COMP 210: TEST 2 – APPENDIX

Matthias Felleisen

NOVEMBER 6, 1996

Name: ____

Instructions

1. Fill in the information above and the pledge below.
2. You may *only* use the following Scheme functions and special forms

   null? null cons? car cdr cons list? list *length member*
   number? = + − * < > / quotient remainder
   boolean? not **and** **or** symbol? eq? procedure?
   define define-structure lambda cond if local let set! begin

3. **You must write down which program design method (structural recursion, generative recursion, x plus accumulator style, instantiation of library functions, memorizing history, changing state) you choose to solve a problem.**
4. Define any help functions you use. If it is at an unobvious place, point to it.
5. Remember that input data is correct. You don't need to handle error cases.
6. There are *two* problems. Together they are worth 25 points; the relative weight is indicated at the beginning of each problem and in the table below.
7. You have 50 minutes.

Pledge:

Problem	Max. Pts.	
1	10	
2	15	

Problem 1. [10pts] Ms. Devika likes to design things with computers. Recently she has gotten into mobiles. She would like you to write a program that helps her build balanced mobiles. Here is the beginning of her program:

```
(define-structure (iron-ball weight))

(define-structure (joint left right))
```

Her data description for a mobile is as follows:

A **mobile** is either

1. (make-iron-ball *n*) where *n* is a some number, or
2. (make-joint *m1 m2*) where *m1* and *m2* are mobiles.

Here are two simple examples:

```
(define example1
  (make-joint (make-iron-ball 10)
              (make-joint (make-iron-ball 5)
                          (make-iron-ball 5))))

(define example2
  (make-joint (make-joint (make-iron-ball 5)
                          (make-iron-ball 5))
              (make-joint (make-iron-ball 3)
                          (make-iron-ball 6))))
```

Derive the program *total-weight*, which accepts a mobile and returns its total weight.

```
;; Structural recursion

;; Template:
(define (fun-mobile a-mbobile)
  (cond
    ((iron-ball? a-mboile) ... (iron-ball-weight a-mbile) ...)
    (else ... (fun-mobile (joint-left a-mboile)) ...
      ... (fun-mobile (joint-right a-mboile)) ...)))

;; Functions:
(define (total-weight a-mobile)
  (cond
    ((iron-ball? a-mobile) (iron-ball-weight a-mobile))
    (else (+ (total-weight (joint-left a-mobile))
             (total-weight (joint-right a-mobile))))))
```

Problem 2. [15pts] Mr Shriram K., the department's former Web master, likes to play games with his Web pages. Here is a Scheme description of his latest organization:

A **Web page** is either

1. 'HelloWorld, or
2. 'GoodBye, or
3. (cons *wp* (cons 'left null)), where *wp* is a Web page, or
4. (cons 'right (cons *wp* null)), where *wp* is a Web page.

Derive the program *path-to-hello*, which accepts a Web page and returns the lr-path to the symbol 'HelloWorld or #f if 'HelloWorld does not occur in the Web page. The lr-path is the list of symbols 'left and 'right that one encounters on the way to the Web page with the word 'HelloWorld. For example,

```
(path-to-hello (list (list 'right 'HelloWorld) 'left)) = (list 'left 'right)
(path-to-hello (list (list 'right (list 'right 'HelloWorld)) 'left)) = (list 'left 'right 'right)
(path-to-hello (list (list 'GoodBye 'right) 'left)) = #f
```

Hint: Use eq? to compare arbitrary values to symbols.

```
;; Structural recursion

;; Template:
(define (fun-wp a-wp)
  (cond
    ((eq? a-wp 'HelloWorld) ...)
    ((eq? a-wp 'GoodBye) ...)
    ((eq? (car a-wp) 'right) ... (fun-wp (car (cdr a-wp))) ...)
    (else ... (fun-wp (car a-wp)) ...)))

;; The function:
(define (path-to-hello a-wp)
  (cond
    ((eq? a-wp 'HelloWorld) null)
    ((eq? a-wp 'GoodBye) #f)
    ((eq? (car a-wp) 'right)
     (let ((r (path-to-hello (car (cdr a-wp)))))
       (if r (cons 'right r) #f)))
    (else
      (let ((r (path-to-hello (car a-wp))))
        (if r (cons 'left r) #f)))))
```

```
;; ...or with a help function:
(define (path-to-hello a-wp)
  (cond
    ((eq? a-wp 'HelloWorld) null)
    ((eq? a-wp 'GoodBye) #f)
    ((eq? (car a-wp) 'right) (combine 'right (path-to-hello (car (cdr a-wp)))))
    (else (combine 'left (path-to-hello (car a-wp))))))

(define (combine s r) (if r (cons s r) #f))
```

```
;; Here is a solution based on structural recursion plus accumulator:
(define (path-to-hello a-wp)
  (local ((define (path-to-hello-acc a-wp accu)
            (cond
              ((eq? a-wp 'HelloWorld) (reverse accu null))
              ((eq? a-wp 'GoodBye) #f)
              ((eq? (car a-wp) 'right) (path-to-hello-acc (car (cdr a-wp)) (cons 'right accu)))
              (else (path-to-hello-acc (car a-wp) (cons 'left accu)))))

          (define (reverse l r)
            (cond
              ((null? l) r)
              (else (reverse (cdr l) (cons (car l) r))))))
    (path-to-hello-acc a-wp null)))
```