Assignment 2
for
Advanced Program Paradigms, Semester 2 2013

Name: Yifei Pei
Student ID number: a1611648

Question 1 a)

The procedure foldr accumulates the operation to the second operand, so it keeps “cons” the “(car ls)” with “(cdr ls)” when “ls” is not null. After the operation, the “ls” will become null and the foldr will add the “z” operand, which is a null list, to the end of the new data structure created by “cons”. Therefore the result will remain the same as “ls”, which makes a copy of a list.

Question 1 b)

The Scheme code is:

```
(define (filtercons predicate)
  (lambda (x y)
    (if (predicate x)
        (cons x y)
        y)))
```

If “(car ls)” can pass the predicate condition, it will be “cons” into the result. If not, the op “filtercons” will just return the “(cdr ls)” for “foldr” to continue the recursion.

Question 1 c)

The Scheme code is:

```
(define (filter predicate xs)
  (foldr (filtercons predicate) '() xs))
```

Question 1 d)

By using the “filter” procedure which implement “foldr”, the result will give all the elements that fit for the predicate condition. Because the infinite stream is infinite, the procedure will never stop to give a non-infinite result to display.

Question 1 e)

By examining the conditions on infinite stream, the result will normally be infinite. A design that may work for the infinite stream filtering is to display the result or a set of results that have certain reference in the stream.

For example, a filter can get all the integers that are divisible by 7. We cannot order the filter to show all the results in integers, but we can order the filter to show the first integer that is larger than 100 and can be divided by 7 or to show the integers between 100 to 200 that can be divided by 7.

Question 2 a)

The Scheme code is:

```
(define (reorder order-stream data-stream)
  (cond ((stream-null? order-stream) the-empty-stream)
        ((stream-null? data-stream) the-empty-stream)
        (else (stream-cons (stream-ref data-stream (- (head order-stream) 1)) (reorder (tail
order-stream) data-stream)))))
```

Question 2 b)

(reorder integers data-stream)

returns a copy of data-stream. Because integers contains all the integers, the elements increment 1 from 1 to forever. The order-stream for the data-stream should be 1, 2, 3, 4, ... , which just follows the natural reference of data-stream. Therefore, the procedure will return a copy of data-stream.

(reorder data-stream integers)

returns a copy of data-stream too. Exactly, it returns the integers at the data-streams' element-th position. Because the integers contain all the integers from 1 incrementing 1 to the largest integer, the result of this procedure is the same as the data-stream.

In answers.scm:

```
(define d (stream 4 13 2 8))

(print-first-n (reorder integers d) 4)

(print-first-n (reorder d integers) 4)
```

give the same result of (4 13 2 8). Even their operations are different as described above, they return the same result.

Question 2 c)

The Scheme code is:

```
(define (print-first-n s n)
  (cond ((= n 0) (newline))
        ((= n 1) (begin
```

```
            (display (head s))
            (print-first-n (tail s) (- n 1))))
      (else (begin
            (display (head s) )
            (display ", ")
            (print-first-n (tail s) (- n 1))))
    )
 )
```

Question 2 d)

The result is

1, 2, 3, 8, 34, 377, 17711

The result of (print-first-n (tail fibs) 7) is “1, 2, 3, 5, 8, 13, 21”, which means the result of (reorder (tail fibs) (tail fibs)) contains 1st, 2nd, 3rd, 5th, 8th, 13th, and 21st elements of (tail fibs). (tail fibs) starts like 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, so the (print-first-n (reorder (tail fibs) (tail fibs)) 7) gives the result as described above.

Question 3 a)

The Scheme code is:

```
(define (make-account balance)

  (define counter
  (let ((number 0))
    (lambda ()
       (set! number (+ number 1))
     number)))

  (define (withdraw amount)
    (if (>= balance amount)
       (begin (set! balance (- balance amount))
             counter
             balance)
       "Insufficient funds"))

  (define (deposit amount)
    (set! balance (+ balance amount))
    counter
    balance
    )

  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                       m))))
  dispatch)
```

Question 3 b)

The Scheme code is:

```
(define (make-account balance)

  (define counter
  (let ((number 0))
    (lambda ()
       (set! number (+ number 1))
     number)))

  (define (withdraw amount)
    (if (>= balance amount)
       (begin (set! balance (- balance amount))
              counter
              balance)
       "Insufficient funds"))

  (define (deposit amount)
    (set! balance (+ balance amount))
    counter
    balance
    )

  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          ((eq? m 'balance) balance)
          ((eq? m 'transaction) counter)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                       m))))
  dispatch)
```

Question 4 a)

The Scheme code is:

```
(define (fib-mem x t)
  (cond ((get x t) (cons (get x t) x))
          ((< x 2) (fib-mem x (put x x t)))
          (else (fib-mem x (put x (+ (car (fib-mem (- x 2) t)) (car (fib-mem (- x 1) t))) t)))))
```

Question 4 b)