```
==========================================================================
==========================================================================
;;                                                                       |
;;           2017 Fall CS 145 Midterm II Review Notes V3.0               |
;;                                                                       |
;;                            Asst. 6~8                                  |
;;                      Lecture/Tutorial Notes                           |
;;                        Prof: Gordon Cormack                           |
;;                         ISA: Ashish Mahto                             |
;;                                                                       |
;;                           By David Duan                               |
;;                                                                       |
;;                              Outline                                  |
;;                    Part A: Asymptotic Analysis                        |
;;                    Part B: Abstract data type                         |
;;                    Part C: Higher order functions                     |
;;                    Part D: Other minor topics                         |
;;                    Part E: Appendix                                   |
;;                                                                       |
==========================================================================
;; If you find any errors (I'm sure there are a lot of them), please     |
;;   email j32duan@edu.uwaterloo.ca, David Duan. Thanks in advance.      |
==========================================================================
;;  Release Note                                                         |
;;                                                                       |
;;  V0.X                                                                 |
;;    Notes from lecture                                                 |
;;    Notes from tutorial                                                |
;;    Ashish's messages (which are, indeed, full of wisdom xd)           |
;;    My own research                                                    |
;;                                                                       |
;;  V1.X                                                                 |
;;    Raw note                                                           |
;;                                                                       |
;;  V2.0 Update: (Approx 1k lines)                                       |
;;    Finished part A, B, and C.                                         |
;;    Still need to finish foldr and foldl.                              |
;;    I'll do that tomorrow after asking Ashish questions.               |
;;                                                                       |
;;  V2.1 Update: (Approx 1.5k lines)                                     |
;;    Added foldr, foldl, and foldr vs. foldl.                           |
;;    Added or/list, disjoin/list, and f^n                               |
;;    Changed formatting                                                 |
;;                                                                       |
;;  V2.2 Update: (Approx 2k lines)                                       |
;;    Added Part D, including D1, D2, D3                                 |
;;    Added Part E, including E1, practice questions from TUT Oct.25     |
;;    Need to add: assignment 6~8                                        |
;;                                                                       |
;;  V2.3 Update: (Approx 2k lines)                                       |
;;    Modified A7                                                        |
;;    Modifed Part E, but haven't finished yet.                          |
;;    Modified Part C, added filter and map                              |
;;    Fixed typo                                                         |
;;                                                                       |
;;  V3.0 Release: 2155 lines                                             |
;;    Modified Part E                                                    |
;;    Fixed typo                                                         |
;;                                                                       |
==========================================================================
;; Big thanks to Teresa Kang and Steven Wong for pointing out countless
;; typos and logic errors.
==========================================================================
```

```
Part A: Asymptotic Analysis

Table of Contents
A1. Introduction
A2. Examples
A3. /subset and /strict-subset
A4. Order
A5. Examples
A6. Examples for using definitions to prove statements involving
    Big-O notation.
A7. Additional remarks

--------------------------------------------------------------------------
A1. Introduction.

  Defn: (Informal)
    f(x) /in O(g(x)) iff O(f(x)) <= O(g(x)).

  Defn: (Formal)
    There exists some constant x0 and c such that f(x) < c*g(x)
    for every x > x0.

  Defn: (About Big-O notation)
    O(g(n)) is the set of functions f(n) such that there exists
    constant c and n0 such that for all n >= n0, f(n) <= c*g(n).

  Remark:
    1. c*g(x) is an upper bound for f(x) when x > x0.
    2. O(g(x)) serves as a placeholder for all f(x) /in O(g(x)).
    3. If O(f(x)) <= O(g(x)), then for all h(x) /in O(f(x)),
       h(x) /in O(g(x)).

--------------------------------------------------------------------------
A2. Examples.

  Ex.
       x  /in   O(x)
     x^2  /in   O(x^2)
    f(x)  /in   O(f(x))

  Ex.
    O(f(x)) + k = O(f(x))
    O(f(x)) * k = O(f(x))

  Ex.
    O(k) = O(1)
    O(log_n(x)) = O(log(x))
    O(x) + O(x^2) + ... + O(x^n) = O(x^n)

--------------------------------------------------------------------------
A3. /subset and /strict-subset

  Thm:
    O(x) /subset O(x).
    O(x) /subset O(x^m) for any m >= 1.

  Defn:
    O(f(x)) /strict-subset O(g(x)) if
      a. O(f(x)) /subset O(g(x)), and
      b. O(g(x)) /not /subset O(f(x))

  Thm:
    O(a^x) /strict-subset O((a+e)^x)
```

```
    O(x^n) /strict-subset O(a^x) given that a > 1
```

A4. Order

```
  O(1) < O(log n)
       < O(n^k)
       < O(n)
       < O(n log n)
       < O(n^p), p>1
       < O(a^n), a>1
       < O(b^n), b>a
       < O(n!)
```

A5. Examples

Ex.

```
|%Racket_file
|
| (define (dedupe lst)
|   (cond
|     [(empty? lst) empty]
|     [(member (first lst) (rest lst))
|      (dedupe (rest lst))]
|     [else
|      (cons (first lst) (dedupe (rest lst)))]))
|
|%END
```

Note: The function member takes O(n), and we apply it to every element in the list, so overall the running time is O(n^2).

Ex.

```
|%Racket_file
|
| (define (append l1 l2)
|   (cond
|     [(empty? l1) l2]
|     [else (cons (first l1) (append (rest l1) l2))]))
|
|%END
```

Note: The function cons takes O(1), and we apply it to every element in list1, so overall the running time is O(n) where n is the size of list1.

Ex.

```
|%Racket_file
|
| (define (reverse l)
|   (cond
|     [(empty? l) empty]
|     [else (append (reverse rest l)
|                   (list first l))]))
|
|%END
```

Note: This is a bad list recursion. We apply append and reverse which both takes O(n) time on each element, so overall it will be O(n^2).

A6. Examples for using definitions to prove statements involving Big-O notation.

Ex. Prove "3n^2 + 6n is O(n^2)" using definition 1.

We need to start with "there exists c and n0", but we don't know what value would make the rest of the statement true, so we leave them as symbolic constants and accumulate information about it, as long as we make sure to specify its value by the end of the proof.

Remark:
This part is very similar to our proof in math 147 where we assume epsilon and delta/N exists, then do calculations until we find an appropriate pair.

The next step is doing algebra. Assume our c works, we want to find an appropriate n0.

Remark:
This is similar to assuming our N/delta works and work towards epsilon.

```
|
|     3n^2 + 6n   <=  c * n^2
|            6n   <=  (c-3) * n^2
|            6    <=  (c-3) * n
|       6 / (c-3) <=  n
|
```

At this point, we can let c be an arbitrary number that is greater than 3, so the left side would be positive. By the Archimedean principle, the set of natural numbers is not bounded, so we can always find some natural number n0 such that n >= n0 implies 6 / (c-3) <= n.

Since the last inequality satisfies the requirement of "3n^2 + 6n <= c*n^2 for all n>n0 for some c in real number and n in natural number", we can conclude that 3n^2 + 6n is indeed O(n^2).

Remark:
Our proof works only because everything we did was reversible (we performed the same operations on both sides). This is not always the case with inequalities. For example, if we use the face that we can increase the larger side of an inequality, we would not be able to work backward.

Ex. Prove that "3n^2-6n is not O(n)."

Express this using definition, we are saying "not" there exists c and n0 such that for all n>n0, 3n^2-6n <= cn. This is equivalent of saying, there exists some n > n0 that does not make our statement true.

Let c in R. We want to prove "there exists n such that for any c, 3n^2-6n is greater than cn."

|

```
    |   3n^2 - 6n >   cn
    |        3n^2 >   (c+6) * n
    |           n >   (c+6) / 3
    |

  There are two restrictions for n. First we have n > n0, and secondly
  we have n > (c+6)/3, so we can let n > max{n0, (c+6)/3}. This way, we
  have found an appropriate counter example of n that supports our
  statement.

!IMPORTANT! EXAM PREPARATION

  """
  At least one of the problem will be in the following form:

  For each of a number of pairs of definitions for f(x) and g(x),
  you must submit a module to do the following:

  ->  Provide a function (findx c x0) that consumes positive values c
      and x0 and produces x >= x0 such that f(x) > c * g(x). If no such
      value exists, produce 'impossible.

  ->  Provide values c and x0 for which (findx c x0) produces 'impossible.
      If there is no such pair of values, define c and x0 both to have
      the value 'none.

  If your implementation of findx is correct, you will be able to
  find suitable values of c and x0 iff f(x) is O(g(x)). In either case,
  you should justify your answer using embedded comments.
  """

  Note:
    To prove that f(x) is O(g(x)), assume c exists, find appropriate n,
    and rewrite the proof in the normal order. This is similar to limits.

    To prove that f(x) is not O(g(x)), try to prove the negation of
    it. That is, find an example of n such that for any c, f(x) > O(g(x)).

--------------------------------------------------------------------------
A7. Additional remarks

Remark:
  We sometimes use the equal sign to express the relationship,
  as in "3n^2 + 6n = O(n^2)". But O(n^2) is not a function nor
  an algebraic object, so this equal sign does not share any
  properties such as reflexivity as the normal mathematical
  equality sign.


Remark:
  Running time of common Racket functions

  Note:
    O(1): cons, first, rest, list, make-foo, foo-x, foo?, eq?

  Note:
    O(n), where n = (size x):
      (append x y) ;; independent of size of y
      (length x)
      (member e x)
      (reverse x)

  Note:
```

```
    O(n*T(n)), where n = (size x) and f has O(T(n)) time
      (foldr f e x)
      (foldl f e x) ;; requires O(n) space
      (map f x)

  Note:
    sort: O(n log n)
    quicksort: on average O(n log n), worst O(n^2)
    (equal? x y): O(n), where n = min((size x), (size y))

==========================================================================
==========================================================================
Part B: Abstract Data Type

  Defn:
    ADT: A set of values and a finite set of operations
         (functions), defined entirely by the behavior of the
         operators.

  Remark:
    Constrast to: Concrete Data Types
      - Defined by its representations.
      - Has infinite number of operations.
      - For example, list of number '(10, 20, 30)

--------------------------------------------------------------------------
Ex. Abstract implementation of set.

  Remark: Big thanks to Teresa for pointing out a crucial mistake.

  Note:
    We represent the set as a function.

     |%Racket_file
     |
     | (provide make-empty-set insert-set member-set)
     |
     | (define (make-empty-set) (lambda (e) false))
     |
     | (define (member-set e s) (s e))
     |
     | (define (insert-set e s)
     | (lambda (x)
     |   (if (= e x) true (s x))))
     |
     | ;; You need to define what equality means.
     |
     |%END

  Note:
    1. This implementation is essentially a member function,
    it tests if the argument passed in is in the set or not.

    2. (make-empty-set) creates a lambda function which returns
    false no matter what argument is passed in.

    3. (member-set e s) returns the result of function application
    s applied onto e, ie. returns (s e).

    4. (insert-set e s) returns a lambda function which takes in
    one argument; its output depends on if the argument x is equal
    to the argument e we passed in in the first place. The variable
    e is what's stored in the lambda function. The variable x is
```

```
    newly-consumed and will be checked if it's in the set already.

    5. If it is still confusing, let's walk through some examples.

-------------------------------------------------------------------------
    Ex. (member-set 5 (make-empty-set))
        This function checks if 5 is in the empty set.

      |%STEPPER
      |
      |   (member-set 5 (make-empty-set))
      |   ((make-empty-set) 5)
      |   ((lambda (e) false) 5)
      |   false
      |
      |%END

-------------------------------------------------------------------------
    Ex. (insert-set 5 (make-empty-set))
      This function inserts 5 into an empty set.

      |%STEPPER
      |
      | (insert-set 5 (make-empty-set))
      |
      | ;; local_var: e = 5, s = (make-empty-set)
      |   ((lambda (e s)
      |   (lambda (x)
      |     (if (= e x) true (s x))))
      |   5 (make-empty-set))
      |
      | ;; (make-empty-set) = (lambda (e) false)
      | (lambda (x) (if (= 5 x) true ((lambda (e) false) x)))
      |
      |%END

      Note: Instead of returning a value, we got a lambda function.
          This function consumes an additional variable x, then
          compares if it equals 5. If yes, the function returns true.
          Otherwise our variable x gets passed into the next layer
          of lambda function, which in this case is the empty
          set function that always returns false.

-------------------------------------------------------------------------
    Ex. (insert-set 3 (insert-set 5 (make-empty-set)))
      This function inserts 3 into a set containing 5.

      |%STEPPER
      | (insert-set 3 (insert-set 5 (make-empty-set)))
      |
      |   ;; local_var: e = 3, s = (insert-set 5 (make-empty-set))
      | ((lambda (e s)
      |   (lambda (x)
      |     (if (= e x) true (s x))))
      |   3 (insert-set 5 (make-empty-set)))
      |
      | ;; substitution for (insert-set 5 (make-empty-set))
      | ((lambda (e s)
      |   (lambda (x)
      |     (if (= e x) true (s x))))
      |   3
      |   (lambda (e s)
      |     (lambda (x)
```

```
|       (if (= e x) true (s x))))
|     5 (make-empty-set))
|
| ;; substitution for (make-empty-set)
| ((lambda (e s)
|   (lambda (x)
|     (if (= e x) true (s x))))
|   3
|   (lambda (e s)
|     (lambda (x)
|       (if (= e x) true (s x))))
|     5
|     (lambda (e) false))
|
| ;; Now 5 gets consumed to produce a new function
| ((lambda (e s)
|   (lambda (x)
|     (if (= e x) true (s x))))
|   3
|   (lambda (x)
|     (if (= 5 x) true ((lambda (e) false) x))))
|
| ;; Now 3 gets consumes to produce a new function
| (lambda (x)
|   (if (= 3 x)
|     true
|     (lambda (x)
|       (if (= 5 x) true ((lambda (e) false) x))) x))
|
|%END
```

Note:

Take a look at what our output function does. This lambda function (call it lambda1) takes in 1 argument x and compares x with 3. If they are equal, we return true. Otherwise we call the next lambda function (call it lambda2), which takes in the same x and compare it with 5. If x = 5 then return true, otherwise call the next lambda function, in our case it's the empty set/function so it always returns false.

Note that every time the if statement fails, we are calling (s x). The s is the lambda function, and we are feeding it with an argument x, which is the variable right after the close bracket of lambda function.

Ex. (insert-set 5 (insert-set 3 (insert-set 5 (make-empty-set))))

This function (tries to) insert the number 5 into a set containing number 3 and 5.

```
|%STEPPER
|
|   (insert-set 5 (insert-set 3 (insert-set 5 (make-empty-set))))
|
| ;; local_var: e = 5, s = (insert-set 3 (insert-set 5
| ;;                        (make-empty-set))))
| ((lambda (e s)
|   (lambda (x)
|   (if (= e x) true (s x))))
|   5 (insert-set 3 (insert-set 5 (make-empty-set))))
|
| ;; substitution for (insert-set 3 (insert-set 5 (make-empty-set)))
```

```
((lambda (e s)
  (lambda (x)
  (if (= e x) true (s x))))
  5
  ((lambda (e s)
       (lambda (x)
       (if (= e x) true (s x))))
       3
       (insert-set 5 (make-empty-set))))

;; substition for both insert-set 5 and the empty set.

((lambda (e s)
  (lambda (x)
  (if (= e x) true (s x))))
  5
  ((lambda (e s)
        (lambda (x)
        (if (= e x) true (s x))))
       3
       ((lambda (e s)
       (lambda (x)
        (if (= e x) true (s x))))
          5
          (lambda (e) false))))

;; now consumes 5 to produce a new function

((lambda (e s)
  (lambda (x)
  (if (= e x) true (s x))))
  5
  ((lambda (e s)
        (lambda (x)
        (if (= e x) true (s x))))
       3
       (lambda (x)
       (if (= 5 x) true
         ((lambda (e) false)) x))))

;; consumes 3

((lambda (e s)
  (lambda (x)
  (if (= e x) true (s x))))
  5
  (lambda (x)
  (if (= 3 x) true
    ((lambda (x)
      (if (= 5 x) true
      ((lambda (e) false) x))) x))))

;; consumes 5

  (lambda (x)
  (if (= 5 x) true
    ((lambda (x)
    (if (= 3 x) true
        ((lambda (x)
         (if (= 5 x) true
          ((lambda (e) false) x))) x))) x)))

%END
```

```

      Note:
        This lambda has similar logic as the last one, except
        it actually contains two functions checking for 5.
        Nevertheless it doesn't affect the set operations since
        if the argument is 5, we will directly return true.

  Note: To further see that this implementation is like a
      member function, try ((insert 5 (make-empty-set)) 5) and see
      what would happen.

===========================================================================
===========================================================================
Part C: Higher-order function

Table of Contents
C1. compose
C2. disjoin
C3. foldr
C4. foldl
C5. foldr vs. foldl
C6. or/list
C7. disjoin/list
C8: f^n
C9. currying
C10. map
C11. filter

---------------------------------------------------------------------------
C1. compose

  Note:
    Consider the contract (f : (b : B) -> C).
    This is the type contract for a generic one-argument function.
    It takes an argument b of type B and returns an output of type C.

    Now suppose we have another function g with type contract
    (g : (a : A) -> B). We can define a new function called
    compose, which represents the function composition of
    function g and f:

  Implementation:

  ;; Type contract:
  ;;  (compose : (f : B -> C) -> (g : A -> B) => (A -> C))

    |%Racket_file
    |
    | (define compose
    |   (lambda (f g)
    |     (lambda (x)
    |       (f (g x)))))
    |
    |%END

  Remark:
    - Arguments:
        f : B -> C
        g : A -> B
    - Return:
        A lambda function
          - Arguments:
              x : A
```

```
    - Return:
        (f (g x)) : C
```

Ex.

```
|%Racket_file
|
| > (define neg-sqrt (compose - sqrt))
| > neg-sqrt
| #<procedure:compose>
|
| > (neg-sqrt 3)
| -1.732.. ;; first (sqrt 3), then (- (sqrt 3))
|
| > ((compose sqrt -) 3)
| 0+1.732..i ;; first (- 3), then (sqrt (- 3))
|
|%END
```

Note:

Compare and contrast the following function with the compose defined above.

Implementation:

```
;; Type contract:
;;  (compose-then-compute : (f : B -> C) -> (g : A -> B) -> (a : A) => C)

|%Racket_file
|
| (define compose-then-compute
|   (lambda (f g a)
|     (f (g a))))
|
|%END
```

Remark:

```
- Arguments:
    f : B -> C
    g : A -> B
    a : A
- Return:
    (f (g a)) : C
```

Remark:

The difference is that compose returns a function, where compose-then-compute returns a value.

C2. disjoin

Note:

The function disjoin consumes two functions that returns booleans and returns a function that returns the disjunction (or-value).

Implementation:

```
|%Racket_file
|
| (define disjoin
|   (lambda (f g)
|     (lambda (x)
```

```
|        (or (f x) (g x)))))
|
|%END
```

Remark:
- Arguments:
  f : A -> Bool
  g : A -> Bool
- Return:
  A function
  - Argument:
    x : A
  - Return:
    (or (f x) (g x)) : Bool

Ex.
Check if the list has length less than 2?

First thought:

```
|            (or (empty? lst)
|                (empty? (rest lst)))
```

Note that the second argument inside the or function is applying two 1-arg functions onto the same argument, thus we can rewrite it using compose:

```
|            (empty? (rest lst)
|            ;; is equivalent to
|            ((compose empty? rest) lst)
```

Also, since we are applying two functions onto the same argument ((compose empty? rest) produces a function!), we can rewrite the whole thing using compose:

```
|            (or (empty? lst)
|            (compose empty? rest) lst)
|            ;; is equivalent to
|            ((disjoin empty? (compose empty? rest)) lst)
```

This way, we can define our new function:

Implementation:

```
|%Racket_file
|
| (define len<2?
|   (disjoin empty? (compose empty? rest)))
|
| (len<2? '(1 2)) => False
|
|%END
```

C3: foldr

Thm:
Main characteristic of foldr: PURE RECURSION

Implementation:

```
|%Racket_file
|
| (define (foldr f z ls)
|   (cond
|     [(empty? ls) z]
|     [else (f (first ls) (foldr f z (rest ls)))]))
|
|%END
```

Ex.

```
|%Racket_file
|
| (foldr cons '() '(1 2 3))
|
| (foldr + 0 '(1 2 3))
|
|%END

|%STEPPER
|
| (foldr cons '() '(1 2 3))
| (cons 1 (foldr cons '() '(2 3)))
| (cons 1 (cons 2 (foldr cons '() '(3))))
| (cons 1 (cons 2 (cons 3 (foldr cons '() '()))))
| (cons 1 (cons 2 (cons 3 '())))
| (cons 1 (cons 2 '(3)))
| (cons 1 '(2 3))
| '(1 2 3)
|
| (foldr + 0 '(1 2 3))
| (+ 1 (foldr + 0 '(2 3)))
| (+ 1 (+ 2 (foldr + 0 '(3))))
| (+ 1 (+ 2 (+ 3 (foldr + 0 '()))))
| (+ 1 (+ 2 (+ 3 0)))
| (+ 1 (+ 2 3))
| (+ 1 5)
| 6
|
|%END
```

Remark:
As you can see, in each step, the second argument for function f is the recursive function application. The calculation starts when the function reaches the end of the list (which returns the init value z), and fold from right towards left.

Thm:
Because of this, we can rewrite foldr like this:

```
|%Racket_file
|
| (foldr f z '(e1 e2 e3 e4 ... eN))
|
|%END
```

Step1: Expand the foldr

```
|%STEPPER
|
| (foldr f z '(e1 e2 e3 e4 ... eN))
| (f e1 (foldr f z '(e2 e3 e4 ... eN)))
```

```
      | (f e1 (f e2 (foldr f z '(e3 e4 ... eN))))
      | (f e1 (f e2 (f e3 (foldr f z '(e4 ... eN)))))
      | (f e1 (f e2 (f e3 (f e4 (foldr f z '(... eN))))))
      | ...
      | (f e1 (f e2 (f e3 (f e4 (...(f eN (foldr f z '()))))))))
      |
      |%END

     Remark:
       Note that (foldr f z '()) returns z, so we can start
       doing calculations now.

     Step2: Evaluate expressions.

       Let rk be the result when applying f onto ek, r(k+1).
       Note that when k = N, e(k+1) = z.

      |%STEPPER
      |
      | (f e1 (f e2 (f e3 (f e4 (...(f eN (foldr f z '()))))))))
      | (f e1 (f e2 (f e3 (f e4 (...(f eN z))))))
      | ...
      | (f e1 (f e2 (f e3 (f e4 r5))))
      | (f e1 (f e2 (f e3 r4)))
      | (f e1 (f e2 r3))
      | (f e1 r2)
      | r1
      |
      |%END

 -------------------------------------------------------------------------
 C4: foldl

   Thm:
     Main characteristic of foldl: ACCUMULATIVE RECURSION

   Implementation:

      |%Racket_file
      |
      | (define (foldl f z ls)
      |   (define (calc ls acc)
      |     (cond
      |       [(empty? ls) acc]
      |       [else (calc (rest ls)
      |                   (f (first ls) acc))]))
      |     (calc ls z))
      |
      |%END

   Ex.

      |%Racket_file
      |
      | (foldl cons '() '(1 2 3))
      |
      | (foldl + 0 '(1 2 3))
      |
      |%END

      |%STEPPER
      |
      | (foldl cons '() '(1 2 3)
```

```
      | (calc '(1 2 3) '())
      | (calc '(2 3) (cons 1 '()))
      | (calc '(3) (cons 2 (cons 1 '())))
      | (calc '() (cons 3 (cons 2 (cons 1 '()))))
      | (cons 3 (cons 2 (cons 1 '())))
      | (cons 3 (cons 2 '(1)))
      | (cons 3 '(2 1))
      | '(3 2 1)
      |
      | (foldl + 0 '(1 2 3))
      | (calc '(1 2 3) 0)
      | (calc '(2 3) (+ 1 0)
      | (calc '(3) (+ 2 (+ 1 0)))
      | (calc '() (+ 3 (+ 2 (+ 1 0))))
      | (+ 3 (+ 2 (+ 1 0)))
      | (+ 3 (+ 2 1))
      | (+ 3 3)
      | 6
      |
      |%END

    Remark:
      In foldl, the second argument for each function application
      of f is the accumulative result.

    Thm:
      We can rewrite foldl like this:

        |%Racket_file
      |
      | (foldl f z '(e1 e2 e3 e4 ... eN))
      |
      |%END

      Step1: Expand the foldl / Use the helper

      |%STEPPER
      |
      | (foldl f z '(e1 e2 e3 e4 ... eN))
      | (calc '(e1 e2 e3 e4 ... eN) z)
      | (calc '(e2 e3 e4 ... eN) (f e1 z))
      | (calc '(e3 e4 ... eN) (f e2 (f e1 z))
      | (calc '(e4 ... eN) (f e3 (f e2 (f e1 z))
      | (calc '(... eN) (f e4 (f e3 (f e2 (f e1 z))
      | (calc '() (f eN (... (f e4 (f e3 (f e2 (f e1 z)))))))
      |
      |%END

      Remark:
        Note that (calc '() <acc> ) returns <acc>, so we can start
        doing calculations now.

      Step2: Evaluate expressions.

        Let rk be the result when applying f onto ek, r(k-1).
        Note that when k = 1, e(k-1) = e0 = z.

      |%STEPPER
      |
      | (f eN (... (f e4 (f e3 (f e2 (f e1 z)))))))
      | (f eN (... (f e4 (f e3 (f e2 r1)))))
      | (f eN (... (f e4 (f e3 r2))))
      | (f eN (... (f e4 r3)))
```

```
      | (f eN r(N-1))
      | rN
      |
      |%END
```

C5: foldr vs. foldl

Thm: Equivalence
For most simple pure-functional functions, (foldr f z ls) is equivalent to (foldl f z (reverse ls)).

Ex.

```
      |%Racket_file
      |
      | (foldr f z '(e1 e2 e3 e4 e5))
      | (foldl f z '(e5 e4 e3 e2 e1))
      |
      |%END

      |%STEPPER
      |
      | (foldr f z '(e1 e2 e3 e4 e5))
      | (f e1 (foldr f z '(e2 e3 e4 e5)))
      | (f e1 (f e2 (foldr f z '(e3 e4 e5))))
      | (f e1 (f e2 (f e3 (foldr f z '(e4 e5)))))
      | (f e1 (f e2 (f e3 (f e4 (foldr f z '(e5))))))
      | (f e1 (f e2 (f e3 (f e4 (f e5 (foldr f z '()))))))
      | (f e1 (f e2 (f e3 (f e4 (f e5 z)))))
      |
      | (foldl f z '(e5 e4 e3 e2 e1))
      | (calc '(e5 e4 e3 e2 e1) z)
      | (calc '(e4 e3 e2 e1) (f e5 z))
      | (calc '(e3 e2 e1) (f e4 (f e5 z)))
      | (calc '(e2 e1) (f e3 (f e4 (f e5 z))))
      | (calc '(e1) (f e2 (f e3 (f e4 (f e5 z)))))
      | (calc '() (f e1 (f e2 (f e3 (f e4 (f e5 z))))))
      | (f e1 (f e2 (f e3 (f e4 (f e5 z)))))
      |
      |%END
```

Remark:
They produce the same outcome!!

Thm: Memory cost
In terms of memory costs, foldl <= foldr.
The main reason is that we use an accumulator for foldl, where in foldr we need memory to store everything.

C6: or/list

Note:
Suppose we want to define a function which takes the "or" value of all elements in a list.

Implementation:

```
;; Type contract:
;; (or/list (listof Bool) -> Bool)
;;
```

```
  ;; Example:
  ;; (or/list '(true false false)) -> true

    |%Racket_file
    |
    | (define or/list
    |   (lambda (bool-list)
    |     (foldr
    |       (lambda (bool acc-val)
    |               (or bool acc-val))
    |       false
    |       bool-list)))
    |
    |%END

  Remark:
    For main function or/list
    - Argument:
        bool-list: listof b : Bool
    - Return:
        b : Bool created by foldr

  Remark:
    For function foldr
    - Argument:
        A function
          - Argument:
              bool : Bool, an element from bool-list
              acc-val : Bool, accumulator
          - Return:
              (or bool acc-val) : Bool
        false : Bool

          Remark:
            For or function, the base case is false.

        bool-list : listof b : Bool
    - Return:
        c : Bool

--------------------------------------------------------------------------
C7: disjoin/list

  Note:
    We want to disjoin all functions in a list.

  Implementation:

    |%Racket_file
    |
    | (define disjoin/list
    |   (lambda (list-of-func)
    |     (foldr
    |       disjoin
    |       (lambda (x) false)
    |       list-of-func)))
    |
    |%END

  Remark:
    For main function disjoin/list
    - Argument:
        listof f : A -> Bool
```

```
      - Returns:
          A function created by foldr

    Remark:
      For function foldr
      - Arguments:
          disjoin : (f : A -> Bool) (g : A -> Bool) -> (h : A -> Bool)
          (lambda (x) false)

          Remark:
            Recall that the base case for or is false, so the initial value
            of foldr inside the or is false. In this case, the base case
            for disjoin is also false, but since we are working with
            functions instead of Booleans, we need a function version of
            "False". In a sense, (lambda (x) false) can be seens as the
            function version of false.

          listof f : A -> Bool
      - Return:
          A function h : A -> Bool

  -------------------------------------------------------------------------
  C8: f^n: repetitive application

    Note:
      If you want to apply one function many times to one list, you can
      let this f^n function help you.

    Implementation:

      |%Racket_file
      |
      | (define f^n
      |   (lambda (f n)
      |     (foldr
      |       compose
      |       identity
      |       (make-list n f))))
      |
      |%END

    Remark:
      For main function f^n
      - Argument:
        f : A -> B
        n : Nat
      - Return:
        A function g : A -> B created by foldr.

    Remark:
      For foldr
      - Argument:
        compose : (f : B -> C) (g : A -> B) -> (A -> C)
        identity: (f : X -> X)

        Remark:
          Recall that in disjoin/list we used (lambda (x) false) as the
          function version of false. Here, we want to compose a list of
          functions together, and we need a start point. This is very
          much alike to + and *. For those two functions, when we fold them,
          we used their identities as the initial value inside foldr
          (0 and 1, respectively). Now we want to find the functional
          version of identity -- the function identity returns the same
```

thing it consumes, which is the ideal function version of identity we are looking for.

```
(make-list n f) : listof f : A -> B, with length n.
```

Ex.

```
%Racket_file

> (f^n rest 2)
#<procedure:compose>

%END
```

```
%STEPPER

(f^n rest 2)

((lambda (f n)
   (foldr
     compose
     identity
     (make-list n f))) rest 2)

(foldr
   compose
   identity
   (make-list 2 rest))

(foldr
   compose
   identity
   '(rest rest))

(compose rest (foldr compose identity '(rest)))

(compose rest (compose rest (foldr compose identity '())))

(compose rest (compose rest identity))

(compose rest rest)

((lambda (f g)
   (lambda (x)
     (f (g x)))) rest rest)

(lambda (x) (rest (rest x)))

%END
```

Ex.

```
%Racket_file

> ((f^n rest 2) '(1 2 3 4 5))
'(3 4 5)

%END
```

```
%STEPPER

((f^n rest 2) '(1 2 3 4 5))

```

```
| ...
|
| ((lambda (x) (rest (rest x))) '(1 2 3 4 5))
|
| (rest (rest '(1 2 3 4 5)))
|
| (rest '(2 3 4 5))
|
| '(3 4 5)
|
|%END
```

Ex.

```
|%Racket_file
|
| > (define cons-7
|     (lambda (lst)
|       (cons 7 lst)))
|
| > ((f^n cons-7 3) '(1 2 3))
| '(7 7 7 1 2 3)
|
|%END
```

Look at (f^n cons-7 3) first.

```
|%STEPPER
|
| (f^n con-7 3)
|
| ((lambda (f n)
|   (foldr
|     compose
|     identity
|     (make-list n f))) cons-7 3)
|
| (foldr
|   compose
|   identity
|   (make-list 3 cons-7))
|
| (foldr
|   compose
|   identity
|   '(cons-7 cons-7 cons-7))
|
| (compose cons-7 (compose cons-7 cons-7))
|
| ((lambda (x) (cons-7 (cons-7 (cons-7 x)))
|
|%END
```

Now back to ((f^n cons-7 3) '(1 2 3))

```
|%STEPPER
|
| ((f^n cons-7 3) '(1 2 3))
|
| ((lambda (x) (cons-7 (cons-7 (cons-7 x)))) '(1 2 3))
|
| (cons-7 (cons-7 '(7 1 2 3)))
|
```

```
      | (cons-7 '(7 7 1 2 3))
      |
      | '(7 7 7 1 2 3)
      |
      |%END
```

C9: currying

Note:
We can curry a 2-arg function into a "wrapped" 1-arg function.

Note:
```
  Before: (f : A -> B -> C)
  After: (f' : A -> (B -> C))
```

Note:
```
  Before: (A -> B -> C)
  After: (A -> (B -> C))
```

Implementation:

```
  |%Racket_file
  |
  | (define curry
  |   (lambda (f)
  |     (lambda (a)
  |       (lambda (b)
  |         (f a b)))))
  |
  |%END
```

Remark:
```
  - Argument:
      f : (A -> B -> C)
  - Return:
      A function
        - Argument
            a : A
        - Return
            A function
              - Argument
                  b : B
              - Return
                  (f a b) : C
```

Ex.
```
  |%Racket_file
  |
  | > (+ 1 2)
  | 3
  | > (+ 1)
  | 1
  | > (curry +)
  | #<procedure>
  | ;;  At this point, we have only provided "f", so we get a
  | ;;  procedure
  | ;;            (lambda (a)
  | ;;              (lambda (b)
  | ;;                (lambda (+ a b))))
  |
  | > ((curry +) 1)
```

```
      | #<procedure>
      | ;;  Now we have provided both f and a, so we have a
      | ;;  "partially applied" function. Our output lambda function
      | ;;  looks like this:
      | ;;                    (lambda (b)
      | ;;                      (lambda (+ 1 b)))
      |
      | > (((curry +) 1) 2)
      | 3
      | ;; We have finally provided all three arguments, so we get
      | ;; a value back in return. In fact, ((curry +) 1) is equivalent
      | ;; to the function add1.
      |
      |%END
```

Note:
Let's look at the stepper.

```
      |%STEPPER
      |
      | ((curry +) 1)
      |
      | (((lambda (f)
      |     (lambda (a)
      |       (lambda (b)
      |         (f a b)))) +) 1)
      |
      | ((lambda (a)
      |   (lambda (b)
      |     (+ a b))) 1)
      |
      | (lambda (b) (+ 1 b))
      |
      |%END
```

Remark:
You can see how at every step, one more argument gets accumulated into the final expression. (f a b) eventually turns into (lambda (b) (+ 1 b)) in a few steps.

Note:
In a sense, what we've done is "storing" some inputs inside a lambda function. Consider the following example:

Implementation:

```
      |%Racket_file
      |
      | (define curry*
      |   (lambda (a)
      |     (lambda (b)
      |       (lambda (f)
      |         (f a b)))))
      |
      |%END
```

Note:
What if we want to uncurry a function?
Let's start with type contract. We want to reverse the curry process.

Note:
For curry:

```

        Before: (f : A -> B -> C)
        After: (f' : A -> (B -> C))

        Before: (A -> B -> C)
        After: (A -> (B -> C))

      Then for uncurry:

        Before: (f' : A -> (B -> C))
        After: (f : A -> B -> C)

        Before: (A -> (B -> C))
        After: (A -> B -> C)

    Implementation:

      |%Racket_file
      |
      | (define uncurry
      |   (λ (f)
      |     (λ (a b)
      |       ((f a) b))))
      |
      |%END

    Remark:
      - Arguments:
          Warning: f must be a curried function.
          A function
            - Arguments:
                a : A
            - Return:
                A function
                  - Arguments:
                      b : B
                  - Return:
                      (f a b) : C
      - Return:
          A function
            - Arguments:
                a : A
                b : B
            - Return:
                ((f a) b) : C

      Ex.

        |%Racket_file
        |
        | (define curried-add
        |   (curry +))
        |
        | (uncurry curried-add)
        |
        |%END

        |%STEPPER
        |
        | (uncurry curried-add)
        |
        | ;; rewrite the uncurry function
        | ((lambda (f)
```

```
        |   (lambda (a b)
        |     ((f a) b))) curried-add)
        |
        | ;; rewrite the curried-add function
        | ((lambda (f)
        |   (lambda (a b)
        |     ((f a) b)))
        |   ((lambda (f)
        |     (lambda (a)
        |       (lambda (b)
        |         (f a b)))) +))
        |
        | ;; pass in +
        | (lambda (f)
        |   (lambda (a b)
        |     ((f a) b))
        |   (lambda (a)
        |     (lambda (b)
        |       (+ a b))))
        |
        | ;; pass in the lambda function as f
        | (lambda (a b)
        |   (((lambda (a)
        |       (lambda (b)
        |         (+ a b)))
        |     a)
        |    b))
        |
        |%END
```

Remark:

```
      - Argument:
          a : A
          b : B
      - Return:
          ((A function
            - Argument:
                a : A
            - Return:
                A function
                  - Argument:
                      b : B
                  - Return:
                      (+ a b) : C) applied onto a and b) : C
```

Warning:

This part is very messy. Please read carefully and make sure you understand it fully.

C10. map

Implementation:

```
    |%Racket_file
    |
    | (define map
    |   (lambda (f ls)
    |     (foldr
    |       (lambda (x acc) (cons (f x) acc))
    |       '()
    |       ls))))
    |
```

```
    |%END
```

Ex.

```
    | (map add1 '(1 2 3)) => '(2 3 4)
```

C11. filter

Implementation:

```
    |%Racket_file
    |
    | (define filter
    |   (lambda (f ls)
    |     (foldr
    |       (lambda (x acc) (if (f x) (cons x acc) acc))
    |       '()
    |       ls)))
    |
    |%END
```

==

==

Part D. Other Minor Topics.

Table of contents

D1. Modules

Theory: Modules

```
    |%Racket_file
    |
    | #lang racket
    |
    | ;; (sumto n) sums the integers from 0 to n, where n
    | ;;           is a non-negative integer.
    | ;; running time: O(n)
    |
    | (provide sumto)
    |
    | (define (sumto n)
    |   (if (zero? n) 0 (+ n (sumto (sub1 n)))))
    |
    |%END
```

Note: Provide statement
This provide statement is called a "directive", which
tells Racket that other files may use this.

```
    |%Racket_file
    |
    | #lang racket
    |
    | ;; running time: O(n^2)
    |
    | (require "sum.rkt")
```

```
|
| (define (sumsumto m)
|   (if (zero? m) 0 (+ (sumto m) (sumsumto (sub1 m)))))
|
| ;; Don't panic. This is just a double summation.
|
|%END
```

Note: Requirement statement
The require statement in Racket is just like import in Python in other imperative langauges (Python, Java, etc.)

D2. Smart helpers

Note:
You can use helper functions in a smart way when dealing with the following situations:

1. a calculation costs too much so you want to avoid doing it more than once, or
2. you want to record the value produced by the current function application.

In short, you want to create a "variable" to store some values for later use.

Implementation:

```
|%Racket_file
|
| (define (f x)
|   (h (g x) (g x)))
|
| ;; Let h be an arbitrary 2-arg function that you don't care about.
| ;; (For example, h can be + or *).
| ;; Let f, g be arbitrary 1-arg functions.
| ;; Suppose (g x) costs too much and thus
| ;; you want to avoid performing it twice.
| ;; Then this function can be rewritten as:
|
| (define (f' x)
|   (helper (g x)))
|
| (define (helper y)
|   (h y y))
|
| ;; or use local helper function:
|
| (define (f'' x)
|   (define (helper y)
|     (h y y))
|   (helepr (g x)))
|
|%END
```

Ex.
Now consider the function (drawcard n).
Suppose you want to do the following:

1. Given a list of cards called list-of-cards.
2. Draw a new card (drawcard n)
3. If the new card's secret number is in the list, discard it. Otherwise cons the new card into the list.

If you use the old approach like this:

```
|%Racket_file
|
| ;; (drawcard n) produces a card which is represented
| ;;    by a two-elemtn list.
| ;; (first (drawcard n)) returns the secret number of the card.
| ;; (second (drawcard n)) returns n
|
| ;; Suppose we have defined a helper function called unique?
| ;;   which can determine if the an identical card (a card
| ;;   with the same secret number is in the list or not.
| ;; (unique : (c : Card) -> (l : listof Card) -> Bool)
|
| ;; (main : (n : Nat) -> (lst : listof Cards) -> (l : lstof Cards))
| (define (main n lst)
|   (cond
|     [(unique? (drawcard n) lst) ;; [Line #1]
|      (cons (drawcard n) lst)]   ;; [Line #2]
|     [else lst]))
|
|%END
```

You would soon realize that the two (drawcard n) applications in line #1 and line #2 actually produces different cards and there's a very high chance these two cards have different secret numbers. Therefore your entire algorithm would be incorrect.

To fix this, you can use the above helper:

```
|%Racket_file
|
| ;; main : (n : Nat) -> (lst : listof Card) -> (l : listof Card)
| (define (main n lst)
|   ;; helper : (c : Card) -> (ls : listof Card) -> (l : listof Card)
|   (define (helper c ls)
|     [(unique? c ls) (cons c ls)]
|     [else ls])
|   (helper (drawcard n) lst))
|
|%END
```

Note:
This way, the c inside the helper function stores your value for (drawcard n) and your program is correct.

D3. Parameterized by total order

Note:
In short, we can define our own rules for ordering, and we can also pass in > and < as arguments.

Note:
When we are defining our own struct, sometimes we need to define our own rules for order. To determine which element is "greater", we can use the following function to consume a function for comparison:

Implementation:

```
|%Racket_file
|
```

```
    | (define (less-than? f)
    |   (lambda (a b)
    |     (cond
    |       [(< (f a) (f b)) true]
    |       [(< (f b) (f a)) false]
    |       [else (< a b)])))
    |
    |%END
```

Ex.

```
    |%Racket_file
    |
    | > ((less-than? abs) -1 -2)
    | #true
    |
    |%END

    |%STEPPER
    |
    | ((less-than? abs) -1 -2)
    |
    | ;; rewrite less-than?
    | ;; plug in abs as f
    | ((lambda (a b)
    |   (cond
    |     [(< (abs a) (abs b)) true]
    |     [(< (abs b) (abs a)) false]
    |     [else (< a b)]))
    |   -1 -2)
    |
    | ;; plug in -1 and -2 as a and b
    |   (cond
    |     [(< (abs -1) (abs -2)) true]
    |     [(< (abs -2) (abs -1)) false]
    |     [else (< -1 -2)])))
    |
    | #true
    |
    |%END
```

==
==

Part E. Appendix

Table of contents

Ex.

1. Rewrite the function in an unsugared form:

   [Level of difficulty: easy]

   Q1.rkt

```
    |%Racket_file
    |
    | (define (make-identity)
```

```
|   (lambda (x) x))
|
|%END
```

S1.rkt

```
|%Racket_file
|
| (define (make-identity x) x)
|
|%END
```

Ex.

2. Define "negate", which takes a boolean function f, and returns a function that produces the "not" of the output.

[Level of difficulty: easy]

```
;; Type contract:
;; (negate : (A -> Bool) -> (A -> Bool))
```

S2.rkt

```
|%Racket_file
|
| (define negate
|   (lambda (f)
|     (lambda (x)
|       (not (f x)))))
|
|%END
```

Ex.

3. Define equal-to?, which is a higher-order function. It should take a value that can be compared using equal? and return a function, that when applied to another argument, returns the equal? of arguments.

[Level of difficulty: easy]

```
;; Type contract:
;; equal-to? : (a1 : A) -> ((a2 : A) -> Bool)
```

S3.rkt

```
|%Racket_file
|
| (define equal-to?
|   (lambda (a1)
|     (lambda (a2)
|       (equal? a1 a2))))
|
|%END
```

Ex.

4. Define a function "dedup" which takes an ordered list and removes any duplicate elements. Write this function using foldr.

[Level of difficulty: medium]

```
;; Type contract:
```

;; dedup : (listof A) -> (listof A)

S4.rkt

```
|%Racket_file
|
| (define dedup
|   (lambda (lst)
|     (foldr
|       (lambda (x acc) (if (not (member x acc))
|                           (cons x acc) acc))
|       '()
|       lst)))
|
|%END
```

Ex.
5. Define "my-map", which takes a function and a list, and produces a list with f applied to all elements of the list. Use foldr.

[Level of difficulty: medium]

;; Type contract:
;; my-map : (A -> B) -> (listof A) -> (listof B)

S5.rkt

```
|%Racket_file
|
| (define my-map
|   (lambda (f lst)
|     (foldr
|       (lambda (x acc) (cons (f x) acc))
|       '()
|       lst)))
|
|%END
```

Ex.
6. Create a function "plus" that adds two natural numbers together. Use only above "f^n" and "add1", do not use recursion.

[Level of difficulty: medium]

;; Type contract:
;; plus: Nat -> Nat -> Nat

S6.rkt

```
|%Racket_file
|
| (define plus
|   (lambda (a b)
|     ((f^n add1 a) b)))
|
|%END
```

Remark:
Here is an example and the step-by-step process.

```
|%Racket_file
```

```
|
| (plus 3 15)
|
|%END

|%STEPPER
|
| (plus 3 15)
|
| ;; rewrite plus
| ((lambda (a b) ((f^n add1 a) b)) 3 15)
|
| ;; pass in a, b
| ((f^n add1 3) 15)
|
| ;; rewrite f^n
| (((lambda (f n)
|     (foldr compose identity
|            (make-list n f))) add1 3) 15)
|
| ;; pass in add1, 3
| ((foldr compose identity '(add1 add1 add1)) 15)
|
| ;; rewrite foldr
| ((lambda (x)
|   (add1 (add1 (add1 x)))) 15)
|
| ;; pass in 15
| (add1 (add1 (add1 15)))
|
| 18
|
|%END
```

Ex.

7. Create a function "mult" that multiplies two natural numbers together. Use only the above "f^n" and "plus", do not use recursion.

[Level of difficulty: hard]

```
;; Type contract:
;; mult: Nat -> Nat -> Nat
```

S7.rkt

```
|%Racket_file
|
| (define mult
|   (lambda (a b)
|     ((f^n ((curry plus) a) b) 0)))
|
|%END
```

Remark:

This is very very confusing.
I'll provide an example and step-by-step analysis.

```
|%Racket_file
|
| (mult 3 5)
|
|%END
```

```
|%STEPPER
|
| (mult 3 5)
|
| ;; rewrite mult
| ((lambda (a b) ((f^n ((curry plus) a) b) 0)) 3 5)
|
| ;; plus in 3, 5
| (((f^n (curry plus) 3) 5) 0)
|
| ;; rewrite curry
| ((f^n
|     ((lambda (f)
|       (lambda (a)
|         (lambda (b)
|           (f a b)))) plus 3)
|     5) 0)
|
| ;; plug in plus, 3
| (((f^n
|   (lamdba (b) (plus 3 b))) 5) 0)
|
| ;; in the step above, we get a partial application of 3.
| ;; now, rewrite f^n
| (((lambda (f n) compose identity (make-list f n))
|   (lambda (b) (plus 3 b)) ;; this thing is the f we are passing
|                           ;; into the outer lambda.
|   5) 0)
|
| ;; plug in 5 as n and (lambda (b) (plus 3 b))
| ;; as f in the outer lambda
| ((compose identity (make-list (lambda (b) (plus 3 b)) 5)) 0)
|
| ;; compose
| ;; now it's the beauty of partial application.
| (plus 3 (plus 3 (plus 3 (plus 3 (plus 3 0)))))
|
| 15
|
|%END
```

Remark:
This process is very challenging. Read it carefully.

E2. Practice from tutorial Nov.1 by Ashish

Note:
The following two functions are equivalent:

```
|%Racket_file
|
| (define map-add1
|   (lambda (lst)
|     (map add1 lst)))
|
| (define map-adder
|   ((curry map) add1))
|
|%END
```

Note that ((curry map) add1) creates the function (lambda (lst) (map add1 lst)).

Note:
The following two functions are equivalent:

```
|%Racket_file
|
| (define sum-list
|   (lambda (lst)
|     (foldr + 0 lst)))
|
| (define sum-list-curried
|   (((curry foldr) + 0)))
|
|%END
```

Currying and partial application too op.

E3. Assignment 6~8 Recap

Asst.6 a~e:
Practice on time complexity.
Read A1 and A6 and you should be good.

Asst.6 f:
Given AVL, define a new ADT set.

Remark:
Try to provide a wrapper struct.

Asst.7:
Create a game to draw cards and collect prizes.
Nothing too crazy but remember to check your running time.

Asst.8:
Use generate function to write (prime? n), (my-build-list n f), (my-foldl f z l), (my-insert e l <), (my-insertion-sort l <). This is an exercise to help us what fold does.

Also there is a merge sort exercise. By now we should be comfortable dealing with sorting and searching algorithms.

E4. Functions you should 100% memorize.

```
|%Racket_file
|
| (define compose
|   (lambda (f g)
|     (lambda (x)
|       (f (g x)))))
|
| (define map
|   (lambda (f lst)
|     (foldr
|       (lambda (x z) (cons (f x) z))
|       '()
|       lst)))
|
| (define filter
|   (lambda (f lst)
```

```
  |     (foldr
  |       (lambda (x z) (if (f x) (cons x z) z))
  |       '()
  |       lst)))
  |
  | (define (foldr f z l)
  |   (cond
  |     [(empty? l) z]
  |     [else (f (first l) (foldr f z (rest l)))]))
  |
  | (define (foldl f z l)
  |   (define (calc l z)
  |     (cond
  |       [(empty? l) z]
  |       [else (calc (rest l) (f (first l) z))]))
  |   (calc l z))
  |
  | (define (f'' x)
  |   (define (helper y)
  |     (h y))
  |   (helper (g x)))
  |
  |%END

-------------------------------------------------------------------------
E5. Functions that may help you on the test.

  |%Racket_file
  |
  | (define curry
  |   (lambda (f)
  |     (lambda (a)
  |       (lambda (b)
  |         (f a b)))))
  |
  | (define f^n
  |   (lambda (f n)
  |     (foldr
  |       compose
  |       identity
  |       (make-list n f))))
  |
  |%END

-------------------------------------------------------------------------
E6. Possible questions on the test.

  1. Relatively short problems
    1.1 Theory
        - One question about big-O
        - One question about ADT
        - One question about modules
    1.2 Short programs
        - One question about big-O
        - One question about ADT
        - One question about writing a lambda function
        - One question about application of a given lambda function

  2. Topic: Big-O
    2.1 Prove why f(x) is O(g(x))
    2.2 Prove why h(x) is not O(g(x))

  3. Topic: Lambda
```

```
      3.1 Write a function to ...
      3.2 Write a function to ...
      3.3 Given this function, do ...

    4. Topic: Stuff from midterm I
      4.1 Use tree to define a new ADT
      4.2 Use tree to do some operations

    5. Prove that merge sort / insertion sort is correct using induction.

---------------------------------------------------------------------------

V3.0 Complete. 2017, Nov.3 by David Duan
```