EECS, 6.001, Structure and Interpretation of Computer Programs, Quiz 1

Massachusetts Institute of Technology

Wednesday, March 11, 1992

Write your name: ____

Write "your name": ____

Check your recitation section:

√	#	TIME	ROOM	INSTRUCTOR	TA
	1	9:00	34-301	Horn	Chiang
	2	9:00	26-314	McAllester	Dong
	3	10:00	34-301	Horn	Peters
	4	10:00	26-314	McAllester	Greenspun
	5	10:00	34-304	Grimson	Sigel
	6	11:00	34-301	Saltzer	Galperin
	7	11:00	34-304	Grimson	Connell
	8	12:00	34-301	Saltzer	Niyogi
	10	1:00	34-301	Ayers	Leigh
	11	1:00	34-303	Wilson	Chew
	12	2:00	24-502	Ayers	Gaunce
	13	2:00	34-303	Wilson	Davis
	14	11:00	26-314	Ward	Shoemaker
	15	12:00	26-314	Ward	Swartz

Problem 1	28	
Problem 2	15	
Problem 3	10	
Problem 4	24	
Problem 5	21	
Sundries	2	
Total	100	

Problem 1 (28 pts)

1a (4 pts). In this problem, you will construct a database of `persons`. A `person` is represented by two symbols, a `last-name` and a `first-name`. We write

```
(define (make-person last-name first-name)
  (cons last-name first-name))
```

Implement the corresponding selectors:

```
1. (define (last-name person)


2. (define (first-name person)
```

1b (6 pts). We can now represent a database as a list of `persons`. For example, we might begin to define the 6.001 staff database as follows:

```
(define rod-brooks (make-person 'brooks 'rod))
(define lynn-stein (make-person 'stein 'lynn))
(define 6001-database (list lynn-stein rod-brooks))
```

Draw a box and pointer diagram showing the value of 6001-database.

1c (6 pts). What will SCHEME print when the following expressions are evaluated?

1. `==> (car (cdr 6001-database))`

2. `==> (cdar 6001-database)`

1d (4 pts). Write a procedure, `find-first-names`, that takes two arguments—a last-name and a list of persons—and returns the list of first-names of persons with that last-name in that list. If there are no persons of that last-name in the list, the procedure should simply return the empty list. You may assume that last-names are symbols, but you should preserve the person abstraction.

```
(define (find-first-names l-name database)
```

1e (6 pts). When 6.001 was taught at Harvard several years ago, a similar scheme was developed. However, so many Harvard students have names like James Bryce Trumbull VIII that the Harvard version of the software uses a different abstraction for `persons`. Implement the constructor `make-person` to represent a person as a list consisting of three elements, the `last-name`, the `first-name`, and the `other-names`: Also re-implement the selectors `last-name` and `first-name`. Don't worry about the implementation of `other-names` for this problem. For example, if we

```
==> (define jbt8 (make-person 'Trumbull 'James '(Bryce VIII)))
JBT8
```

then

```
==> (last-name jbt8)
TRUMBULL
==> (first-name jbt8)
JAMES
```

```
1.  (define (make-person


2.  (define (last-name


3.  (define (first-name
```

1f (2 pts). What would happen if we tried to run `find-first-names` on the Harvard code?

Problem 2 (15 pts)

In SCHEME `if` and `cond` do very similar things. In fact, we might be tempted to define `if` as a procedure in terms of `cond`. Consider the following definition:

```
(define (new-if test-expr then-expr else-expr)
  (cond (test-expr then-expr)
        (else else-expr)))
```

We also define `(show x)` to do the same thing as `(print x)` but make sure that it returns `x`. (In many versions of SCHEME, `print` returns some funny unprintable value.)

```
(define (show x)
  (print x)
  x)
```

2a (12 pts). Assume that this particular version of SCHEME evaluates its arguments left to right. What is printed when the following expressions are evaluated by SCHEME?

```
1. ==> (new-if (> 3 5) (show 'true) (show 'false))
```

```
2. ==> (new-if 6 'true 'false)
```

```
3. ==> (define printing-if
         (lambda (test-expr)
           (new-if test-expr
                   'true
                   (show 'false))))
```

```
4. ==> (printing-if (= 5 5))
```

2b (3 pts). Using our same `new-if` assume that we evaluate

```
(define (smart-/ x y)
  (new-if (zero? y)
          (print ``attempted divide by zero'')
          (/ x y))
```

What happens when we then evaluate

```
==> (smart-/ 6 0)
```

Problem 3 (10 pts)

In problem set 3, you experimented with a triangle picture language based on Peter Henderson's square limit language. One of the procedures that you used split a triangle into two smaller triangles:

Consider split-2 which is a procedure that implements this:

```
(define (split-2 pict1 pict2 ratio)
  (lambda (triangle)
    (let ((s1 (side1 triangle))
          (s2 (side2 triangle))
          (oa (origin triangle)))
      (let ((p (scale ratio s2))
            (ob (shift s2 oa zero-vector)))
        (pict1 (make-triangle oa s1 p))
        (pict2 (make-triangle ob
                              (shift p oa ob)
                              (shift s1 oa ob)))))))
```

Note that split-2 is not exactly the same as the procedure split defined in problem set 3. In split-2, the origin of the second triangle is side2 of the original triangle (shifted to absolute coordinates). As a result, the **B** picture winds up oriented sideways.

We can exploit the fact that the triangle language is embedded in SCHEME to define a spiraling picture. For example,

Use `split-2` to define a procedure, `spiral-push`, that takes a picture, a ratio, and a number of times to "push" the picture as arguments and returns a spiral picture as shown above. (This can be done in as few as six lines.)

```
(define (spiral-push pict r n)
```

Problem 4 (21 pts)

4a (15 pts). For each of the following write [procedure] if it returns a procedure, [error] if it causes an error, and otherwise the value that is returned. (Hint: *substitution model*.)

```
1. ==> ((lambda (a b) (b 3 a))
        (lambda (x y) (+ 5 x))
        (lambda (x y) (- 5 x)))
```

```
2. ==> ((lambda (a b c d e)
           (cond ((= d 0) (a e 2))
                 ((= d 1) (b e 2))
                 ((= d 2) (c e 2))))
        + * / 1 6)
```

```
3. ==> ((lambda (a b) (b a))
        2
        (lambda (x)
          (lambda (y) (+ x y))))
```

```
4. ==> ((lambda (a b) (a b))
        2
        (lambda (x)
          (lambda (y) (+ x y))))
```

```
5. ==> (((lambda (x y)
           (lambda (f)
             (f x y)))
         'a 'b)
        (lambda (x y) y))
```

Recall our examples from lecture of defining a `sum` operator, and using it to compute the definite integral of a function represented by a procedure `f`.

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))

(define (integral f a b dx)
  (define (add-dx x) (+ x dx))
  (* (sum f a add-dx b) dx))
```

Now consider the problem of writing a higher order procedure `make-integral` which takes a procedure representing a function, and returns another procedure representing the integral of the function. I.e., given f we would like to return the procedure for F in the following equation.

$$F(x) = \int_0^x f(t)dt$$

For instance we would like to be able to do the following:

```
==> (define intcos (make-integral cos))
intcos
==> (intcos (/ pi 6))
0.5
```

We might not get exactly `0.5` in our example because we are using a discrete approximation to the integral.

4b (6 pts). Complete the definition of `make-integral` (it is quite short—you can use the procedures defined above if you wish)—it need only work for positive x, and you can use a value of `0.001` for `dx`:

```
(define (make-integral f)
```

Problem 5 (25 pts)

Consider the following orders of growth:

a. $O(n)$

b. $O(n^2)$

c. $O(2^n)$

d. $O(\log n)$

e. $O(n^2 + n)$

5a (6 pts). For each of the expressions below, fill in *a*, *b*, *c*, *d*, or *e* to indicate the smallest order of growth from above which applies:

1. $n^2 + 2n + 3$ ____

2. $3n$ ____

3. $n^2 + 2^n$ ____

Consider the following two definitions of procedures which return a reversed list that is given as an argument.

```
(define (reverse1 lst)
  (define (append x y)
    (if (null? x)
        y
        (cons (car x) (append (cdr x) y))))
  (if (null? lst)
      '()
      (append (reverse1 (cdr lst))
              (cons (car lst) '()))))

(define (reverse2 lst)
  (define (reverse2-aux rem sofar)
    (if (null? rem)
        sofar
        (reverse2-aux (cdr rem)
                      (cons (car rem) sofar))))
  (reverse2-aux lst '()))
```

5b (6 pts). Circle the appropriate cases in the following.

1. The process generated by `reverse1` is	**iterative**	**recursive**
2. The process generated by `reverse2` is	**iterative**	**recursive**

5c (6 pts). Suppose that the argument to `reverse1` or `reverse2` is a list of length n. In each case give the order of growth for the number of calls to `cons` made by each of the two procedures:

1. `reverse1`:

2. `reverse2`:

Here is a definition of `length` that is recursive:

```
(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (cdr lst)))))
```

5d (6 pts). Complete the following to write an iterative version of `length`. It can be done in the style of other examples in this quiz with only five more lines of code. Excessively longer solutions will be penalized.

```
(define (length lst)
```