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

Homework 7, Dec 17, 2000

This homework is due by 18:00 on Thursday, Dec 28. 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.

Environment Diagrams - Let

When drawing an environment diagram with the let special form, one could desugar the let into a lambda application. A short cut is to "hang" a new frame (that is, create a new frame that points to the current frame) with the variables of the let statement bound to their appropriate values (be careful when finding these values – they are evaluated in the current frame, not the new frame). Then evaluate the body of the let in the new frame. This means that we do not create the procedure object for the "hidden" lambda.

In this homework you are encouraged to use the short cut.

Block Structure, Lexical Scoping and the Environment Model

1. Draw the environment diagram for the following example. Number the frames and procedure objects to show the order in which they are generated.

```
(define (make-container size)
  (let ((cont 0))
    (lambda (op)
      (cond ((eq? op 'inc)
             (if (< cont size) (set! cont (+ cont 1))) 'ok)
            ((eq? op 'dec)
             (if (> cont 0) (set! cont (- cont 1))) 'ok)
            ((eq? op 'show) (list cont size))))))

(define c (make-container 3))

(c 'inc)

(c 'show)
```

2. Draw the changes that occur in an empty environment as the following code is evaluated:

```
(define (make-counter init)
  (lambda () (set! init (+ init 1))    init))
(define count! (make-counter 10))
(define count-1 (count!))
(define (count-2) (count!))
(define count-3 count!)
(define (count-4) count!)
```

What is the value of each of the following expressions (evaluated in the order shown)? Add to your environment diagram as you evaluate them.

```
count-1
count-2
count-3
count-4
(count-1)
(count-2)
(count-3)
(count-4)
count-1
(count-3)
(count-2)
```

Using Environments

3. Define a procedure `make-counting-proc` that takes a single parameter - a procedure of one argument - and returns a similar procedure, that prints how many times it was called, before returning its value. For example:

```
==> (define my-length (make-counting-proc length))
my-length
==> (my-length '(a b c))
applied 1 times
3
==> (my-length '(a b c))
applied 2 times
3
==> (my-length '(b c))
applied 3 times
2
```

4. Define a one-parameter procedure `prev` that returns as its value the argument that was passed to it the previous time that it was called. The first time `prev` is called, it should return the symbol `*first-call*`. In this version of `prev`, use a global variable to hold state information.

```
(prev 'a)
```
*;Returns *first-call**
```
(prev 3)
```
;Returns a
```
(prev (+ 1 5))
```
;Returns 3

Consider the following `let` statement. Show the effect of this statement in the environment.

```
(let ((last-value '*init-value*))
  (define (prev new)
  ... ))
```

5. Define a procedure `make-prev` that can be used to create the procedure `prev` and add it to the environment. `Make-prev` should take as an argument the value that should be returned by the first call to `prev`. That is, if we use `(define prev (make-prev '*first-call*))`, then `prev` should behave as shown above. Add `make-prev` to the environment. Show the effect of `(define prev (make-prev '*first-call*))` followed by `(prev 'a)`.