COMP 210: TEST 2 – SOLUTION

Matthias Felleisen

OCTOBER 31, 1996

Problem 1 [20pts] Mr. Prof. Dr. Druschel is supposed to implement a file system for a brand new OS, nick-named Doors 99. Naturally he chooses Scheme structures to represent files and directories, which are collections of files:

(**define-structure** (*file name size contents*))

(**define-structure** (*directory name size contents*))

His data description for a file system is as follows:

A **file system** is always implemented as a *directory* value whose *name* field is a symbol, *size* is a natural number, and *contents* is a **listing**.

A **listing** is either

1. null, or
2. (cons f l) where f is a file and l is a listing, or
3. (cons d l) where d is a directory and l is a listing.

Here is a simple example:

```
(define example
  (make-directory '/ 10
    (list (make-file 'README 4 (list 'a 'b 'c))
          ...)))
```

Part 1:a. Your job is to derive the program *exists-file?*. The programs inputs are a name N and a file system FS. It returns #t if N occurs in FS as the name of a file or of a directory and #f if it doesn't.

Templates: one per data description (not necessary this time but helpful)

```
(define (f-file dir)
  ... (directory-name dir) ...
  ... (directory-size dir) ...
  ... (f-listing (directory-contents dir)) ...)

(define (f-listing lofd)
  (cond
    ((null? lofd) ...)
    ((file? (car lofd))
     ... (car lofd) ... (f-listing (cdr lofd)) ...)
    (else
      ... (f-file (car lofd)) ... (f-listing (cdr lofd)) ...)))
```

Functions:

```
(define (exists-file? fname dir)
  (if (eq? (directory-name dir) fname)
      #t
      (exists-listing fname (directory-contents dir))))

(define (exists-listing fname lofd)
  (cond
    ((null? lofd) #f)
    ((file? (car lofd)) ;; yes, this could be formulated with or
     (if (eq? (file-name (car lofd)) fname)
         #t
         (exists-listing fname (cdr lofd))))
    (else (or (exists-file? fname (car lofd))
              (exists-listing fname (cdr lofd))))))
```

Part 1:b. Next Professor Druschel asks you to modify the program into the program *path-to-file*. The new programs also inputs a name N and a file system FS. Like the function *file-exists?*, it returns #f if N does not occur in FS and the list of directory names that leads to the file or directory with the given name if the name occurs in the file system. For example,

```
(path-to-file 'gbye example) = (list '/ 'tmp/ 'unix/)
(path-to-file 'README example) = (list '/)
```

```
(define (path-to-file fname dir)
  (if (eq? (directory-name dir) fname)
      null
      (let ((r (path-to-in-listing fname (directory-contents dir))))
        (if r (cons (directory-name dir) r) #f))))

(define (path-to-in-listing fname lofd)
  (cond
    ((null? lofd) #f)
    ((file? (car lofd))
     (if (eq? (file-name (car lofd)) fname)
         null
         (path-to-in-listing fname (cdr lofd))))
    (else
          (or (path-to-file fname (car lofd))
              (path-to-in-listing fname (cdr lofd))))))
```

Yes, these programs are instances of a common abstract program. Here we go:

```
(define (general-process use-result base)
  (local ((define (process-file fname dir)
            (if (eq? (directory-name dir) fname)
                base
                (use-result
                  (process-listing fname (directory-contents dir))
                  (directory-name dir))))

          (define (process-listing fname lofd)
            (cond
              ((null? lofd) #f)
              ((file? (car lofd))
               (if (eq? (file-name (car lofd)) fname)
                   base
                   (process-listing fname (cdr lofd))))
              (else
                (or (process-file fname (car lofd))
                    (process-listing fname (cdr lofd)))))))
    process-file))

(define exists-file?
  (general-process (lambda (x name) x) #t))

(define find-file
  (general-process (lambda (r name) (if r (cons name r) #f)) null))
```

Problem 2. [20pts] Prof. Daniel P. Friedman, a visitor from a northern land, is into abstracting functions. So he designed a general *sort* function, which he claims will remove all further needs for future 210 students to design a sort algorithm again:

```
(define sort
  (lambda (P? l)
    (local ((define qsort
              (lambda (alon)
                (if (null? alon)
                    null
                    (append
                      (qsort (partition P? (cdr alon) (car alon)))
                      (list (car alon))
                      (qsort (partition notP? (cdr alon) (car alon)))))))

            (define notP? (lambda (x y) (not (P? x y))))

            (define partition
              (lambda (p? alon N)
                (cond
                  ((null? alon) null)
                  (else (if (p? (car alon) N)
                            (cons (car alon) (partition p? (cdr alon) N))
                            (partition p? (cdr alon) N)))))))
      (qsort l))))
```

He claims that you can define the following three functions using his *sort* procedure:

Part 2:a. *sortd*, which accepts a list of numbers and sorts it in descending order.

```
(define (sortd l) (sort > l))
```

Part 2:b. *sort-stocks*, which accepts a list of *stock* records and returns it sorted in the ascending order of stock prices. Here is the definition of a stock record:

```
(define-structure (stock name price dividend))

(define (sort-stocks l)
  (sort (lambda (sr1 sr2) (< (stock-price sr1) (stock-price sr2))) l))
```

Part 2:c. *sort-stocks2*, which accepts a list of *stock* records and returns it sorted in ascending order of stock price and dividend. That is, a stock is "lower" than an other if it has a smaller prices *or* if it has the same price and a lower dividend.

```
(define (sort-stocks2 l)
  (sort
    (lambda (sr1 sr2)
      (or (< (stock-price sr1) (stock-price sr2))
          (and (= (stock-price sr1) (stock-price sr2))
               (< (stock-dividend sr1) (stock-dividend sr2)))))
    l))
```

Problem 3. [20pt] Chairman Vardi loves to verify programs. He claims that evaluating (f (list 2.0 3.0)) and (g (list 2.0 3.0) 0) yields the exact same results:

```
(define f
  (lambda (l)
    (cond
      ((null? l) 0)
      (else (+ (f (cdr l)) (expt (car l) 2))))))

(define g
  (lambda (l a)
    (cond
      ((null? l) a)
      (else (g (cdr l) (+ (expt (car l) 2) a))))))
```

Hand-evaluate the two expressions. Show all major steps, i.e., you may skip all arithmetical, list-processing, and conditional replacement steps.

```
  (f (list 2.0 3.0))
= (* 4.0 (f (list 3.0)))
= (+ 4.0 (+ 9.0 (f null)))
= (+ 4.0 (+ 9.0 0))
= (+ 4.0 9.0)
= 13.0

  (g (list 2.0 3.0) 0)
= (g (list 3.0) 4.0)
= (g null 13.0)
= 13.0
```

Is a computer really going to compute the same results for (f l) and (g l 0) if l is a list of numbers?

(f l) and (g l 0) compute the same mathematical function of l but add up the elements in the opposite order. Hence, if l contains floating-point numbers, the final results may differ because the order of addition matters for floating-point values.

Problem 4. [20pt] Ms. Myrtle was asked to design a root search algorithm for monotonic integer functions. An integer function is a function that accepts only integers and returns integers. Here is an example:

```
(define (f n) (if (integer? n) (- (* n n) 4) (error 'f "not an integer: ~s" n)))
```

An integer function is monotonic if for all $n < m$ it is also true that $(f\ n) < (f\ m)$.

The root search algorithm, *rooter*, accepts an integer function *f* that is guaranteed to be monotonic and two integers: *left* and *right*, which represent the left and right end of the interesting interval. Moreover,

```
(<= (f left) 0 (f right))
```

is guaranteed to hold. Mr. Corky Cartwright, who asked Ms. Myrtle to implement the search algorithm, suggested the following strategy:

1. Determine the length of the interval between *left* and *right*.
2. If it is large enough, let *mid* be (quotient (+ *left* *right*) 2).[1] Then, if
   (<= (f *left*) 0 (f *mid*))
   holds, continue the search between *left* and *mid*. Otherwise, continue the search between *mid* and *right*.
3. If the interval is too small to have a mid-point, check each element of the remaining, small interval.
4. If *f* has no root, return 'no-root.

Develop the function *rooter*.
Hint: Convince yourself that (*rooter* *f* 1 6) finds a root.

```
(define rooter
  (lambda (f left right)
      (if (<= (- right left) 1)
          (cond
            ((= (f left) 0) left)
            ((= (f right) 0) right)
            (else 'no-root))
          (let ((mid (quotient (+ left right) 2)))
            (if (<= (f left) 0 (f mid))
                (rooter f left mid)
                (rooter f mid right))))))
```

[1] Recall that quotient is integer division.

Problem 5. [20pts] Instructor Paco has implemented a simple Web browser for his course. The basic function of the browser is *get-page*. It accepts two numbers, which represent crucial properties of mouse clicks, and returns the corresponding Web page with the help of the current Web server:

```
(define get-page
  (lambda (x y wp)
    (fetch-page (translate-name x y wp) the-current-server)))
```

The function *translate-name* accepts two numbers and a Web page and returns a name (symbol). Then, the function *fetch-page* uses this name and the current Web server to retrieve the contents of the next page.

The problem is that *fetch-page* takes a long time to retrieve the contents of a page. Thus you are charged to extend *get-page* so that it remembers the names and the contents of all the pages it has retrieved before. If it is asked to return a Web page again, it can then use its "memory", which is significantly faster than using *fetch-page*.

```
(define get-page
  (lambda (x y wp)
    (local ((define name (translate x y wp))
            (define r (lookup name cache)))
      (if r ; or (entry? r)
          (entry-page r)
          (let ((page (fetch-page name current-server)))
            (begin
              (extend name page)
              page))))))

(define-structure (entry name page))

(define cache null)

(define extend
  (lambda (name page)
    (set! cache (cons (make-entry name page) cache))))

(define lookup
  (lambda (name cache)
    (cond
      ((null? cache) #f)
      (else (if (eq? (entry-name (car cache)) name)
                (car cache)
                (lookup name (cdr cache)))))))
```

Extra Credit Problem I: [3pts] Matthias wants you to evaluate the following program by hand:

```
(define f
  (lambda (x l)
    (local ((define g
              (lambda (l)
                (cond
                  ((null? l) null)
                  (else (cons (+ (car l) x) (g (cdr l))))))))
      (cons 3 (g l)))))

(f 3 (list 1))
```

You may skip all arithmetical, list-processing, conditional steps as well as those that replace placeholders by **define**d values. You must show all others.

```
= (local ((define g
            (lambda (l)
              (cond
                ((null? l) null)
                (else (cons (+ (car l) 3) (g (cdr l))))))))
    (cons 3 (g (list 1))))
= (define g1
    (lambda (l)
      (cond
        ((null? l) null)
        (else (cons (+ (car l) 3) (g1 (cdr l))))))) – lifted
  (cons 3 (g1 (list 1)))
= ; skip (define g1 . . . )
  (cons 3 (cond
            ((null? (list 1)) null)
            (else (cons (+ (car (list 1)) 3) (g1 (cdr (list 1)))))))
= ; skip (define g1 ...)
  (cons 3 (cons 4 (g1 null)))
= ; skip (define g1 . . . )
  (cons 3 (cons 4 null))
= (list 3 4)
```

Extra Credit Problem II: [3pts] Ian is worse. He asks "what is the result of the following program?"

```
(define f
  (local ((define c 0))
    (lambda (x)
      (if (symbol? x)
          c
          (begin
            (set! c (+ c 1))
            x)))))

(define g f)

(begin
  (f 0)
  (g 'show))
```

The result is 1 as the following evaluation shows. You did not have to show the evaluation to get full credit.

```
=
(define c 0)

(define f
  (lambda (x)
    (if (symbol? x)
        c
        (begin
          (set! c (+ c 1))
          x))))

(define g f)

(begin
  (f 0)
  (g 'show))

=
(define c 0)

(define f
  (lambda (x)
    (if (symbol? x)
        c
        (begin
```

```
          (set! c (+ c 1))
          x))))

(define g
  (lambda (x)
    (if (symbol? x)
        c
        (begin
          (set! c (+ c 1))
          x))))

(begin
  (f 0)
  (g 'show))

=
(define c 1)

(define f
  (lambda (x)
    (if (symbol? x)
        c
        (begin
          (set! c (+ c 1))
          x))))

(define g
  (lambda (x)
    (if (symbol? x)
        c
        (begin
          (set! c (+ c 1))
          x))))

(begin
  1
  (g 'show))

=
(define c 1)

(define f
  (lambda (x)
    (if (symbol? x)
```

```
        c
        (begin
          (set! c (+ c 1))
          x))))

(define g
  (lambda (x)
    (if (symbol? x)
        c
        (begin
          (set! c (+ c 1))
          x))))

(begin
 1)
= 1
```