arsDigita University

Structure and Interpretation of Computer Programs
October 2000

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: ____________

Problem	*Score*	*Value*
1		4
2		2
3		10
4		2
5		2
6		20
7		15
8		5
9		15
10		10
11		15
E.C.		+10
Total		100 (+10)

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, $\langle e_1 \rangle$ through $\langle e_4 \rangle$, should be. Make sure the interface to a soda object is uniform in usage.

$\langle e_1 \rangle$: ____________________

$\langle e_2 \rangle$: ____________________

$\langle e_3 \rangle$: ____________________

$\langle e_4 \rangle$: ____________________

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

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)           ⟨e5⟩)
(define (caffeine? soda)       ⟨e6⟩)
(define (caffeine-free? soda) ⟨e7⟩)
(define (sugar? soda)          ⟨e8⟩)
(define (sugar-free? soda)    ⟨e9⟩)
```

Determine what the five missing fragments of code above, $\langle e_5 \rangle$ through $\langle e_9 \rangle$, should be.

$\langle e_5 \rangle$: ____________________

$\langle e_6 \rangle$: ____________________

$\langle e_7 \rangle$: ____________________

$\langle e_8 \rangle$: ____________________

$\langle e_9 \rangle$: ____________________

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?

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

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).

```
(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)           ...)
            ((eq? m 'num-in-fridge)  ...)
            ((eq? m 'num-in-cabinet) ...)
            ((eq? m 'sugar)          ...)
            ((eq? m 'caffeine)       ...)
            (else (error "Unrecognized request" m))))))

(define (make-inventory . sodas) ...)

(define 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⟩:	________	⟨8⟩:	________
⟨2⟩:	________	⟨9⟩:	________
⟨3⟩:	________	⟨10⟩:	________
⟨4⟩:	________	⟨11⟩:	________
⟨5⟩:	________	⟨12⟩:	________
⟨6⟩:	________	⟨13⟩:	________
⟨7⟩:	________	⟨14⟩:	________

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.

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.

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.

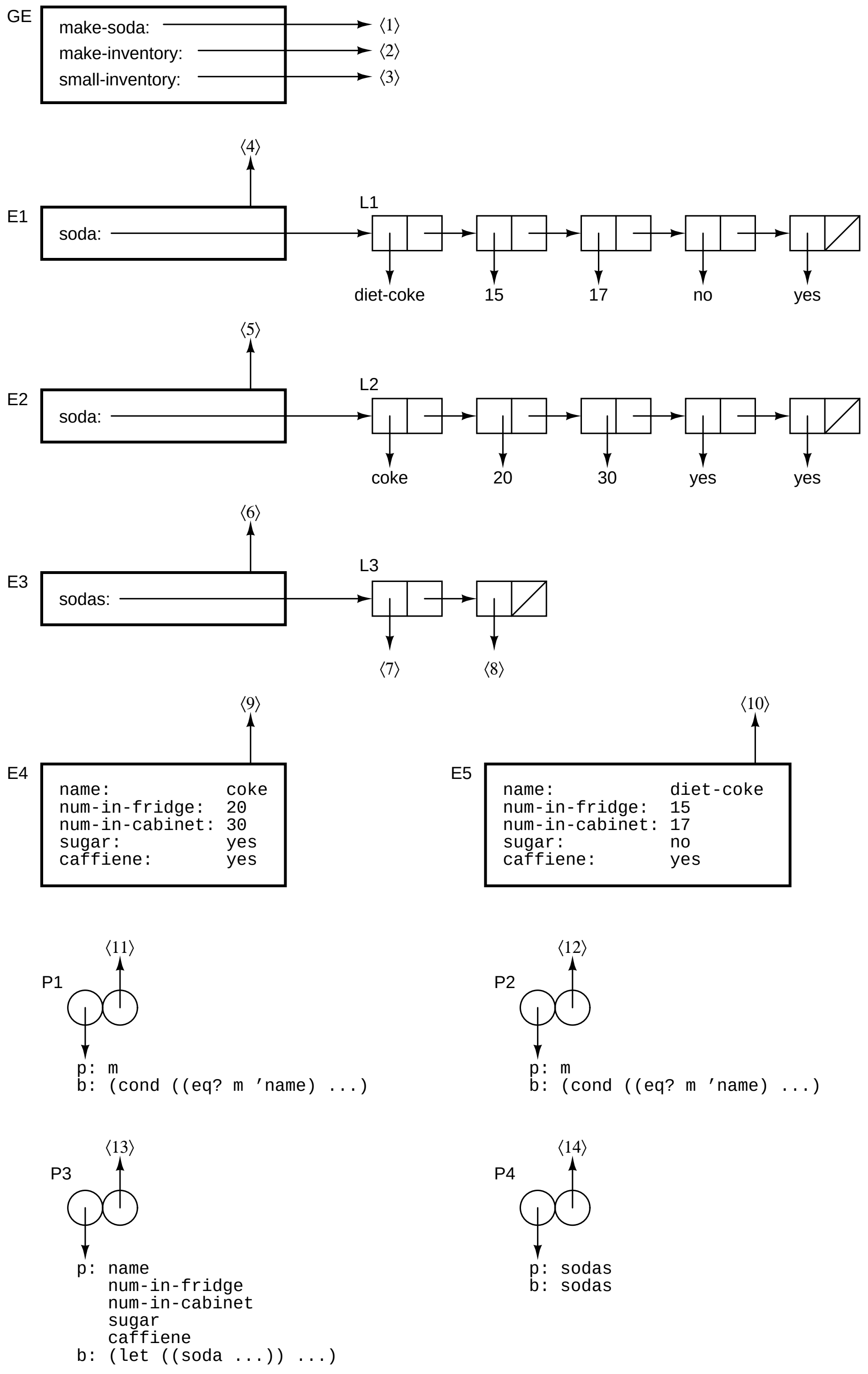
Problem 10: 10 points Add the `take-one-from-fridge` method to `make-soda`. It should decrement the number of sodas in the fridge by 1 if there are enough in the fridge to do so. If a soda is successfully removed, `take-one-from-fridge` should return `#t`, otherwise, it should return `#f`. You do not 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))))))                  ; (9)
```

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`.

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.

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.