Assignment:	09
Due:	**Friday**, April 4, 2025 9:00 pm
Coverage:	L19
Language level:	Intermediate Student with Lambda
Allowed recursion:	See assignment policies
Files to submit:	`partition.rkt`, `super.rkt` `alf.rkt` `engine.rkt`

Assignment policies:

- Make sure you read the official assignment post on **ed**.
- Unless otherwise indicated, you may use abstract list functions and/or lambda to answer any question.
- Unless otherwise indicated, you may define helper functions as needed.
- If a question requires the use of abstract list functions, you may not use any kind of recursion in your solution. Otherwise, you will not receive **any** marks for the question.
- If a question requires the use of lambda, you must use lambda as indicted. Otherwise, you will not receive **any** marks for the question.
- If a question places restrictions on the use of particular constructs, you must follow those restrictions. Otherwise, you will not receive **any** marks for the question.
- Functions and symbols must be named exactly as they are written in the assignment questions.
- You test cases must provide full coverage, i.e., no highlighting, but **you do not need to provide a purpose or contract for any function**.

Here are the assignment questions you need to solve and submit.

1. **[12%]**: Write a function `partition` that consumes a predicate and a list. It produces a two element list, `(list X Y)`, where `X` is a list of those items in the consumed list that satisfy the predicate, and `Y` is a list of those items that don't satisfy the predicate. The order of items in each list must be the same as the original list.

```
(check-expect
 (partition symbol? '(6 apple 12 egg bread))
 '((apple egg bread) (6 12)))
(check-expect
 (partition odd? '(1 2 3 4 5 6 7 8 9))
 '((1 3 5 7 9) (2 4 6 8)))
(check-expect (partition empty? '(() (()) ())) '((()()) ((()))))
```

Place your solution in `partition.rkt`.

2. **[28%]**: Place your solutions to the following questions in `super.rkt`. You may use **`lambda`** expressions, but you may not **`define`** helper functions, even inside a **`local`**. Your solution to each of the parts of this question should have no more than one **`define`**.

   (a) **[10%]**: Lecture L17 introduced the function `(filter pred? lst)` that produces a list with the elements of `lst` for which the predicate `pred?` produces a true value. Write a function `super-filter` that generalizes `filter` to work on an arbitrarily nested list applying the predicate to filter non-list elements in each nested list.

```
(check-expect
 (super-filter
  odd?
  (list 1 (list 2 (list 2 3 4) 5 6 (list 7 8 9)) 10 11 12))
 (list 1 (list (list 3) 5 (list 7 9)) 11))
```

   (b) **[6%]** Using a predicate with `super-filter`, define a function `(ruthless lst)` that removes the symbol `'ruth` from a nested list of symbols.

```
(check-expect
 (ruthless
  (list 'rabbit
        (list 'apple 'pluto
              (list 'ruth 'blue) 'ruth) 'hello))
 (list 'rabbit
       (list 'apple 'pluto
             (list 'blue)) 'hello))
```

   (c) **[6%]** Using a predicate with `super-filter`, define a function `(supersize n lst)` that removes all numbers less than `n` from a nested list of natural numbers.

```
(check-expect
 (supersize 4 (list 8 1 (list 2 6 3) 10 1))
 (list 8 (list 6) 10))
```

   (d) **[6%]** Using a predicate with `super-filter`, define `(super-keeper pred? lst)`, a function that produces a list with the elements of `lst` for which the predicate `pred?` produces a `false` value.

```
(check-expect
 (super-keeper
  odd?
  (list 1 (list 2 (list 2 3 4) 5 6 (list 7 8 9)) 10 11 12))
 (list (list 2 (list 2 4) 6 (list 8)) 10 12))
```

3. **[36%]**: This question depends on material in lecture L19. Implement the following functions with abstract list functions. You may not use recursion. You may use **`lambda`** expressions, but you may not **`define`** helper functions, even inside a **`local`**. Your solution to each of the parts of this question should have no more than one **`define`**.

   (a) **[6%]**: `occurrences` consumes a list of numbers and a number, in that order, and produces the number of times that the given number occurs in the list of numbers.

   (b) **[6%]**: `absolutely-odd` consumes a list of integers and produces the sum of the absolute values of the odd integers in the list.

   (c) **[6%]**: `zip` consumes two lists of equal length, and produces a list of pairs (two element lists) where the *i*th pair contains the *i*th element of the first list followed by the *i*th element of the second list.

   (d) **[6%]**: `unzip` consumes a list of pairs, and produces a list of two lists. The first list contains the first element from each pair, and the second list contains the second element from each pair, in the original order. Unzipping an empty list produces `'(() ())`.

   (e) **[6%]**: `dedup` ("de-duplicate") consumes a list of numbers and produces a new list with only the first occurrence of each element of the original list.

   (f) **[6%]**: `(subsequence lst from to)` consumes a list and two natural numbers. It produces the subsequence from `lst` that begins at index `from` and ends just before index `to`. Indexing starts at 0.

```
(check-expect (occurrences '(1 2 1 2 2 3 1) 2) 3)
(check-expect (occurrences '() 2) 0)
(check-expect (occurrences '(1 2 1 2 2 3 1) 4) 0)
(check-expect (absolutely-odd '(1 -5 4 6 5)) 11)
(check-expect (zip '(1 2 3) '(a b c)) '((1 a)(2 b)(3 c)))
(check-expect (unzip '((1 a)(2 b)(3 c))) '((1 2 3) (a b c)))
(check-expect (unzip '()) '(()()))
(check-expect (dedup '(1 2 1 3 3 2 4)) '(1 2 3 4))
(check-expect (subsequence '(a b c d e f g) 1 4) '(b c d))
(check-expect (subsequence '(a b c d e f g) 1 1) '())
(check-expect (subsequence '(a b c d) 0 400) '(a b c d))
(check-expect (subsequence '(a b c d e f g) 0 7) '(a b c d e f g))
(check-expect (subsequence '(a b c d) 0 400) '(a b c d))
```

Place your solutions in `alf.rkt`.

If you are having trouble answering a question, you might start by solving it using recursion, in the way you might have solved it for an earlier assignment. Once it works, try to evolve it towards a solution that uses **`lambda`** instead of helper functions and abstract list functions instead of recursion.

4. **[24%]** This question continues from related questions on assignments #5 and #6. In this question, you will write a game engine that allows three players to play a game of Dou Dizhu. For this question, we will supply the players for testing purposes.

   We use the definitions of `Card` and `Hand` from assignments #5 and #6. Recall from those assignments that a `Hand` is a sorted list of `Card`, where a Card is one of `3`, `4`, `5`, `6`, `7`, `8`, `9`, `10`, `'Jack`, `'Queen`, `'King`, `'Ace`, `2`, `'Black`, and `'Red`. Playable hands are limited to those given in assignment #6, specifically rockets, bombs, solos, pairs, trios, straights, straight pairs, and airplanes, plus `empty`, which will be used to indicate a pass. If two players in a row pass, the third player may not pass, but can play any of the cards they are holding that form a playable hand.

   At the start of a Dou Dizhu game, each of the three players bid for the role of "landlord". In this assignment, we assume that has already taken place. The landlord will play first and will have 20 cards, while the other players will have 17 cards. These hands together will form a standard deck of 54 cards. In the actual game, bids can be between 1 and 3, and can double when certain hands are played. In this assignment, we will ignore all aspects of the game related to bidding and scoring. Your job is to write a function that plays the game and decides the winner, where the winner is the first player to have an empty hand.

   A `Player` is a function: `Hand Role (listof Hand) -> Hand`

   A `Player` consumes the Hand the player is holding, a `Role`, which is `(anyof 'Landlord, 'Right, 'Left)` and a `(listof Hand)` indicating the hands played so far. The `'Landlord` starts play, followed by the `'Right` player, then the `'Left` player, then the `'Landlord` again, and so on. The list of played hands records all plays in order, including `empty` for a pass, with the most recent hand first. The players we provide for testing will always produce playable hands from the cards they are holding. While the rules for which hand may follow another are given in the bonus question of assignment #6, you do not need to worry about them. Our test players will play fairly and not attempt to cheat.

   To make things clearer, here is a helper function that consumes a list of played hands and determines if both previous players have passed:

   ```
   (define (both-passed played)
     (and (cons? played) (empty? (first played))
          (cons? (rest played)) (empty? (second played))))
   ```

   You are free to include this helper function in your solution.

   The provided file `players.rkt` contains three examples of players: `goldfish`, `cautious`, and `reckless`. For example, the goldfish player passes unless the rules of the game require it to play, in which case it will play the lowest single card it holds:

   ```
   (define (goldfish hand role played)
     (cond [(both-passed played) (list (first hand))]
           [else empty]))
   ```

To use this provided file, you must place it in the same folder as your solution file and include `(require "players.rkt")` at the top of your solution file. You are welcome to use code in this file in your solution if that helps somehow.

Write a function `(doudizhu players hands)` that consumes a list of three players, as defined above and a list of the three hands they are holding, and plays the game. Both lists are ordered by role: `'Landlord` first, `'Right` second, and `'Left` third. The function produces the role of the winning player. The critical step is removing the played cards from the hand the player is holding on the next recursive call.

For example, if it is the landlord's turn to play, your engine should:

- Call the landlord's `Player`, passing it the `Hand` the landlord is currently holding, the role `'Landlord`, and the list of hands played so far. The landlord's `Player` returns the `Hand` to play.
- Remove the played `Hand` from the `Hand` the landlord is holding.
- If the landlord now has an `empty` hand, the game ends. Produce `'Landlord`.
- Otherwise, cons the played `Hand` on to the front of the list of hands played.
- Recusively call the game engine with `'Right` as the next `Player`.

```
(define hand0
  '(3 3 3 3 4 5 6 7 7 7 9 9 Jack Jack Queen King 2 2 Black Red))
(define hand1
  '(4 4 4 5 5 6 6 7 8 9 10 Jack Queen King Ace 2 2))
(define hand2
  '(5 6 8 8 8 9 10 10 10 Jack Queen Queen King King Ace Ace Ace))
(check-expect
  (doudizhu (list goldfish goldfish goldfish) (list hand0 hand1 hand2))
  'Left)
(check-expect
  (doudizhu (list reckless goldfish goldfish) (list hand0 hand1 hand2))
 'Landlord)
(check-expect
  (doudizhu (list cautious reckless goldfish) (list hand0 hand1 hand2))
 'Landlord)
```

Place your solutions in `engine.rkt`

This concludes the list of questions for you to submit solutions. Don't forget to always check the basic test results after making a submission.