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

Homework 9, Jan 21, 2000

This homework is due by 18:00 on Sunday, Jan 28, 2000. 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.

Streams

1. a) Define a procedure `partial-sums` that takes as argument a stream S and return the stream whose elements are S_0, $S_0 + S_1$, $S_0 + S_1 + S_2$,.... For example, `(partial-sums integers)` should be the stream 1,3,6,10,15,....
b) Define a procedure `powers` that takes as argument an integer x and returns the stream 1,x,x^2,x^3,....
c) One can approximate a function $g(x)$ using the power series

$$g(x) = g(0) + xg'(0) + \frac{x^2}{2}g''(0) + \frac{x^3}{3!}g'''(0) + \cdots$$

where $g'(0)$ is the value of the derivative of g at zero, $g''(0)$ is the value of the second derivative of g at zero, and so on. For example

$$cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \cdots ,$$

and

$$sin(x) = x - \frac{x^3}{6} + \frac{x^5}{120} - \cdots .$$

Write a procedure `power-series` that gets as an argument the stream $g(0), g'(0), g''(0), g'''(0), \ldots$ and return a function of one argument. The returned function when applied to an argument x it should produce the stream

$$g(0),\ g(0) + xg'(0),\ g(0) + xg'(0) + \frac{x^2}{2}g''(0), \ldots$$

of approximations to $g(x)$. (You may find the procedures `powers`, `partial-sums` and the stream `factorials` from exercise 8 useful for constructing `power-series`.)

d) Note that the stream $sin(0), sin'(0), sin''(0), \ldots$ is periodic and equals $0, 1, 0, -1, 0, 1, 0, -1, \ldots$. Define this stream and use it together with the function `power-series` to approximate $sin(\pi/8)$.

2. Using the `interleave`, we defined a procedure called `pairs` that takes two streams and creates a stream of pairs as follows:

```
(define (pairs s1 s2)
    (cons-stream
        (list (stream-car s1)(stream-car s2))
        (interleave
            (stream-map (lambda (x) (list (stream-car s1) x))
                        (stream-cdr s2))
            (pairs (stream-cdr s1) (stream-cdr s2)))))
```

Examine the stream `(pairs integers integers)`. How many pairs precede the pair (1,100)? How many pairs precede (99,100)? If you find it hard to give the precise mathematical statement you can give more qualitative answers, the idea is to understand to order of elements in this stream.

Write a procedure `(pair-index i j)` that receives two integers `i` and `j`, such that `i` $<$ `j` , and returns the place in the stream `(pairs integers integers)` in which the pair `(i,j)` appears. Your procedure should not use `pairs`, or any other procedure that uses streams. Can you find the index of the pair (99,100) using your procedure?

3. By slightly modifying the `pairs` procedure defined above, produce a procedure called `all-pairs` such that `(all-pairs integers integers)` will produce the stream of *all* pairs of integers (i, j), not only those for which $i < j$.

4. Consider the following alternative definition for the `sqrt-stream` procedure which was defined in class.

```
(define (sqrt-stream x)
  (cons-stream 1.0
               (stream-map (lambda (guess)
                             (sqrt-improve guess x))
                           (sqrt-stream x))))
```

a) Draw an environment diagram describing the frames and pairs existing in memory after the evaluation of `(define t (sqrt-stream 2))`.
b) Draw the frames and pairs which are created by a subsequent evaluation of `(stream-cdr t)`.

Assume that `(cons-stream <x> <y>)` is a syntactic sugar for `(cons <x> (memo-proc (lambda () <y>)))`, and that `(stream-cdr <x>)` is a syntatic sugar for `((cdr <x>))`.

Explain in what sense this implementation is less efficient than the following implementation which was given in class.

```
(define (sqrt-stream x)
  (define guesses
   (cons-stream 1.0
             (stream-map (lambda (guess)
                            (sqrt-improve guess x))
                         guesses)))
   guesses)
```

Vectors and sorting

5. Write procedures `vector-map`, `vector-filter` and `vector-accumulate`, that are similar to `map`, `filter` and `accumulate`, but work on vectors instead of lists. `vector-map` should be like the more general version of `map` – it should be able to get any number of vectors as arguments (following a procedure argument).

6. Use procedures from the previous exercise to implement the following operations on vectors:

`(+vec v w)` returns a vector which is the sum of the vectors v and w

`(*vec v w)` returns the scalar product of `v` and `w`

`(=vec v w)` returns **true** iffv = `w`

Your procedures must check that `v` and `w` are indeed of the same length. Otherwise, they should print an appropriate error message.

7. Implement a *bounded stack* abstract data type (a stack whose maximum size is determined upon creation) using object oriented style. Use a *vector* to hold the stack data (not a list).

The stack object will support the following operation:

- `make-bounded-stack` - the constrcutor returning a stack object that is initially empty. Accept one argument - the size.
- `push!` (we used the name `insert`) - a mutator adding an item (the value of the argument) to the top of the stack, and returns the symbol ok.
- `pop!` (we used the name `delete`) - a mutator that removes the item at the top of the stack, and returns the symbol ok.
- `top` - a selector that returns the value of the item at the top.
- `empty-stack?` - a predicate selector, that returns **true** if and only if the stack is empty.
- `full-stack?` - a predicate selector, that returns **true** if and only if the stack is full (the number of elements is equal to the maximum size).

8. Mergesort as Quicksort is also a recursive sorting algorithm. It splits the collection into two groups, sorts each separately, and then combines them by merging the sorted pieces into one sorted collection. The following is an implementation of mergesort for lists.

```
(define (mergesort s)
  (cond ((null? s) s)
        ((null? (cdr s)) s)
        (else
          (let ((s1 (odd-elements s))
                (s2 (even-elements s)))
            (merge (mergesort s1) (mergesort s2))))))
```

Write the subroutines `odd-elements`, `even-elements`, and `merge` to complete the implementation. Note that input lists to merge are sorted. What is the running time of mergesort ?