TEL AVIV UNIVERSITY
Department of Computer Science
0368.1105 – Extended Introduction to Computer Science
Spring Semester, 2000/2001

Homework 7, Dec 31, 2000

This homework is due by 18:00 on Thursday, Jan 12. Write your name, id and the **Tirgul group number** on top of the first page. Keep a copy of the homework. Please answer all questions that require you to write a procedure by submitting the procedure itself, and the result of testing your code on the computer (a printout). Try to pick at least one meaningful example of an execution to show that the procedure works. There is a lot of work so we suggest that you start working right away.

Mutation - the basic concepts

1. While `set!` is a special form, `set-car!` and `set-cdr!` are both simply procedures. The contracts are that for a cons cell `<z>`, after

```
(set-car! <z> <a>)
(set-cdr! <z> <b>)
```

then

```
(car <z>) ==> <a>
(cdr <z>) ==> <b>.
```

For each of the following sets of expressions fill in what is returned.

```
(define a '(1 2 3))
(set-car! (cdr a) 'x)

a ==>

(define a '(a b c))
(define b '(d e f))
(define c (append a b))

c ==>

(set-car! (cdr a) 2)
(set-cdr! (cddr b) '(g h i))

a ==>
b ==>
```

```
c ==>

(set-car! c 1)
(set-car! (cdddr c) 4)

a ==>
b ==>
c ==>
```

2. Recall that `(eq? a b)` returns true if pair `a` and pair `b` are the same pair. Pair `a` and pair `b` are the same pair if a change to `a` with `set-car!` or `set-cdr!` is reflected by the same change to `b`. What would be the result of evaluating the following expressions?

```
(define a (list 'x 'y 'z))
(define b (list 'x 'y 'z))

(eq? a b) ==>

(set-cdr! b (cdr a))

a ==>
b ==>

(eq? (cdr a) (cdr b)) ==>

(set-car! (cdr a) 'foo)

a ==>
b ==>
(eq? (cddr a) (cddr b)) ==>
```

3. Write a procedure called `rotate!` which takes a list, makes the last element in the list the first one, and returns the rotated list. Your procedure should accomplish this task *without* creating any new pairs, i.e., without using `cons, list` etc. For example:

```
(rotate! '(1 2 3 4))  ==>  (4 1 2 3)
```

What would be returned by the following expressions?

```
(define a '(1 2 3 4))
(define b (cddr a))
(define c (rotate! a))
```

```
a ==>
b ==>
c ==>
```

Explain!

4.

Write a version of `reverse` which creates no new pairs (does not use `cons, list` etc), but instead changes all the cdr's of the cons cells of its argument.

```
(define (reverse!  lst) ... )
```

5. *Memoization* is a technique that enables a procedure to record, in a local table, values that have previously been computed. This technique can make a vast difference in the performance of a program. A memoized procedure maintains a table in which values of previous calls are stored using as keys the arguments that produced the values. When the memoized procedure is asked to compute a value, it first checks the table to see if the value is already there and, if so, just returns that value. Otherwise, it computes the new value in the ordinary way and stores this in the table. As an example of memoization, recall the exponential process for computing Fibonacci numbers:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

The memoized version of the same procedure is

```
(define memo-fib
  (memoize (lambda (n)
             (cond ((= n 0) 0)
                   ((= n 1) 1)
                   (else (+ (memo-fib (- n 1))
                            (memo-fib (- n 2))))))))
```

where the memoizer is defined as

```
(define (memoize f)
  (let ((table (make-table)))
    (lambda (x)
      (let ((previously-computed-result (lookup x table)))
        (or previously-computed-result
            (let ((result (f x)))
              (insert! x result table)
              result))))))
```

The code for the table can be found in the handout of lecture 18, but can also be downloaded (together with the code for this entire question) from the course web page. (We renamed `assoc` as `my-assoc` to avoid name clashes with another procedure).

Draw an environment diagram to analyze the computation of `(memo-fib 3)`. In order to simplify the diagram, you should treat the table procedures `make-table`, `lookup`, and `insert!` as if they were primitive diagram (do not draw frames for the calls on these procedures). Also, use the short form for `let` - do not draw the procedure object for the "hidden" lambda.

Explain why `memo-fib` computes the n'th Fibonacci number in a $\Theta(n)$ steps. (Assuming that table operations take constant time. This is not true when using the table implementation we have used, but it is possible using another data structure, called hash table). Would this scheme still work if we had simply defined `memo-fib` to be `(memoize fib)`?.

6.

Do excercises 3.16 and 3.17 from SICP2.

7. Implement the procedure `(reverse stack s1 s2)` in the definition of `make-queue` given in class. This routine should delete the elements from $s1$ one by one and insert them in reverse order into $s2$. Download the code given in class, add your implementation, and try it on some nontrivial sequence.