TEL AVIV UNIVERSITY
Department of Computer Science
0368.1105 – Extended Introduction to Computer Science
Spring Semester, 1999/2000

Homework 3, Wed. March 8, 2000

This homework is due by 18:00 on Wednesday, March 15, at the hand-in boxes near room 005 of the Schreiber building. 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.

Higher Order Procedures

1. Consider the following procedures.

```
(define why
  (lambda (f a b)
    (if (< (f a) (f b))
        a
        b)))

(define who
  (lambda (y)
    (+ (* y y) y 1)))
```

Use the substitution model to show the evaluation of the following expression. Be careful when substituting into special forms.

```
==> (why who 2 5)
;value:
```

2. As defined in the lecture, `sum` generates a recursive process. Define a new version of `sum` that generates an iterative process.

How does this affect `sum-int`, `sum-squares`, and `pi-sum`?

3. Write a procedure `product`, analogous to `sum`, which computes the product of the values of a given function at points over a given range.

4. Define a procedure `make-mul` with one parameter c that returns a procedure with one parameter that multiplies its argument by c.

```
==> (define double (make-mul 2))
 ; double -> [# compound procedure...]
 ; value: the-undefined-value
==> (double 4)
 8
==> ((make-mul 3) 4)
 12
```

5. Suppose we define

```
==> (define apply-to-ten
      (lambda (f) (f 10)))
```

What will the results of evaluating the following expressions be?

```
==> (apply-to-ten 1+)
;value:
==> (apply-to-ten (lambda (p) (* 7 p)))
;value:
==> (apply-to-ten apply-to-ten)
;value:
```

6. Write a procedure `apply-to` that, given a number n, returns the procedure analogous to `apply-to-ten` (for $n = 10$). That is, you should be able to write

```
(define apply-to-ten (apply-to 10))
```

7. Use substitution to evaluate `(apply-to-ten triple)` where `apply-to-ten` is as defined in 6 and `triple` is `(define (triple x) (* 2 x))`

8. Write a procedure `shift` which, given a function F and a value a produces the function G such that $G(x) = F(x + a))$. That is, you should have

```
((shift square 3) 2) → 25
```

9. Write a procedure `compose` such that `(compose F G)` evaluated at `x` returns `F` applied to `(G x)`. For example:

```
((compose square 1+) 7) → 64
((compose 1+ square) 7) → 50
```

10. Use `compose` to give a definition for `repeatedly`, where `(repeatedly fn n)` is a procedure that applies `fn` to its argument `n` times:

```
(define 5+ (repeatedly 1+ 5))

(5+ 100) → 105
```

Hint: *Wishful thinking.*

11. (Instant insanity). Suppose `mad` is defined as follows:

```
(define (mad fn)
  (lambda (x) (fn (fn (fn x)))))
```

What are

```
((mad 1+) 0)
((mad (mad 1+)) 0)
(((mad mad) 1+) 0)
```

12. For each of the cases below, give an expression involving `foo` whose evaluation will return the value 2.

```
(define foo
  (lambda (x) (* 2 x)))

(define foo
  (lambda (x) (x 1)))

(define foo
  (lambda (x) (lambda (y) (* x y))))

(define (foo x)
  (x (lambda (y) (* y 2))))
```

13. Do exercises 1.37, 1.38. 1.39 from SICP2u on pages 71-72.

Review of `let` and Lexical Scoping (Block Structure)

14. What does scheme return as the result of the `let` expression below?

```
(define x 2)
(define y 3)

(let ((x y)
      (y (+ x y)))
  (* x y))
```

Expand the `let` as a `lambda`.

15. Consider the following definition:

```
(define d 2)
(define (f a b)
  (let ((a 3)
        (b (b (+ a 1)))
        (c (* a 2)))
    (+ a b c d)))
(define (g d) (f d (lambda (x) (* 2 x))))
```

Show the different variable's scopes, as follows. First, unfold the sugared definitions of `g`, `f`, and the `let` into lambdas. Then draw boxes around the various scopes. Draw arrows from each occurence of a variable to the definition that binds it. Remember that in lexical scoping, we always look for a variable inductively up the levels of the lambdas (procedures) in which it was defined. Using the substitution model, show what
`(g 5)` evaluates to.