

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: _____ *Solutions*

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`.

```
(define (book-name b) (car b))
(define (book-price b) (cdr b))
```

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`.

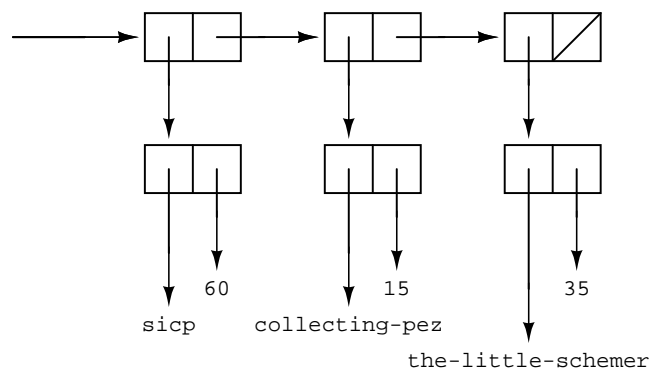
Any of

```
(define make-inventory list)
(define (make-inventory . books) books)
(define make-inventory (lambda (books) books))
```

would work.

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.

```
(define (find-book name inventory)
  (if ((null? inventory) nil)
      ((eqv? (book-name (first-book inventory)) name)
       (first-book inventory))
      (else
       (find-book name (rest-books inventory)))))
```

Points were commonly lost here for breaking the abstraction barriers and for passing a book instead of the book's name.

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.

```
(define book-name car)
(define book-price cadr)
(define book-stock caddr)
```

Will `find-book` need to be changed to accommodate our new representation? no (unless barriers were broken above)

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.

```
(define (in-stock? name inventory)
  (let ((book (find-book name inventory)))
    (and book
          (> 0 (book-stock book)))))
```

Notice the use of `and` instead of `if`; remember that `and` evaluates its arguments in left-to-right order, each in turn, and only until the first evaluated argument is false (or it runs out of arguments). If all arguments evaluate non-false, then the value returned is the value of the last argument. If you used `if`, that would have been just fine.

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`.

```
(define (add-n n)
  (lambda (x) (+ x 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.

There were three kinds of answers to this question. The first was along the lines of the formulation of `new-add` from the first quiz; the second used `repeated` (which worked only for positive values for the variable chosen for iteration, and thus lost points); the third was a very nice recursive formulation to deal with negative numbers. The three approaches are shown below.

```
(define (mult a b)
  (let ((op-b (add-n b)) ; save the operator for b
        (op-i (if (< a 0) dec inc)) ; same for the iterator
        (define (m-helper i prod) ; the helper function to iterate
              (if (= i a) ; are we done?
                  prod ; yes, return the answer
                  (m-help (op-i i) (op-b prod)))) ; no, advance counter and result
        (m-helper 0 0))) ; start out at zero
```

```
(define (mult a b) ; works only for non-negative a
  ((repeated (add-n a) b) 0)) ; add b to 0 a times
```

```
(define (mult a b)
  (cond ((= a 0) 0) ; done?
        ((< a 0) ; if negative a
         (- (mult (- a) b))) ; then flip and continue
        ((add-n b) (mult (dec a) b)))) ; iterate
```

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* [or *unspecified*].

$\langle operand_1 \rangle$	$\langle operand_2 \rangle$	<code>eq?</code>	<code>eqv?</code>	<code>equal?</code>
a	c	<code>#t</code>	<code>#t</code>	<code>#t</code>
a	b	<code>#f</code>	<code>#f</code>	<code>#t</code>
a	(cdr d)	<code>#t</code>	<code>#t</code>	<code>#t</code>
d	e	<code>#f</code>	<code>#f</code>	<code>#t</code>
(car a)	(car e)	<code>#f</code>	<code>#f</code>	<code>#f</code>
(car a)	(cadr e)	<code>#f</code>	<code>#f</code>	<code>#t</code>
(caar a)	(caadr e)	<code>#t</code>	<code>#t</code>	<code>#t</code>

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))))))
```

Many, but not all of the students used `accumulate-tree` in their answer, as suggested by the hint.

```
(define (occurrences s tree)
  (accumulate-tree tree
                   (lambda (x) (if (eqv? x s) 1 0))
                   +
                   0))
```

Those who did not use `accumulate-tree` ended up writing a procedure with the same functionality.

Problem 5: 15 points Two images are shown in Figure 1. On the left is Happy Stick Man, waving at us. On the right is a composite image created with the Henderson Picture Language from Problem Set 4.

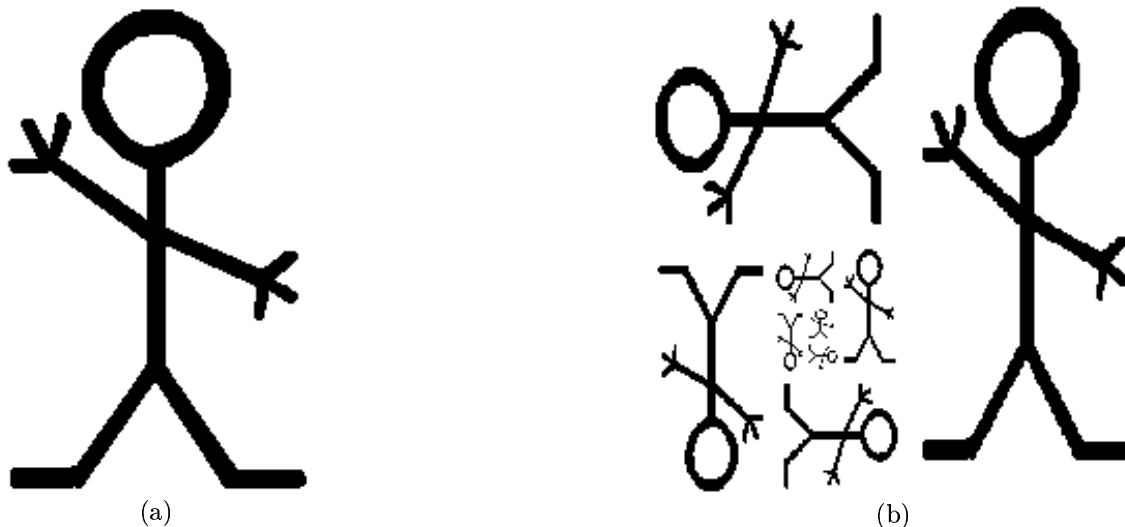


Figure 1: (a) Image of happy-stick-man. (b) Image of (spiral happy-stick-man 4)

Assume that happy-stick-man contains a painter to generate the left hand image above in Figure 1. Finish the definition of spiral below so that evaluating

```
(paint graphics-window (spiral happy-stick-man 4))
```

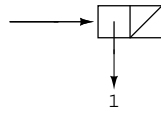
will generate the right hand image in Figure 1. You may use any of the procedures from the problem set, including beside, below and the rotation procedures.

```
(define (spiral painter n)
  (if (= n 0)
      painter
      (beside (below (rotate180 (spiral painter (- n 1)))
                    (rotate90 painter))
              painter)))
```

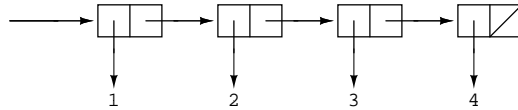
Answers which caused Happy Stick Man to appear at all (modulo minor syntactic corrections) were worth 5 points, those which appeared to have a recursive nature were worth 10 points, and those which were correct were worth 15 points. Examples which fell between these signposts received interpolated scores.

Problem 6: 15 points Draw the box and pointer diagrams for the following structures.

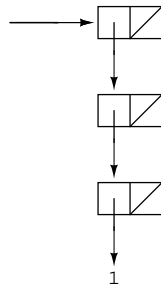
'(1)



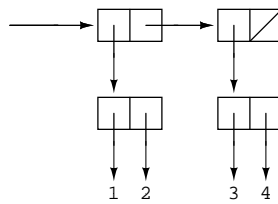
'(1 2 3 4)



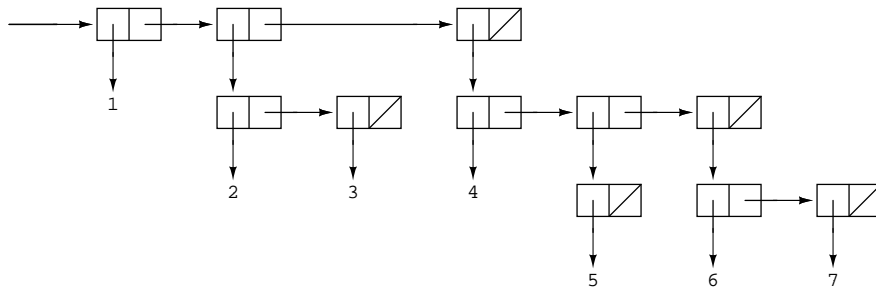
'(((1)))



'((1 . 2) (3 . 4))



'(1 (2 3) (4 (5) (6 7)))



Problem 7: OPTIONAL BONUS POINTS *This problem is optional. You do not need to complete this problem, and you should work on it only after completing the rest of the quiz.*

This problem examines *template matching*, a technique that appears often in digital signal processing. The idea is to search a very long list of integers (the *signal*) for all occurrences of a much shorter list of integers (the *template*), declaring a match to be found at a particular point in the signal when all elements from the template are found in order.

Write a procedure `template-match` that takes two arguments `sig` (the signal) and `tpl` (the template) and returns a list of indexes into `sig` where `tpl` is found exactly. For example, with inputs `sig: '(9 1 2 5 3 2 5 7 9 2 5 5)` and `tpl: '(2 5)`, your function should return the list `(3 6 10)`. Be sure that your procedure works for arbitrary length `sig` and `tpl`.

```
(define (template-match sig tpl)
  (define (match? long short)
    (cond ((null? long) #f)
          ((null? short) #t)
          (else
           (and (= (car long) (car short))
                (match? (cdr long) (cdr short))))))
  (define (tm sig tpl ctr)
    (cond ((null? sig) nil)
          ((match? sig tpl)
           (cons ctr
                 (tm (cdr sig) tpl (inc ctr))))
          (else
           (tm (cdr sig) tpl (inc ctr))))
  (tm sig tpl 1))
```

This function performs a task that is fundamentally like `filter`, in that it collects only elements of a list that match certain criteria. However, unlike `filter`, the criteria here depend on many elements of the list, not just a single isolated one. Further, the returned value (containing position information) is not directly associated with the list, but must be constructed on its own. These two aspects are orthogonal, and have been separated out into the two helper functions above. The first, `match?`, determines if there is a match for the sequence `tpl` at the current position in the signal `sig`. The second, `tm`, is an iterative helper function to loop through all elements of `sig`, keeping track of which element (through `ctr`), testing for matches at each, and possibly consing the current position on to the returned list.