Structure and Interpretation of Computer Programs
October 2000

Quiz 2
Scheme Data Structures

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: _____

P1 _____ P2 _____ P3 _____ P4 _____ P5 _____ P6 _____ P7 _____ Total _____

Problem 1: 25 points A local bookstore has contracted aD University to provide an inventory system for their web site. We can create a database of books using Scheme. The constructor for a single book will be called `make-book` and takes the name of a book and its price as parameters.

```
(define (make-book name price)
  (cons name price))
```

Write the selectors `book-name` and `book-price`.

The inventory of books will be stored in a list. The selectors for our inventory data structure are `first-book` and `rest-books`, defined as follows:

```
(define first-book car)
(define rest-books cdr)
```

Write the constructor `make-inventory`.

Draw the box-and-pointer diagram that results from the evaluation of

```
(define store-inventory
  (make-inventory (make-book 'sicp 60)
                  (make-book 'collecting-pez 15)
                  (make-book 'the-little-schemer 35)))
```

Problem 1 (continued) Write a procedure called `find-book` which takes the name of a book and an inventory as parameters and returns the book's data structure (name and price) if the book is in the store's inventory, and nil otherwise.

The bookstore has asked us to change our system to include a count of the number of copies of each book the store has on hand. We redefine our book constructor as follows:

```
(define (make-book name price num-in-stock)
  (list name price num-in-stock))
```

Write the selectors `book-name`, `book-price`, and `book-stock` for our new constructor.

Will `find-book` need to be changed to accommodate our new representation? _____

Problem 1 (continued) Now that we are storing the number of copies in stock, write a procedure called `in-stock?` that takes a book name and an inventory as the parameters, and returns `#t` if at least one copy of the book is in stock, or `#f` otherwise. If the book is not listed in the inventory at all, `in-stock?` should also return `#f`. You may want to use your `find-book` procedure from above.

Problem 2: 20 points Write a function `add-n` of one argument `n` that returns a procedure. The returned procedure takes one argument `x` and returns the sum of `x` and `n`.

Using `add-n`, and without using the built-in Scheme procedure `*`, write `mult` which takes two integer arguments `a` and `b` and returns their product.

Problem 3: 10 points Assume the following expressions have been evaluated in the order they appear.

```
(define a (list (list 'q) 'r 's))
(define b (list (list 'q) 'r 's))
(define c a)
(define d (cons 'p a))
(define e (list 'p (list 'q) 'r 's))
```

Complete the table below with the result of applying the functions `eq?`, `eqv?`, and `equal?` to the two expressions on the left of each row. For example, the elements of the top row will represent the result from evaluating `(eq? a c)`, `(eqv? a c)`, and `(equal? a c)`. Your result should be written as `#t`, `#f` or *undefined*.

$\langle operand_1 \rangle$	$\langle operand_2 \rangle$	`eq?`	`eqv?`	`equal?`
`a`	`c`			
`a`	`b`			
`a`	`(cdr d)`			
`d`	`e`			
`(car a)`	`(car e)`			
`(car a)`	`(cadr e)`			
`(caar a)`	`(caadr e)`			

Problem 4: 15 points Write `occurrences`, a procedure of two arguments `s` and `tree` that returns the number of times the first argument (an atom) appears in the second (a tree). You may find `accumulate-tree`, shown below, to be helpful.

```
(define (accumulate-tree tree term combiner null-value)
   (cond ((null? tree) null-value)
         ((not (pair? tree)) (term tree))
         (else (combiner (accumulate-tree (car tree) term combiner null-value)
                         (accumulate-tree (cdr tree) term combiner null-value)))))
```