CS 135 Fall 2017
Final Exam Review
Final Exam Information

- The final exam will be held on **Wednesday, December 20th**
  9:00AM - 11:30AM in the PAC

- Check your exam seating assignment on Odyssey.
ALFs - increasing-lists

Without using explicit recursion, write a function called `increasing-lists` that consumes a positive integer \( n \) and produces a list of \( n \) lists of natural numbers, where the \( i^{th} \) list contains the first \( i + 1 \) natural numbers.

\[
\text{(increasing-lists 1) } \Rightarrow \text{'((0))}
\]

\[
\text{(increasing-lists 4) } \Rightarrow \text{'((0)}
\]

\[
'(0)
\]

\[
(0 1)
\]

\[
(0 1 2)
\]

\[
(0 1 2 3))
\]
ALFs - increasing-lists

(define (increasing-lists n)
  (build-list n (lambda (x) (build-list (add1 x) identity)))))
ALFs - digits->num

Without using explicit recursion, write a function called digits->num that consumes a non-empty list of digits and produces the number those digits form. Include a contract.

Try to do it without reversing the list.

(digits->num '(5)) ⇒ 5

(digits->num '(1 7 4 9 4)) ⇒ 17494
ALFs - digits- > num

;; digits- > num: (listof Nat) → Nat
;; requires: lst is non-empty
;; 0 <= each number in lst <= 9

(define (digits- > num lst)
    (foldl (lambda (frst acc) (+ frst (* 10 acc))) 0 lst))
ALFs - count

Without using explicit recursion, write a function called count that consumes 2 arguments, of which the second one is a list. It produces the number of times the first argument occurs in the list.

(count 'cat '(3 "cat" 11)) ⇒ 0

(count 'cat '("cat" cat 4 cat 6)) ⇒ 2
ALFs - count

(define (count item lst)
  (length (filter (lambda (x) (equal? x item)) lst)))
Without using explicit recursion, write a function called `count-chars` that consumes a `Char` and a `(listof Any)`. It should count the number of occurrences of the character in all the strings in the list.

```
(count-chars #\a '(a "Aardvark" #\a ("abba" 12) "hagrid!?")) ⇒ 3
```
ALFs - count-chars

(define (count-chars char lst)
  (local
    [(define all-strings (map string->list (filter string? lst)))
     (define char-counts (map (lambda (loc) (count char loc)) all-strings))]
    (foldr + 0 char-counts)))
ALFs - count-chars/nested

Now, write a function called `count-chars/nested` that consumes a `Char` and a `Nest-List-Any`. It should count the number of occurrences of the character in all the strings in the nested list. You can use explicit recursion.

\[
\text{(count-chars/nested } \#\backslash a \ 'a \ "Aardvark" \ (\#\backslash a \ "hag" \ "ah") \ "at" \ ("doggo"))) \Rightarrow 5
\]

Recall the data definition for a nested list:

;; A `Nest-List-Any` is one of:

;; * empty

;; * (cons `Any` `Nest-List-Any`)

;; * (cons `Nest-List-Any` `Nest-List-Any`)

CS 135 Fall 2017 Final Exam Review 7
ALFs - count-chars/nested

```
(define (count-chars/nested char nlst)
  (cond
    [(empty? nlst) 0]
    [(string? (first nlst)) (+ (count char (string->list (first nlst)))
                               (count-chars/nested char (rest nlst)))]
    [(list? (first nlst)) (+ (count-chars/nested char (first nlst))
                             (count-chars/nested char (rest nlst)))]
    [else (count-chars/nested char (rest nlst))]))
```
Graphs - Adjacency List Representation

We can represent a graph as a list of pairs. Each pair consists of a symbol (the node’s name) and a list of symbols (the names of the node’s out-neighbours).

This is called the **adjacency list** representation.

```scheme
;; A Node is a Sym

;; A Graph is a (listof (list Node (listof Node)))
```
Graphs - Adjacency List Representation

Define a constant \( g1 \) that represents the graph below as an adjacency list.

```
(define g1 '(
  (A (B C F))
  (B ())
  (C (A B))
  (D (A C F))
  (E (A B C))
  (F (A))
))
```
Graphs - Adjacency List Representation

Define a constant g1 that represents the graph below as an adjacency list.

\[
\text{(define g1}
\text{ '((A (B C F))}
\text{ (B ()))}
\text{ (C (A B)))}
\text{ (D (A C F)))}
\text{ (E (A B C)))}
\text{ (F (A)))))}
\]
Graphs - neighbours

Without using explicit recursion, write the `neighbours` function for an explicit graph. Recall that `neighbours` consumes a Node \( v \) and a Graph \( g \) and produces a list of out-neighbours of \( v \) in \( g \). If \( v \) does not exist in \( g \), produce false.

\[
\begin{align*}
(\text{neighbours } 'D g1) & \Rightarrow '(A C F) \\
(\text{neighbours } 'B g1) & \Rightarrow '()
\end{align*}
\]

(\text{neighbours } 'H g1) \Rightarrow \text{false}
Graphs - neighbours

(define (neighbours v g)
  (local
    [(define found (filter (lambda (x) (symbol=? (first x) v)) g))]
    (cond
      [(empty? found) false]
      [else (second (first found))]))))
Graphs - delete-node

Write a function `delete-node` that consumes a Node \( v \) and a Graph \( g \) and produces the graph with \( v \) removed. If \( v \) does not exist in \( g \), the same graph is produced.

\[
\text{(define g1 } \quad \text{ (define g1 } \\
\text{'((A (B C F)) } \quad \text{'((A (B F))} \\
\text{(B () } \quad \text{(B ()} \\
\text{(C (A B)) } \quad \text{(D (A F))} \\
\text{(D (A C F)) } \quad \text{(E (A B))} \\
\text{(E (A B C)) } \quad \text{(F (A))} \\
\text{(F (A))) })}
\]

\[
\text{(delete-node } 'C \text{ g1)} \Rightarrow \text{(delete-node } 'C \text{ g1)} \quad \Rightarrow
\]

\[
\text{'((A (B F)) } \quad \text{'((A (B F))} \\
\text{(B () } \quad \text{(B ()} \\
\text{(D (A F)) } \quad \text{(E (A B))} \\
\text{(E (A B)) } \quad \text{(F (A))})
\]
(define (delete-node v g)
  (local
    [(define remove-node
        (filter (lambda (x) (not (symbol=? (first x) v))) g))
     (define remove-edges
        (map (lambda (x)
               (list (first x)
                  (filter (lambda (y)
                            (not (symbol=? y v))) (second x)))
               remove-node))]
    remove-edges))
Stepping Lambda

Recall that the rule for evaluating anonymous lambda expression is

\[ ((\text{lambda} \ (x_1 \ldots \ x_n) \ \text{exp}) \ v_1 \ldots \ v_n) \Rightarrow \text{exp}' \]

where \text{exp}' is \text{exp} with all occurrences of \( x_1 \) replaced by \( v_1 \), all occurrences of \( x_2 \) replaced by \( v_2 \), and so on.
Stepping Lambda: Example

Given that the following definition has been processed in the Intermediate Student with Lambda language:

```
(define fun
  (lambda (x y)
    (lambda (y z)
      (lambda (z) (+ x y z))))
```

Produce a step-by-step evaluation of the following program:

```
(((fun 1 2) 3 4) 5)
```
Stepping Lambda: Example

⇒ (((((\(x\) \(y\))

(\(y\) \(z\))

(\(z\) (+ \(x\) \(y\) \(z\)))))) 1 2) 3 4) 5)
Stepping Lambda: Example

⇒ (((((lambda (x y)
    (lambda (y z)
        (lambda (z) (+ x y z)))) 1 2) 3 4) 5)
⇒ (((lambda (y z)
    (lambda (z) (+ 1 y z))) 3 4) 5)
Stepping Lambda: Example

⇒ ((((lambda (x y)  
    (lambda (y z)  
      (lambda (z) (+ x y z)))) 1 2) 3 4) 5) 
⇒ ((((lambda (y z)  
    (lambda (z) (+ 1 y z))) 3 4) 5) 
⇒ ((lambda (z) (+ 1 3 z)) 5)
Stepping Lambda: Example

⇒ (((((\lambda (x y) \\
    (\lambda (y z) \\
    (\lambda (z) (+ x y z)))) 1 2) 3 4) 5) \\
⇒ (((\lambda (y z) \\
    (\lambda (z) (+ 1 y z))) 3 4) 5) \\
⇒ ((\lambda (z) (+ 1 3 z)) 5) \\
⇒ (+ 1 3 5)
Stepping Lambda: Example

⇒ (((((lambda (x y)

  (lambda (y z)

    (lambda (z) (+ x y z)))))(lambda (y z)

    (lambda (z) (+ 1 y z)))))(lambda (z) (+ 1 z)))(lambda (z) (+ 1 y z))) (lambda (z) (+ 1 3 z)) 5)

⇒ (+ 1 3 5)

⇒ 9
BTs

;; An Integer Binary Tree (IntBT) is one of:
;; * empty
;; * an IntNode

(define-struct int-node (key left right))

;; An IntNode is a (make-int-node Int IntBT IntBT)
**BTs - min-key**

Write a function, `min-key` that consumes a non-empty IntBT and produces the minimum key in that tree.

```
(define b1
  (make-int-node 3
    (make-int-node 0
      empty
      (make-int-node 1 empty empty))
    (make-int-node 6
      (make-int-node 8 empty empty)
      (make-int-node 10
        (make-int-node 2 empty empty)
        empty)))))

(min-key b1) ⇒ 0
```
BTs - min-key

(define (min-key bt)
  (cond [(and (empty? (int-node-left bt))
              (empty? (int-node-right bt)))
         (int-node-key bt)]
    [(empty? (int-node-left bt))
     (min (int-node-key bt)
          (min-key (int-node-right bt)))]
    [(empty? (int-node-right bt))
     (min (int-node-key bt)
          (min-key (int-node-left bt)))]
    [else (min (int-node-key bt)
               (min-key (int-node-left bt))
               (min-key (int-node-right bt)))]))
BSTs

;; A Binary Search Tree (BST) is one of:

;; * empty

;; * a Node

(define-struct node (key left right))

;; A Node is a (make-node Int BST BST)

;; requires:

;;   key > every key in the left BST

;;   key < every key in the right BST
BSTs - min-key/bst

Write a function, min-key/bst that consumes a non-empty BST and produces the minimum key in that tree. Do you need to traverse the entire tree again?

\[
\text{(define b2}
\text{ \\
\text{ (make-node 3}
\text{ \\
\text{ (make-node 1 empty}
\text{ empty (make-node 2 empty empty)))}
\text{ \\
\text{ (make-node 6}
\text{ (make-node 4 empty empty) (make-node 10}
\text{ (make-node 8 empty empty) empty))})}
\text{ (min-key/bst b2) } \Rightarrow 1
\]
BSTs - min-key/bst

(define (min-key/bst bst)
  (cond
    [(empty? (node-left bst)) (node-key bst)]
    [else (min-key/bst (node-left bst))])))
BSTs - keys-in-range

Write a function, `keys-in-range` that consumes 2 numbers, `hi` and `lo`, and a BST. It produces a list of all the keys in the BST that are between `hi` and `lo` inclusive. You should not try to convert the entire tree into a list first.

\[
\text{(keys-in-range 2 7 empty)} \Rightarrow \text{empty}
\]

\[
\text{(keys-in-range 2 7 b1)} \Rightarrow '(2 3 4 6)
\]

\[
\text{(keys-in-range } -5 -2 \text{ b1)} \Rightarrow \text{empty}
\]
BSTs - keys-in-range

;; keys-in-range: Num Num BST → (listof Num)
;; requires: lo <= hi
(define (keys-in-range lo hi bst)
  (cond
    [(empty? bst) empty]
    [(< (node-key bst) lo)
      (keys-in-range lo hi (node-right bst))]
    [(and (>= (node-key bst) lo) (<= (node-key bst) hi))
      (append (keys-in-range lo hi (node-left bst))
              (cons (node-key bst)
                    (keys-in-range lo hi (node-right bst))))]
    [(> (node-key bst) hi)
      (keys-in-range lo hi (node-left bst))])))
Stepping Local

An expression of the form

\[
\text{(local [(define x1 exp1) \ldots (define xn expn)] bodyexp)}
\]

is handled as follows.

\(x1\) is replaced with a fresh identifier (call it \(x1\_new\)) everywhere it’s used in the \text{local} expression.

The same thing is done with \(x2\) through \(xn\).

The definitions \((\text{define} \ x1\_new \ \text{exp1}) \ldots (\text{define} \ xn\_new \ \text{expn})\) are then lifted out (all at once) to the top level of the program, preserving their order.
Stepping Local

When all the rewritten definitions have been lifted out, what remains looks like \((\text{local } []) \, \text{bodyexp}'\), where \text{bodyexp}' is the rewritten version of \text{bodyexp}. This is just replaced with \text{bodyexp}'. All of this is a single step.

\[
(\text{local } [(\text{define } x_1 \, \text{exp}_1) \ldots (\text{define } x_n \, \text{exp}_n)] \, \text{bodyexp})
\Rightarrow (\text{define } x_1 \_0 \, \text{exp}_1')
\]

\[
\ldots
\]

\[
(\text{define } x_n \_0 \, \text{exp}_n')
\]

\text{bodyexp}'
Stepping Local: Example 1

When renaming local definitions append “.0” if possible, or else “.1”, “.2”, etc. **Do not** recopy any line that is already in simplest form.

```scheme
(define x 5)
(define y 10)
(define (my-fn a b)
  (local [(define x (+ a b))
    (define y (+ x (local [(define x 20)] x)))]
  (list x y))

(my-fn x y)
```
Stepping Local: Example 1

(define x 5)
(define y 10)
(define (my-fn a b)
  (local [(define x (+ a b))
    (define y (+ x (local [(define x 20)] x)))]
  (list x y)))

(my-fn x y) ⇒ (my-fn 5 y)
Stepping Local: Example 1

(define x 5)
(define y 10)
(define (my-fn a b)
  (local [(define x (+ a b))
           (define y (+ x (local [(define x 20)] x)))]
          (list x y)))

(my-fn x y) ⇒ (my-fn 5 y) ⇒ (my-fn 5 10)
Stepping Local: Example 1

⇒ (local [(define x (+ 5 10))
   (define y (+ x (local [(define x 20)] x)))]
   (list x y))
Stepping Local: Example 1

⇒ (local [(define x (+ 5 10))
  (define y (+ x (local [(define x 20)] x)))]
  (list x y))

⇒ (define x_0 (+ 5 10))
  (define y_0 (+ x_0 (local [(define x 20)] x)))
  (list x_0 y_0)
Stepping Local: Example 1

⇒ (local [(define x (+ 5 10))
            (define y (+ x (local [(define x 20)] x)))]
    (list x y))

⇒ (define x_0 (+ 5 10))
    (define y_0 (+ x_0 (local [(define x 20)] x)))
    (list x_0 y_0)

⇒ (define x_0 15)
    (define y_0 (+ x_0 (local [(define x 20)] x)))
    (list x_0 y_0)
Stepping Local: Example 1

⇒ (define y_0 (+ 15 (local [(define x 20)] x)))
   (list x_0 y_0)
Stepping Local: Example 1

⇒ (define y_0 (+ 15 (local [(define x 20)] x)))
   (list x_0 y_0)
⇒ (define x_1 20)
   (define y_0 (+ 15 x_1))
   (list x_0 y_0)
Stepping Local: Example 1

⇒ (define y_0 (+ 15 (local [(define x 20)] x)))
   (list x_0 y_0)
⇒ (define x_1 20)
   (define y_0 (+ 15 x_1))
   (list x_0 y_0)
⇒ (define y_0 (+ 15 20))
   (list x_0 y_0)
Stepping Local: Example 1

⇒ (define y₀ (+ 15 (local [(define x 20)] x)))
    (list x₀ y₀)
⇒ (define x₁ 20)
    (define y₀ (+ 15 x₁))
    (list x₀ y₀)
⇒ (define y₀ (+ 15 20))
    (list x₀ y₀)
⇒ (define y₀ 35)
    (list x₀ y₀)
Stepping Local: Example 1

⇒ (define y_0 (+ 15 (local [(define x 20)] x)))
   (list x_0 y_0)
⇒ (define x_1 20)
   (define y_0 (+ 15 x_1))
   (list x_0 y_0)
⇒ (define y_0 (+ 15 20))
   (list x_0 y_0)
⇒ (define y_0 35)
   (list x_0 y_0)
⇒ (list 15 y_0)
Stepping Local: Example 1

⇒ (define y_0 (+ 15 (local [(define x 20)] x)))
  (list x_0 y_0)
⇒ (define x_1 20)
  (define y_0 (+ 15 x_1))
  (list x_0 y_0)
⇒ (define y_0 (+ 15 20))
  (list x_0 y_0)
⇒ (define y_0 35)
  (list x_0 y_0)
⇒ (list 15 y_0) ⇒ (list 15 35)
Stepping Local: Example 2

When renaming local definitions append “_0” if possible, or else “_1”, “_2”, etc. Do not recopy any line that is already in simplest form.

```
(define x 5)
(define y 10)
(define (my-fn x y) ; parameter names are different
  (local [(define x y) ; this is different as well
    (define y (+ x (local [(define x 20)] x)))]
    (list x y)))
```

(my-fn x y)
(define x 5)
(define y 10)
(define (my-fn x y)
  (local [(define x y)
           (define y (+ x (local [(define x 20)] x)))]
            (list x y)))

(my-fn x y) ⇒ (my-fn 5 y)
Stepping Local: Example 2

(define x 5)
(define y 10)
(define (my-fn x y)
  (local [(define x y)
    (define y (+ x (local [(define x 20)] x)))]
    (list x y)))]

(my-fn x y) ⇒ (my-fn 5 y) ⇒ (my-fn 5 10)
Stepping Local: Example 2

⇒ (local [(define x y)

    (define y (+ x (local [(define x 20)] x)))]

    (list x y))
Stepping Local: Example 2

⇒ (local [(define x y)
            (define y (+ x (local [(define x 20)] x))]])
   (list x y))

⇒ (define x_0 y_0)
   (define y_0 (+ x_0 (local [(define x 20)] x)))
   (list x_0 y_0)
Stepping Local: Example 2

⇒ (local [(define x y)
    (define y (+ x (local [(define x 20)] x)))]
    (list x y))
⇒ (define x_0 y_0)
    (define y_0 (+ x_0 (local [(define x 20)] x)))
    (list x_0 y_0)
⇒ Error: local variable used before its definition: y
Higher order functions - straight-line

Write a function that consumes 2 Posns and produces a function that defines the equation of the straight line between them. You may assume that the x-coordinates of the 2 points are distinct from each other.

\[
\text{(define f1 (straight-line (make-posn 0 0) (make-posn 1 1)))}
\]
\[
(f1 4) \Rightarrow 4
\]

\[
\text{(define f2 (straight-line (make-posn 8 3) (make-posn 3 3)))}
\]
\[
(f2 2) \Rightarrow 3
\]

\[
\text{(define f3 (straight-line (make-posn 0 0) (make-posn 1 4)))}
\]
\[
(f3 3) \Rightarrow -12
\]
Higher order functions - straight-line

;; straight-line: Posn Posn → (Num → Num)
;; requires: x-coordinates of p1 and p2 are not equal to each other

(define (straight-line p1 p2)
  (local
    [(define slope (/ (− (posn-y p1) (posn-y p2))
          (− (posn-x p1) (posn-x p2))))
     (define intercept (− (posn-y p1) (* slope (posn-x p1))))
     (define (y-at x) (+ (* slope x) intercept))]
  y-at))
Which of the following was one of Grace Hopper’s achievements?

A) Proof that the Halting Problem is undecidable
B) Design for the "Analytical Engine"
C) Creation of the first compiler
D) Axiomatization of geometry
E) Development of FORTRAN
Which of the following was one of Grace Hopper’s achievements?

A) Proof that the Halting Problem is undecidable
B) Design for the ”Analytical Engine”
C) Creation of the first compiler
D) Axiomatization of geometry
E) Development of FORTRAN
Mutual recursion - Tournament

For this example, we are going to use trees to model the results of a tournament.

```
(define-struct tournament (winner round))
;; A Tournament is a (make-tournament Str Round)
;; requires: winner is the name of the player with the best time
;; in the topmost round

;; A Round is a (listof Player)
```

```
(define-struct player (name time last-round))
;; A Player is a (make-player Str Num Round)
;; requires: time > 0
;; name is the player with the best time in the topmost round
;; round of last-round if it is non-empty
```
(define cs135-cup
  (make-tournament "Paul"
    (list (make-player "Dave" 12.3 (list (make-player "Troy" 13.1 empty)
                                      (make-player "Dave" 11 empty)))
          (make-player "Ian" 16.9 (list (make-player "Ian" 15.1 empty)
                                        (make-player "Craig" 15.8 empty)
                                        (make-player "Byron" 18.3 empty)))
          (make-player "Paul" 12.1 (list (make-player "Paul" 19.9 empty)
                                        (make-player "Dustin" 9999 empty))))))
Mutual recursion - Templates

Write a template function for each of Tournament, Round and Player.

```
(define-struct tournament (winner round))
;; A Tournament is a (make-tournament Str Round)
;; requires: winner is the name of the player with the best time
;; in the topmost round

;; A Round is a (listof Player)

(define-struct player (name time last-round))
;; A Player is a (make-player Str Num Round)
;; requires: time > 0
;; name is the player with the best time in the topmost
;; round of last-round if it is non-empty
```
Mutual recursion - Templates

;; tournament-template: Tournament → Any
(define (tournament-template tourn)
  (... (tournament-winner tourn) ...)
  (round-template (tournament-round tourn)) ...))
Mutual recursion - Templates

;; round-template: Round → Any
(define (round-template round)
  (cond
    [(empty? round) . . . ]
    [else (. . . (player-template (first round)). . .
            (round-template (rest round)). . . )]))

;; player-template: Player → Any
(define (player-template pl)
  ( . . . (player-name pl) . . .
    (player-time pl) . . .
    (round-template (player-last-round pl)) . . .)))
Mutual recursion - new-record

Here’s a structure we will use to store information about a record:

```scheme
(define-struct record (name time))
;; A Record is a (make-record Str Num)
;; requires: time > 0
```

A player makes a new record if they have a faster time than the current record. Using your templates, write a function `new-record` that consumes a current record and a Tournament. It should produce a Record structure with the name and time of the player in the tournament who has the new record. If no one in the tournament had a faster time than the current record, produce `false`. You may assume that all times in the tournament are unique.

```scheme
(new-record (make-record "Byron" 11.2) cs135-cup) ⇒ (make-record "Dave" 11)
(new-record (make-record "Karen" 3.1) cs135-cup) ⇒ false
```
Mutual recursion - new-record

;; new-record: Record Tournament → (anyof Record false)

(define (new-record rec tourn)
  (local
    [(define tourn-record
        (new-record/round (tournament-round tourn) rec))]
    (cond
      [(< (record-time tourn-record) (record-time rec)) tourn-record]
      [else false])))
Mutual recursion - new-record

;;; new-record/round: Round Record → Record

(define (new-record/round round record-so-far)
  (cond
    [(empty? round) record-so-far]
    [else (local
            [(define round-record
               (new-record/player (first round) record-so-far))]
            (cond
              [(< (record-time round-record) (record-time record-so-far))
               (new-record/round (rest round) round-record)]
              [else (new-record/round (rest round) record-so-far)]))))
Mutual recursion - new-record

;;; new-record/player: Player Record → Record

(define (new-record/player pl record-so-far)
  (new-record/round (player-last-round pl)
    (cond
      [(< (player-time pl) (record-time record-so-far))
       (make-record (player-name pl) (player-time pl))]
      [else record-so-far])))))
Who developed the Lisp family of programming languages?

A) John von Neumann
B) Alonzo Church
C) Kurt Gödel
D) Grace Hopper
E) John McCarthy
Who developed the Lisp family of programming languages?

A) John von Neumann
B) Alonzo Church
C) Kurt Gödel
D) Grace Hopper
E) John McCarthy
Accumulative and Generative Recursion

Accumulative Recursion

- All parameters in the recursive call are either unchanged or one step closer to the base case (as with pure structural recursion), plus one or more parameters that accumulate partial answers.

- Accumulators are usually produced in the base case(s) directly, or manipulated before being produced.

- In the recursive call, it does not matter how complicated the code is to update an accumulator, as long as it adds on to the partial answer.
Generative Recursion — A parameter is considered generative if:

- It is changed in the recursive call without a clear base case
- It doesn’t get one step closer to the base case in the recursive call, according to the data definition
- It is not an accumulator, and thus isn’t produced in the base case

Examples of generative recursive calls:

- `(collatz (+ 1 (* 3 n)))`
- `(foo (not bool))`
- `(foo (rest (rest (rest lst))))`
- `(foo (remove elem lst))`

If in any recursive call of a function, there is at least one parameter that satisfies the conditions above, regardless of how the other parameters are changing, the function uses generative recursion.
A Matrix is a (listof (listof Num))

requires: the length of each sub-list is the same as the length of the Matrix, and must be at least 1

(minor r c mat) deletes the r-th row and c-th column from mat

(minor: Nat Nat Matrix → Matrix)

(define (minor r c mat) . . . ) ; we won’t worry about the definition of minor

(determinant-from: Matrix Nat → Num)

(define (determinant-from mat cur-row)

  (local [((define dim (length mat))]]

    (cond [ (> = cur-row dim) 0]

      [(= 1 dim) (first (first mat))]

      [else (+ (∗ (first (list-ref mat cur-row)) (expt − 1 cur-row)

                (determinant-from (minor cur-row 0 mat) 0))

                (determinant-from mat (add1 cur-row)))]))

What kind of recursion is this?
A) Pure structural      B) Generative      C) Accumulative      D) I don’t know
A Matrix is a (listof (listof Num))
requires: the length of each sub-list is the same as the length of the Matrix, and must be at least 1

(minor r c mat) deletes the r-th row and c-th column from mat

(minor: Nat Nat Matrix → Matrix)

(define (minor r c mat) ...); we won’t worry about the definition of minor

(determinant-from: Matrix Nat → Num)

(define (determinant-from mat cur-row)
  (local [(define dim (length mat))]
    (cond [(>= cur-row dim) 0]
      [(= 1 dim) (first (first mat))]
      [else (+ (∗ (first (list-ref mat cur-row)) (expt −1 cur-row)
               (determinant-from (minor cur-row 0 mat) 0))
                 (determinant-from mat (add1 cur-row)))]))

What kind of recursion is this?
A) Pure structural       B) Generative       C) Accumulative       D) I don’t know
;; A MysteryType is one of:
;; * empty
;; * (cons Num (cons Str MysteryType))

;; mystery-fn1: MysteryType → (listof (list Str Num))

(define (mystery-fn1 mystery-param)
  (cond
   [(empty? mystery-param) empty]
   [else (cons (list (second mystery-param)
                    (first mystery-param))
            (mystery-fn1 (rest (rest mystery-param))))]))

What kind of recursion is this?
A) Pure structural       B) Generative       C) Accumulative       D) I don’t know
;;; A MysteryType is one of:
;;; * empty
;;; * (cons Num (cons Str MysteryType))

;;; mystery-fn1: MysteryType → (listof (list Str Num))

(define (mystery-fn1 mystery-param)
  (cond
   [(empty? mystery-param) empty]
   [else (cons (list (second mystery-param)
                   (first mystery-param))
            (mystery-fn1 (rest (rest mystery-param))))]))

What kind of recursion is this?
A) Pure structural    B) Generative    C) Accumulative    D) I don’t know
(define (mystery-fn2 param1 param2)
  (cond
   [(empty? param1) (+ (expt param2 3) 100)]
   [(number? (first param1))
    (mystery-fn2 (rest param1) (/ (abs param2) 5))]
   [(string? (first param1))
    (mystery-fn2 (rest param1) (max param2 30))]
   [else (mystery-fn2 (rest param1)
      (* (ceiling (add1 param2))
      (floor (sub1 param2)))]))

What kind of recursion is this?
A) Pure structural B) Generative C) Accumulative D) I don’t know
(define (mystery-fn2 param1 param2)
  (cond
    [(empty? param1) (+ (expt param2 3) 100)]
    [(number? (first param1))
     (mystery-fn2 (rest param1) (/ (abs param2) 5))]
    [(string? (first param1))
     (mystery-fn2 (rest param1) (max param2 30))]
    [else (mystery-fn2 (rest param1)
                        (* (ceiling (+ 1 param2))
                           (floor (- 1 param2))))]))

What kind of recursion is this?
A) Pure structural       B) Generative       C) Accumulative       D) I don’t know
Generative recursion - count-uniq

Write a function `count-uniq` that consumes a list and produces the number of unique elements in it. Do not use an accumulator.

`(count-uniq '(a a 1 b 1 2)) ⇒ 4`
`(count-uniq '(a b c)) ⇒ 3`
`(count-uniq '()) ⇒ 0`
Generative recursion - count-uniq

;;; remove-all-first: (listof Any) → (listof Any)
;;; requires: lst is non-empty

(define (remove-all-first lst)
  (filter (lambda (x) (not (equal? x (first lst)))) (rest lst)))

;;; count-uniq: (listof Any) → Nat

(define (count-uniq lst)
  (cond
    [(empty? lst) 0]
    [else (add1 (count-uniq (remove-all-first lst))))])
ALFs - adjfoldr

We will now write our own abstract list function, adjfoldr. adjfoldr abstracts recursion on a list, like foldr, but the combine function takes 3 arguments:

;; adjfoldr: (X X Y -> Y) Y (listof X) -> Y

adjfoldr combines the first element of the list, the second element of the list and the recursive call on the rest of the list.

This might be useful when you need to process adjacent elements together instead of just one element at a time.
ALFs - adjfoldr

Here are some examples:

Create a list of the pairwise differences between adjacent elements in (list 1 3 4 9):

(adjfoldr (lambda (first second rror)
            (cons (abs (− first second)) rror)) empty (list 1 3 4 9))

⇒ (list 2 1 5)

Check if (list 1 2 5 6 7) is sorted in increasing order:

(adjfoldr (lambda (first second rror)
            (and (<= first second) rror)) true (list 1 2 5 6 7))

⇒ true
ALFs - adjfoldr

;; adjfoldr: (X X Y → Y) Y (listof X) → Y

(define (adjfoldr combine base lst)
  (cond
   [(empty? lst) base]
   [(empty? (rest lst)) base]
   [else (combine (first lst) (second lst) (adjfoldr combine base (rest lst)))]))
ALFs - arith-seq?

Use adjfoldr to write a function arith-seq? that consumes a non-empty list of numbers and determines whether or not the list forms an arithmetic sequence.

In an arithmetic sequence, the difference between adjacent elements is constant.

\[(\text{arith-seq? } '(3)) \Rightarrow \text{true}\]

\[(\text{arith-seq? } '(-3 0 3 6 9)) \Rightarrow \text{true}\]

\[(\text{arith-seq? } '(9 7 3 1)) \Rightarrow \text{false}\]
ALFs - arith-seq?

(define (arith-seq? lon)
  (cond [(empty? (rest lon)) true]
        [else (local [(define diff (− (first lon) (second lon)))]
                    (adjfoldr (lambda (frst sec rror)
                                (and (= diff (− frst sec)) rror))
                               true (rest lon)))]))