# Assignment 4

April 30, 1996

## 1 Due Date

May 9, 1996. Please turn in your solution at the beginning of the lecture.

## 2 Format

The assignment consists of converting a number of procedures to continuation-passing style (CPS). The continuation argument should be the last argument of the new procedure. For example if you were asked to convert the following procedure into CPS,

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (sub1 n))))))
```

your answer would be

```
(define fact-cps
  (lambda (n k)
    (if (zero? n)
        (k 1)
        (fact-cps (sub1 n) (lambda (v) (k (* n v)))))))
```

## 3 Assignment

Convert the following procedures to CPS. (The last four questions are harder than the rest).

1.

```
(define dupla
  (lambda (a n)
    (if (zero? n)
        '()
        (cons a (dupla a (sub1 n))))))
```

2.

```
(define listmul
  (lambda (ls)
    (if (null? ls)
        1
        (* (car ls) (listmul (cdr ls))))))
```

3.

```
(define treemul
  (lambda (tr)
    (cond
      [(null? tr) 1]
      [(not (pair? (car tr))) (* (car tr) (treemul (cdr tr)))]
      [else (* (treemul (car tr)) (treemul (cdr tr)))])))
```

4.

```
(define union
  (lambda (a b)
    (cond
      [(null? a) b]
      [(memv (car a) b) (union (cdr a) b)]
      [else (cons (car a) (union (cdr a) b))])))
```

5.

```
(define fib
  (lambda (n)
    (if (< n 2)
        1
        (+ (fib (- n 1)) (- n 2)))))
```

6.

```
(define snoc
  (lambda (i ls)
    (if (null? ls)
        (list i)
        (cons (car ls) (snoc i (cdr ls))))))
```

7. Assume `f` has been converted to CPS:

```
(define map
  (lambda (f ls)
    (if (null? ls)
        '()
        (cons (f (car ls)) (map f (cdr ls))))))
```

8. Assuem `pred` has been converted to CPS:

```
(define filter
  (lambda (pred ls)
    (cond
      [(null? ls) '()]
      [(pred (car ls)) (filter pred (cdr ls))]
      [else (cons (car ls) (filter pred (cdr ls)))])))
```

9. Assume `f` and `g` have been converted to CPS. Not only should `compose` be converted to CPS, but the procedure it returns should also be converted.

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))
```

10.

```
(define depth
  (lambda (ls)
    (cond
      [(null? ls) 1]
      [(not (pair? (car ls))) (depth (cdr ls))]
      [else (let ([head (add1 (depth (car ls)))]
                  [tail (depth (cdr ls))])
              (if (< head tail) tail head))])))
```

# 4 Challenges

- Write a Scheme procedure that does the homework for you.
- Towers of Hanoi revisited: Convert your original (recursive) solution to the towers of Hanoi problem to CPS. *Derive* the non-recursive implementation from the CPS vesion by changing the representation of continuations (using records instead of procedures).
- Convert the following programs to CPS:

```
(define listmul
  (lambda (l)
    (cond
      [(null? l) 1]
      [(zero? (car l)) (error 0)]
      [else (* (car l) (listmul (cdr l)))])))

(define listmul
  (lambda (l)
    (call/cc (lambda (exit)
```

```
                    (letrec ([loop (lambda (l)
                                     (cond
                                       [(null? l) 1]
                                       [(zero? (car l)) (exit 0)]
                                       [else (* (car l) (loop (cdr l)))]))])
                      (loop l))))))
```

- Implement coroutines in Scheme. More specifically write the code for the Scheme procedure `make-coroutine` below such that the pseudo-code for battleship works.

```
(define player-1-code
  (let ([Board (make-board)])
    (lambda (resume other-player-first-shot)
      (letrec ([loop (lambda (other-player-shot)
                       (if (other-player-shot-is-fatal?)
                           (I-lost-the-game)
                           (let ([his-next-shot
                                   (resume player-2 (compute-my-shot))])
                             (loop his-next-shot))))])
        (loop other-player-first-shot)))))

(define player-2-code ...)

(define player-1 (make-coroutine player-1-code))
(define player-2 (make-coroutine player-2-code))
```

- Read about the `samefringe` problem on page 315 of the book and understand the coroutine solution. Write a solution in your favorite programming language. The only constraint is that you don't needlessly traverse the entire trees.