter (lambda (soda-entry) (eq? soda-name (soda-entry 'name))) inventory))
```

A solution using `accumulate` might look like this:

```
(define (find-soda-by-name soda-name inventory)
  (accumulate append nil (lambda (soda-entry) (eq? soda-name (soda-entry 'name))) inventory))
```

but, this depends on `append` swallowing intermediate empty lists (e.g., `(append nil (append nil (append '(coke ...) nil))) => (coke ...))`, rather than picking out the exact desired entry, as `filter` is intended to do.
Problem 9: 15 points  Write a procedure called any-cold-soda? which takes an inventory and returns a list of the names of all of the sodas with positive (non-zero) stock in the fridge. If no there are no sodas in the fridge at all, your procedure should return nil. Do not break the abstraction barriers.

;;; ANY-COLD-SODA?
;;;
;;; Returns a list of names of sodas in an inventory that have some cold.
(define (any-cold-soda? inventory)
  (cond ((inventory-empty? inventory) nil)
        ((cold? (inventory-first-entry inventory))
            (cons ((inventory-first-entry inventory) 'name)
                  (any-cold-soda? (inventory-rest-entries inventory))))
        (else
            (any-cold-soda? (inventory-rest-entries inventory))))

Or, you could write a filter operator for inventory and use that instead (contrast this version with a similar function that appears later in the solutions).

;;; ANY-COLD-SODA?
;;;
;;; Returns a list of names of sodas in an inventory that have some cold.
(define (any-cold-soda? inventory)
  (inventory-filter (lambda (soda) (cold? soda)) inventory))

;;; INVENTORY-FILTER
;;;
;;; Implements a FILTER operation for INVENTORY data structures. This version is above the data abstraction.
(define (inventory-filter pred inventory)
  (cond ((inventory-empty? inventory) nil)
        ((pred (inventory-first-entry inventory))
            (cons (inventory-first-entry inventory)
                  (inventory-filter pred (inventory-rest-entries inventory))))
        (else
            (inventory-filter pred (inventory-rest-entries inventory))))
Problem 10: 10 points  Add the take-one-from-fridge method to make-soda. It should decrement the number of sodas in the fridge by 1 if there are enough in the fridge to do so. If a soda is successfully removed, take-one-from-fridge should return #t, otherwise, it should return #f. You do not need to duplicate the full function, just write the new lines and indicate where they would be placed: use the line numbers to the right of the code below and label your code with, e.g., “insert after line XXX”.

```
(define (make-soda name num-in-fridge num-in-cabinet sugar caffeine) ; (1)
  (let ((soda (list name num-in-fridge num-in-cabinet sugar caffeine))) ; (2)
    (lambda (m) ; (3)
      (cond ((eq? m 'name) (...) ; (4)
             ((eq? m 'num-in-fridge) ...) ; (5)
             ((eq? m 'num-in-cabinet) ...) ; (6)
             ((eq? m 'sugar) ...) ; (7)
             ((eq? m 'caffeine) ...) ; (8)
             (else (error "Unrecognized request" m)))))
    )
    )

;;; after line 8
((eq? m 'take-one-from-fridge)
  ;; We are UNDER the abstraction barrier, thus we use CAR, CDR, etc
  ;; on the SODA list value. We could, potentially, modify the value
  ;; NUM-IN-FRIDGE, but changes there would not be reflected in our
  ;; methods for, e.g., 'NUM-IN-FRIDGE which access the values in the
  ;; SODA list.
  (if (< 0 (cadr soda)) ; do we have any cold ones?
    (begin ; yes, remove one of them
      (set-car! (cdr soda) (dec (cadr soda)))
      #t ; successful decrement!
    #f) ; unsuccessful decrement.
  )
)
Problem 11: 15 points  Write get-soda, a procedure that takes the name of a soda (that is, a symbol) and an inventory. If there is a cold soda of the requested type available, get-soda should return (list 'cold name). If there is no cold soda available, but there is one in the cabinet, it should return (list 'warm name). If there is no soda of that type available, it should return (list 'no name 'available). Assume that make-soda now includes the take-one-from-fridge method that you defined above and a similar method called take-one-from-cabinet.

```
(define (get-soda name inventory)
  (let ((soda (find-soda-by-name name inventory)))
    (cond ((null? soda)
        (list 'no name 'available)) ; could be (list 'we 'don't 'carry name)
             ((soda 'take-one-from-fridge) ; attempts decrement, returns #t if success
              (list 'cold name))
             ((soda 'take-one-from-cabinet) ; attempts decrement, returns #t if success
              (list 'warm name))
             (else
              (list 'no name 'available))))))
```
EXTRA CREDIT: 10 points  Write the procedure call-justin-if-the-fridge-is-getting-empty, a procedure of one argument, inventory, that invokes the procedure call-justin when at least half of the entries in the inventory have no cold stock available. Assume call-justin restocks the fridge and updates the inventory values.

;;; CALL-JUSTIN-IF-THE-FRIDGE-IS-GETTING-EMPTY
;;; (define (call-justin-if-the-fridge-is-getting-empty inventory)
  (let ((num-soda-types-with-cold-stock
    (inventory-accumulate (lambda (x) (if (cold? s) 1 0)) 0 + inventory))
        (num-soda-types
        (inventory-accumulate (lambda (x) 1) 0 + inventory))
        (if (<= num-soda-types (* 2 num-soda-types-with-cold-stock))
          (call-justin inventory)
          (leave-justin-alone-he-has-plenty-to-do)))))

;;; INVENTORY-ACCUMULATE
;;; (;; This one will be BELOW the abstraction barrier. That means we are, by
;;; ;; executive fiat, extending the INVENTORY data abstraction, and thus we will
;;; ;; have the underlying representation at our disposal. Contrast this with
;;; ;; INVENTORY-FILTER from above.
;;; (define (inventory-accumulate term null-value combiner inventory)
;;;    ;; see the R4 Report for REDUCE
;;;    (reduce (lambda (val elt) (combiner val (term elt))) null-value inventory))

However, a clever modification of our algorithm will make the first procedure much simpler (look at the term function passed to inventory-accumulate.

;;; CALL-JUSTIN-IF-THE-FRIDGE-IS-GETTING-EMPTY
;;; (define (call-justin-if-the-fridge-is-getting-empty inventory)
  (let ((excess-soda-types-with-cold-stock
    (inventory-accumulate (lambda (x) (if (cold? s) 1 -1)) 0 + inventory))
        (if (<= 0 excess-soda-types-with-cold-stock)
          (call-justin inventory)
          (leave-justin-alone-he-has-plenty-to-do))))
TEAR OUT THIS PAGE FOR USE WITH PROBLEM 6.

Note that P1 and P2 are interchangeable (and both should be used). Also note that object numbers have no relation to the order in which they were created.