Eddy Lin (ELL22) and Nathan Kim (NJK24)
18 November 2019

Problem Set 5

2.1 Part One: Finite Sets

Problem 1

```
(define Cart
  (lambda ((a <list>) (b <list>))
    (foldr (lambda (a_el lst) ; for every element in a, ...
            ; append itself to every element of b
            (append
             (map (lambda (b_el)
                          (list a_el b_el))
                        b)
                  lst)) ;
           `()
           a)))

(define alist `(1 2 3 4 5))
(define blist `(6 7 8))

(display "Cartesian:\n")
(Cart alist blist)
; => ((1 6) (1 7) (1 8) (2 6) (2 7) (2 8) (3 6) (3 7) (3 8) (4 6) (4
7) (4 8) (5 6) (5 7) (5 8))
```

Problem 2

```
(define fcn?
  (lambda (S) ; S is a set of ordered pairs, assuming no duplicated
ordered pairs
    ; for each ordered pair in S,
    ; >> compare to the rest of S, to see if the same first entry
exists in other pairs
    ; base case: S is empty, return #t
    (cond [(null? S) #t]
          ; otherwise, compare (car (car S)), first entry) with
          ;    (map car (cdr S)) to see if it is a member
          ;     >> if it is a member, return #f
          [(member (car (car S))
                   (map car (cdr S))) #f]
          ; else, check the rest of the list
          [else
           (fcn? (cdr S))])))
```

```
(display "fcn?:\n")
(fcn? `((1 2) (2 4) (3 6) (4 8)))
; ==> #t
(fcn? `((1 2) (2 4) (3 6) (4 6)))
; ==> #t
(fcn? (Cart alist blist))
; ==> #f
(fcn? (Cart `(1 2 2) blist))
; ==> #f
(fcn? `((1 2) (2 4) (3 6) (3 8)))
; ==> #f
```

Problem 3

```
(define (remove-duplicates l)
  (cond [(null? l)
         '()]
        [(member (car l) (cdr l))
         (remove-duplicates (cdr l))]
        [else
         (cons (car l) (remove-duplicates (cdr l)))]))

(define pi1
  (lambda (S)
    (remove-duplicates (map car S))))

(define pi2
  (lambda (S)
    (remove-duplicates (map cadr S))))

(display "pi1:\n")
(pi1 `((1 2) (2 4) (3 6) (4 8)))
; ==> (1 2 3 4)
(pi1 `((1 2) (2 4) (3 6) (4 6)))
; ==> (1 2 3 4)
(pi1 (Cart alist blist))
; ==> (1 2 3 4 5)
(pi1 (Cart `(1 2 2) blist))
; ==> (1 2)

(display "pi2:\n")
(pi2 `((1 2) (2 4) (3 6) (4 8)))
; ==> (2 4 6 8)
(pi2 `((1 2) (2 4) (3 6) (4 6)))
; ==> (2 4 6)
(pi2 (Cart alist blist))
; ==> (6 7 8)
(pi2 (Cart `(1 2 2) blist))
; ==> (6 7 8)
```

Problem 4

```
(define diag
  (lambda (A)
    (foldr (lambda (el lst)
             (cons (list el el) lst))
           `()
           A)))
```

```
(display "diag:\n")
(diag alist)
; ==> ((1 1) (2 2) (3 3) (4 4) (5 5))
(diag blist)
; ==> ((6 6) (7 7) (8 8))
(diag `(10 20 30 40 50))
; ==> ((10 10) (20 20) (30 30) (40 40) (50 50))
```

Problem 5

```
(define diag?
  (lambda (D)
    (cond
      [(null? D) #t]
      [(not (= (caar D) (cadar D)))
       #f]
      [else
       (diag? (cdr D))])))

(display "diag?:\n")
(display "  #t tests:\n")
(diag? (diag alist))
; ==> #t
(diag? (diag blist))
; ==> #t
(diag? (diag `(10 20 30 40 50)))
; ==> #t
(diag? `((1 1) (2 2) (3 3) (4 4)))
; ==> #t
(display "  #f tests:\n")
(diag? `((0 1)))
; ==> #f
(diag? `((1 2)))
; ==> #f
(diag? `((1 1) (2 2) (3 3) (4 4) (5 6)))
; ==> #f
```

Problem 6

```
(define diag-inv
  (lambda (Delta)
    (foldr
     (lambda (elem lst)
       (cons (car elem) lst))
     `()
     Delta)))

(display "diag-inv:\n")
(diag-inv (diag alist))
; ==> (1 2 3 4 5)
(diag-inv (diag blist))
; ==> (6 7 8)
(diag-inv (diag `(10 20 30 40 50)))
; ==> (10 20 30 40 50)
(diag-inv `((1 1) (2 2) (3 3) (4 4)))
; ==> (1 2 3 4)
```

Problem 7

a)

Let A consist of two elements, a_1 and a_2; obviously, the power set of A, $\mathcal{P}$(A) consists of the following elements: (), (a_1), (a_2), (a_1 a_2)

b)

If A has one element, the power set $\mathcal{P}$(A) contains 2 elements.

c)

If A has three elements, the power set $\mathcal{P}$(A) contains 8 elements.

d)

If A has 0 elements, the power set $\mathcal{P}$(A) contains 1 element, the null list.

e)

```
(define powerset
  (lambda (A)
    (foldr (lambda (elem lst)
             (append
              lst
              (map (lambda (lst1)
                     (cons elem lst1))
                   lst)))
           `(())
           A)))

(display "powerset:\n")
(powerset blist)
; ==> (() (8) (7) (7 8) (6) (6 8) (6 7) (6 7 8))
(powerset `(1))
; ==> (() (1))
(powerset `(1 2))
; ==> (() (2) (1) (1 2))
(powerset `(1 2 3))
; ==> (() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))
(powerset `(1 2 3 4))
; ==> (() (4) (3) (3 4) (2) (2 4) (2 3) (2 3 4) (1) (1 4) (1 3) (1 3
4) (1 2) (1 2 4) (1 2 3) (1 2 3 4))
```

f)

If A contains n elements, the power set $\mathcal{P}$(A) contains 2^n elements.

g)

Claim: If A contains n elements, the power set $\mathcal{P}$(A) contains 2^n elements.
Proof: Let P(n) be true if the power set of A, $\mathcal{P}$(A), contains 2^n elements.
Proof by induction on n.

Base Case: let n=0. P(0) is true because the set containing 0 elements has a power set consists of just the null list, so it contains $1 = 2^0$ elements. Thus, the claim P(0) is true.

Inductive Hypothesis: For a fixed integer n, where $0 < k < n$, we assume that P(k) is true.

Inductive Step: NOS that P(n) is true.
Without loss of generality, a set of size n can be represented as two parts: the FIRST element and the REST of the set (which contains n-1 elements).

The power set of the list of size n and can be expressed in two halves: the first half contains the power set of the rest of the set (size n-1). The second half of the power set can be thought of as the same as the first half but with the FIRST element of the original set appended to the front of each element in the first half.

Thus, the power set of a list of size n will have double the size of the power set of a list of size (n-1). By the IH and because n-1<n, P(n-1) is true, so a list of n-1 elements has 2^{n-1} elements. From the above logic, adding another element makes a list containing n elements, doubling the size of the power set. Thus, a list of n elements has a power list of size $2*2^{n-1}=2^n$, so P(n) is true.

QED.

h)

It is called the power set because the power set $\mathcal{P}(A)$ contains 2^n elements, where n is the number of elements in A. Therefore, the number of elements in the power set is <u>2 to the power of the number of elements in A.</u>

2.2 Part Two: Infinite Sets

Problem II-1

```
(define flatten
  (lambda (A)
    ; if the first element of the stream A is a list with only one
value,
    (cond [(null? (cdr (stream-first A)))
           ; cons the list element...
           (stream-cons (car (stream-first A))
                        ; to the rest of the stream A
                        (flatten (stream-rest A)))]
          ; otherwise, if (stream-first A) is a list of more than one
ordered pair...
          [else
           ; cons the list element...
           (stream-cons (car (stream-first A))
                        ; to the flattened stream with...
                        (flatten
                         ; the first pair of the first list removed
                         (stream-cons (cdr (stream-first A))
                                      ; and the rest of the stream
                                      (stream-rest A))))])))

(define stream-cart
  (lambda (A B)
    ; count up i as all integers starting from 1.
    (flatten (for/stream ([i integers])
               ; count up k starting from 1 to i.
                (for/list ([k (stream->listn integers i)])
                  ; get A[k-1], starting from 0 and going up
                  (list (stream-ref A (- k 1))
                        ; get B[i-k], starting from i and going down
                        (stream-ref B (- i k))))))))

(display "stream-cart:\n")
(define triangular
  (stream-cons 3 (add-streams (add-streams (scale-stream ones 2)
integers) triangular)))
(define fibs
  (stream-cons 0
               (stream-cons 1
                                  (add-streams fibs (stream-rest fibs)))))

(stream->listn (stream-cart integers integers) 20)
; ==> ((1 1) (1 2) (2 1) (1 3) (2 2) (3 1) (1 4) (2 3) (3 2) (4 1) (1
5) (2 4) (3 3) (4 2) (5 1) (1 6) (2 5) (3 4) (4 3) (5 2))
(stream->listn (stream-cart ones integers) 20) ; note: ones is not a
set, but it is a stream
; ==> ((1 1) (1 2) (1 1) (1 3) (1 2) (1 1) (1 4) (1 3) (1 2) (1 1) (1
5) (1 4) (1 3) (1 2) (1 1) (1 6) (1 5) (1 4) (1 3) (1 2))
(stream->listn (stream-cart integers fibs) 20) ; note: fibs is not a
set, but it is a stream
; ==> ((1 0) (1 1) (2 0) (1 1) (2 1) (3 0) (1 2) (2 1) (3 1) (4 0) (1
3) (2 2) (3 1) (4 1) (5 0) (1 5) (2 3) (3 2) (4 1) (5 1))
(stream->listn (stream-cart triangular fibs) 20)
; ==> ((1 0) (1 1) (1 0) (1 1) (1 1) (1 0) (1 2) (1 1) (1 1) (1 0) (1
3) (1 2) (1 1) (1 1) (1 0) (1 5) (1 3) (1 2) (1 1) (1 1))
```

Problem II-2
Creating a procedure that takes a countably infinite stream and returns whether or not the stream represents a function is impossible. A function traversing this stream determining if each ordered pair is composed by a function (meaning not having two input values returning different outputs) would never halt because the stream is infinite, so it is impossible to write correctly. Another way to imagine this conclusion is that, if the procedure were to halt with #t early, there could always be an ordered pair value that came after the procedure call halted that violates the function condition, which would have made the return #f were it to have run infinitely.

Problem II-3

```
(define stream-member?
  (lambda (val S)
    (not (= (stream-count (lambda (elem)
                            (= elem val))
                          S)
            0))))

(define stream-pi1
  (lambda (S)
    (let ((proj (lambda (set_elem)
                  (car set_elem))))
      (stream-cons (proj (stream-first S))
                   (stream-filter
                    (lambda (val)
                      (if val #t #f))
                    (for/stream ([i integers])
                      ; if val is a member in a pair before i
                      (if
                       (stream-member?
                        ; val is equal to S[i]
                        (stream-ref (stream-map proj S) i)
                        ; take first values of S up to (but excluding)
S[i]
                        (stream-take (stream-map proj S) i))
                       ; do not add to stream
                       #f
                       (stream-ref (stream-map proj S) i))))))))

(define stream-pi2
  (lambda (S)
    (let ((proj (lambda (set_elem)
                  (cadr set_elem))))
      (stream-cons (proj (stream-first S))
                   (stream-filter
                    (lambda (val)
                      (if val #t #f))
                    (for/stream ([i integers])
                      ; if val is a member in a pair before i
                      (if
                       (stream-member?
                        ; val is equal to S[i]
                        (stream-ref (stream-map proj S) i)
```

```
                         ; take first values of S up to (but excluding)
S[i]
                         (stream-take (stream-map proj S) i))
                        ; do not add to stream
                        #f
                        (stream-ref (stream-map proj S) i))))))))

; infinite test case: (1 1) (1 2) (1 3) (1 4)...
(define infinite-test
  (stream-map (lambda (set_elem)
                (list 1 set_elem))
              integers))

(displayln "stream-pi1:")
(stream->listn (stream-pi1 (stream-cart integers integers)) 10)
; ==> (1 2 3 4 5 6 7 8 9 10)
(stream->listn (stream-pi1 (stream-cart integers (scale-stream
integers 2))) 10)
; ==> (1 2 3 4 5 6 7 8 9 10)
(stream->listn (stream-pi1 (stream-cart (scale-stream integers 2)
integers)) 10)
; ==> (2 4 6 8 10 12 14 16 18 20)
(stream->listn (stream-pi1 (stream-cart integers (stream-tail fibs
2))) 10)
; ==> (1 2 3 4 5 6 7 8 9 10)
(stream->listn (stream-pi1 infinite-test) 1) ; this stream only has
one value (which is 1)!
; ==> (1)

(displayln "stream-pi2:")
(stream->listn (stream-pi2 (stream-cart integers integers)) 10)
; ==> (1 2 3 4 5 6 7 8 9 10)
(stream->listn (stream-pi2 (stream-cart integers (scale-stream
integers 2))) 10)
; ==> (2 4 6 8 10 12 14 16 18 20)
(stream->listn (stream-pi2 (stream-cart (scale-stream integers 2)
integers)) 10)
; ==> (1 2 3 4 5 6 7 8 9 10)
(stream->listn (stream-pi2 (stream-cart integers (stream-tail fibs
2))) 10)
; ==> (1 2 3 5 8 13 21 34 55 89)
(stream->listn (stream-pi2 infinite-test) 10)
; ==> (1 2 3 4 5 6 7 8 9 10)
```

Problem II-4

```
(define stream-diag
  (lambda (A)
    (stream-map (lambda (elem)
                  (list elem elem))
                A)))

(display "stream-diag:\n")
(stream->listn (stream-diag integers) 10)
; ==> ((1 1) (2 2) (3 3) (4 4) (5 5) (6 6) (7 7) (8 8) (9 9) (10 10))
(stream->listn (stream-diag (stream-tail fibs 2)) 10) ; using
fibs[2:], a set
; ==> ((1 1) (2 2) (3 3) (5 5) (8 8) (13 13) (21 21) (34 34) (55 55)
(89 89))
```

Problem II-5

Creating a procedure that takes a countably infinite stream and returns whether or not the stream represents a diagonal set is impossible. A function traversing this stream determining if each ordered pair is composed as a diagonal set (meaning that each first value is equal to the corresponding second value) would never halt because the stream is infinite, so it is impossible to write correctly. Another way to imagine this conclusion is that, if the procedure were to halt with #t early, there could always be an ordered pair value that came after the procedure call halted that violates the diagonal condition, which would have made the return #f were it to have run infinitely.

Problem II-6

```
(define stream-diag-inv
  (lambda (Delta)
    (stream-map car Delta)))

(displayln "stream-diag-inv:")

(stream->listn (stream-diag-inv (stream-diag integers)) 20)
; ==> (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)
(stream->listn (stream-diag-inv (stream-diag (stream-tail fibs 2)))
20) ; using fibs[2:], a set
; ==> (1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
6765 10946)
```

Problem II-7

```
; stream-powerset-with-i returns a stream of all sublists of S[0:i]
that...
; ... contain S[i]
; input S: stream to get power set
; input i: index of stream to be in power set
; return a list of elements of P(S[0:i]) containing S[i]
(define stream-powerset-with-i
  (lambda (S i) ; S is set, return a stream of only elements of...
    ; ... the power set P(S[0:i]) that contain S[i]
     (map
      ; append S[i to every element of...
      (lambda (elem) ; elem is element of P(S[0:i-1])
        (cons (stream-ref S (- i 1)) elem))
      ; powerset of S[0:i-1],
      (powerset (stream->listn S (- i 1))))))


;(stream-powerset-with-i integers 4)
; ==> ((4) (4 3) (4 2) (4 2 3) (4 1) (4 1 3) (4 1 2) (4 1 2 3))

; stream-powerset creates the infinite power set of a infinite set...
; ... A, starting with `(), and performing (cons i elem), where
elem...
; ... is an element of the power set of A[0:i-1], where i is the ...
; ... index of S, to get S[i], starting from i=1 to infinity.
; input A: stream to make power set from
; returns a stream representing the infinite power set in the above
order.
(define stream-powerset
  (lambda (A)
    (stream-cons
     `()
     (flatten
      (for/stream ([i integers])
        (stream-powerset-with-i A i))))))

(displayln "stream-powerset:")
(stream->listn (stream-powerset integers) 20)
; ==> (() (1) (2) (2 1) (3) (3 2) (3 1) (3 1 2) (4) (4 3) (4 2) (4 2
3) (4 1) (4 1 3) (4 1 2) (4 1 2 3) (5) (5 4) (5 3) (5 3 4))
(stream->listn (stream-powerset (scale-stream integers 2)) 20)
```

```
; ==> (() (2) (4) (4 2) (6) (6 4) (6 2) (6 2 4) (8) (8 6) (8 4) (8 4
6) (8 2) (8 2 6) (8 2 4) (8 2 4 6) (10) (10 8) (10 6) (10 6 8))
(stream->listn (stream-powerset (stream-tail fibs 2)) 20) ; using
fibs[2:], a set
; ==> (() (1) (2) (2 1) (3) (3 2) (3 1) (3 1 2) (5) (5 3) (5 2) (5 2
3) (5 1) (5 1 3) (5 1 2) (5 1 2 3) (8) (8 5) (8 3) (8 3 5))
```