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

Homework 8, Jan 11, 2001

This homework is due by 18:00 on Sunday, Jan 21, 2001. 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.

If you are using PCscheme, you need to use the following definitions:

```
(define stream-car head)
(define stream-cdr tail)
(define stream-null? empty-stream?)
```

You will also need

```
(define (display-line x) (display x) (newline))
```

Streams

1. What will each of the following commands print? Explain!

```
==> (define s (cons 5 (begin (display-line 7) 9)))
==> (car s)
==> (cdr s)
```

Now we change `cons` to `cons-stream`. What will each of the following print? Explain!

```
==> (define s (cons-stream 5 (begin (display-line 7) 9)))
==> (stream-car s)
==> (stream-cdr s)
```

Note that the value of `s` in the second definition is not really a stream, just as the value in the first definition is not a list!! A stream must always end with a `the-empty-stream` just as a list ends with a `nil`.

2. What would be the value returned by each of the following commands? Explain!

```
==> (define a 1)
==> (define b (delay a))
==> (define c (delay a))
==> (define d (let ((a 1)) (delay a)))
==> (set! a 2)
==> (force b)
==> (set! a 3)
==> (force b)
==> (force c)
==> (force d)
```

Answer twice. Once, assuming that `(delay <exp>)` is syntactic sugar for `(lambda () <exp>)`, and once assuming that a memoizing procedure, like `memo-proc`, is used in the implementation of `delay`. Which implementation is used by the interpreter that you are using?

3. In order to take a closer look at delayed evaluation, let us use the following procedure, which simply returns its argument after printing it:

```
(define (show x)
  (display-line x) x)
```

What would the interpreter print in response to each expression in the following sequence? (The procedures were defined in class, and will be available for downloading from the course web page). Explain!

```
==> (define x (stream-map show (stream-enumerate-interval 0 10)))
==> (stream-ref x 5)
==> (stream-ref x 10)
```

Answer twice. Once, assuming that `(delay <exp>)` is syntactic sugar for `(lambda () <exp>)`, and once assuming that a memoizing procedure, like `memo-proc`, is used in the implementation of `delay`.

4. Complete the following definition, which generalizes `stream-map` to allow procedures that take multiple arguments and multiple streams of equal length.

```
(define (stream-map proc . argstreams)
  (if (______________ (car argstreams))
    the-empty-stream
    (______________
      (apply proc (map ______________ argstreams))
      (apply stream-map
             (cons ______________
                   (map ______________ argstreams))))))
```

Using this version of `stream-map`, complete the definitions of `add-streams` and `mul-streams` that take an arbitrary number of streams as inputs and return a stream whose i-th element is the sum, or product, of the i-th elements of the argument streams.

```
(define (add-streams . argstreams)
   (apply stream-map (cons __________ _______________ )))
(define (mul-streams . argstreams)
   (apply stream-map (cons __________ _______________ )))
```

5. Complete the following definitions so that `integers` would be the stream $1, 2, 3, 4, 5 \ldots$, `odds` would be the stream $1, 3, 5, 7, \ldots$, and `factorials` would be the stream of factorials, i.e., $1, 2, 6, 24, \ldots$.

```
(define integers (cons-stream 1
                              (stream-map _______________ integers)))
(define odds (stream-filter _______________ integers))

(define odds (cons-stream 1
                          (add-streams _______________ )))

(define factorials (cons-stream 1 _______________ )))
```

6. How many elements are there in the stream

```
(stream-filter odd? (stream-filter even? integers))
```

How will the interpreter respond to the following inputs?

```
==> (stream-filter odd? (stream-filter even? integers))

==> (stream-null? (stream-filter odd? (stream-filter even? integers)))
```

7. Write a procedure `make-repeated` which takes as its argument a procedure `f` (of one argument) and returns the infinite stream: `[` $\mathtt{f}^{(0)}$ $\mathtt{f}^{(1)}$ $\mathtt{f}^{(2)}$ `... ]` , Where $\mathtt{f}^{(n)}$ is `f` applied n times. (For example, $\mathtt{f}^{(3)}(x) = \mathtt{f}(\mathtt{f}(\mathtt{f}(x)))$.

8. Write a procedure `apply-streams` which takes two arguments - a stream of procedures, and a stream of arguments and returns the stream of the results of applying the i-th procedure to the i-th argument. For example:

```
(define ones (cons-streams 1 ones))
(apply-streams (make-repeated +1) ones)
```

should return the stream of integers.

9. Consider the following definitions:

```
(define (twice f) (lambda (x) (f (f x))))
(define s1 (make-repeated twice))
(define s2 (make-repeated +1))
(define s3 (apply-streams s1 s2))
```

What would be the result of:

```
(apply-streams s3 ones)
```

Give the resulting stream, specifying at least its five first arguments.