Assignment: 5  
Examples Due: Friday, October 20, 2023 8:00am  
Remainder Due: Tuesday, October 24, 2023 9:00 pm  
Coverage: Slide 44 of Module 07  
Language level: Beginning Student with List Abbreviations  
Files to submit: examples-a05.rkt, course-selection.rkt, doudizhu.rkt, bonus-a05.rkt

• Make sure you read the OFFICIAL A05 post on Piazza for the answers to frequently asked questions.

• Unless otherwise specified, you may only use Racket language features we have covered up to the coverage point above.

• It is possible that your functions will not be very efficient, and may be slow on long lists. There is no need to test your functions with excessively long lists, especially for the Dou Dizhu questions, where the game itself allows at most 20 cards in a hand.

• It’s simpler to write many of the required functions on this assignment (especially Q3) by first writing and thoroughly testing a number of helper functions. To encourage this approach, you are required provide appropriate tests for each helper function, but you do not need to provide a purpose and contract. You should know the purpose and contract (and if you don’t you should rethink what you are doing) but you do not need to explicitly write them out.

• Policies from Assignment A04 carry forward.

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

1. (15%): Complete all the required stepping problems in Module 7: Nested Lists at https://www.student.cs.uwaterloo.ca/~cs135/assign/stepping/

You should refer to the instructions from A01 Question 1 for the stepper question instructions.

2. (32%): Before the start of every term, UWWaterloo asks students to choose the courses they intend to take in the following term during the Course Selection period. In this question, we will be representing students and the courses they have chosen as a DesiredCourses.

    ;; A DesiredCourses is one of:
    ;; * empty
    ;; * (cons (list Str (listof Sym)) DesiredCourses)
An example DesiredCourses:

(define selections
(list
  (list "mpines" (list 'CS135 'MATH135 'MATH137 'ENGL109 'FINE100))
  (list "w46dles" (list 'ARBUS101 'ECON101 'ECON102 'ECON206 'LS101))
  (list "d32pines" (list 'CS115 'MATH135 'MATH137 'ENGL109 'FINE100))
  (list "gnclstan" (list 'ANTH241 'LS201 'AMATH231 'PMATH347))))

Each key-value pair in a DesiredCourses represents an enrolled student. The first element in the pair is a student’s alphanumeric Quest username (e.g. "mpines") and the second element is a list of the codes of the courses they wish to take, represented as symbols (e.g. 'CS135).

Implement the following functions:

(a) taking-course? consumes a DesiredCourses, a student’s Quest username, and a course code, and produces true if the student has selected the course (and false otherwise). For example, (taking-course? selections "d32pines" 'CS115) produces true, but (taking-course? selections "d32pines" 'CS135) produces false. If the consumed Quest username does not appear in the DesiredCourses, produce false.

(b) missed-deadline-add consumes a DesiredCourses and a student’s Quest username. Unless the student already appears in DesiredCourses, it adds the student to the end. Since an added student has sadly missed the course selection deadline, they should have no courses associated with them in the DesiredCourses. For example, (missed-deadline-add selections "w2cordur") produces:

(list
  (list "mpines" (list 'CS135 'MATH135 'MATH137 'ENGL109 'FINE100))
  (list "w46dles" (list 'ARBUS101 'ECON101 'ECON102 'ECON206 'LS101))
  (list "d32pines" (list 'CS115 'MATH135 'MATH137 'ENGL109 'FINE100))
  (list "gnclstan" (list 'ANTH241 'LS201 'AMATH231 'PMATH347))
  (list "w2cordur" empty))

(c) add-course consumes a DesiredCourses, a student’s Quest username, and a course code, and adds the given code to the end of the list of courses the student wishes to take. If the student is currently taking the course, do not change the DesiredCourses. If the Quest id does not appear in the DesiredCourses, add it. For example, (add-course selections "gnclstan" 'CS246) produces:

(list
  (list "mpines" (list 'CS135 'MATH135 'MATH137 'ENGL109 'FINE100))
  (list "w46dles" (list 'ARBUS101 'ECON101 'ECON102 'ECON206 'LS101))
  (list "d32pines" (list 'CS115 'MATH135 'MATH137 'ENGL109 'FINE100))
  (list "gnclstan" (list 'ANTH241 'LS201 'AMATH231 'PMATH347 'CS246)))
(d) `create-classlist` consumes a `DesiredCourses` and a course code, and produces a list of all the students that want to take the consumed course. The students’ Quest username must appear in the same order as in the consumed `DesiredCourses`. For example, `(create-classlist selections 'MATH135)` produces `(list "mpines" "d32pines")`.

Place your solutions to the above problems in `course-selection.rkt`.

3. **(53% in total)**: The remainder of the assignment continues from Question 3 on Assignment 4, which was based on the game Dou Dizhu. The game is played by three players, one of which is the “landlord”. The object of Dou Dizhu is to have no cards left in your hand, and the first player with an empty hand wins the game. Starting with the landlord, each player plays a combination of one or more cards following rules and limitations that we will cover later, until one player is out of cards. Only certain card combinations are valid, (e.g. a pair, three of a kind, a straight).

In the following questions, we will write functions that consume a `Hand` (a sorted list of `Card`) and produce a list of all combinations of a given valid type (e.g. all pairs) that could be played from the hand. Then we will combine these functions into a larger function that plays a single turn of the game.

You can reuse any of the functions you wrote for Assignment 4 and/or the official solutions for that assignment. The `find-kind` function should be particularly helpful, especially for the first three questions. Place your solutions in `doudizhu.rkt`.

(a) **(5%)**: The simplest combination is a single `Card`, a solo. Write a function `(solos hand)` that consumes a `Hand` and produces a `(listof Hand)`, where each produced hand consists of a single card from the consumed hand. The list should not contain duplicates. Each hand in the produced list should be different (no duplicate solos) and should be ordered by increasing card value. Remember that a `Hand` is a sorted list of `Card`, where `Card` as defined in Assignment 4.

```scheme
(check-expect
  (solos
    (list 3 3 3 3 4 4 5 5 6 7 7 7 9
         'Jack 'Jack 'Queen 'King 'Ace 2 2))
  (list (list 3) (list 4) (list 5) (list 6) (list 7)
        (list 9) (list 'Jack) (list 'Queen) (list 'King)
        (list 'Ace) (list 2)))
```

(b) **(5%)**: A pair is two cards with the same value. Write a function `(pairs hand)` that consumes a `Hand` and produces a `(listof Hand)` containing all the pairs that can be constructed from that hand. The pairs should be ordered according to card value with no duplicates.
(check-expect
  (pairs (list 3 3 3 4 5 6 7 7 7  'Jack  'Jack  'Queen  'King  'Ace 2 2))
  (list (list 3 3) (list 7 7) (list 'Jack 'Jack) (list 2 2)))

(c) (5%): A trio is three cards with the same value. Write a function (trios hand) that consumes a Hand and produces a (listof Hand) containing all the trios that can be constructed. The trios should be ordered according to card value with no duplicates.

(check-expect
  (trios (list 3 3 3 4 5 5 5 6 6 6 6 7 7 7 8 8 9  'Queen  'Queen))
  (list (list 3 3 3) (list 5 5 5) (list 6 6 6) (list 7 7 7)))

(d) (18%): In completing the previous question it was relatively straightforward to produce a sorted list, ordered by card value, with no duplicates. In the questions that follow, with more complex card combinations, it will get harder to keep lists sorted and to avoid duplicates. Instead, it will be easier to first generate combinations and then to sort and dedup them as a final step.

Write a function (sort-hands hands) that consumes a (listof Hand) and produces a sorted (listof Hand) containing the same hands, excluding duplicates. The function should first sort the list of hands and then eliminate the duplicates with a helper function.

We sort the list of hands in order of hand valuation. To do this we need to introduce two special hands, the rocket and the bomb. The rocket is the hand consisting of the two jokers (list 'Black 'Red). The rocket ranks higher than any other hand and can always be played. A bomb is a hand consisting of four cards of the same type (e.g., (list 5 5 5 5)). A bomb ranks higher than any other hand except a rocket or another bomb containing higher valued cards. A bomb can always be played, except after a rocket or another bomb containing cards of greater value.

More specifically, we sort a (listof Hand) in increasing order according to the rules below.

i. Except for rockets and bombs, hands are sorted in increasing order of length, shorter before longer.

ii. Hands of equal length are sorted in increasing order according to the value of the first card in each list. If these cards have equal value, we compare the second card in each list, and so on, i.e. the lists are sorted in lexicographical order.

iii. Bombs have greater value than any hand except rockets.

iv. Rockets have the greatest value and are always sorted to the end.

This sort order is more specific than actually required by the rule of Dou Dizhu. For example, the hand (list 4 5 6 7 8 9) (a “straight”, see below) can’t be played after hand (list 3 3 3 4 4 4) (an “airplane”) even though they are the same length, but if the rules allow one hand to be played after another, the second hand will appear after the first hand in our sort order.
We suggest starting this question by writing and testing a `hand<?` predicate, which will itself likely require `rocket?` and `bomb?` predicates.

(e) (10%): In Dou Dizhu, a straight consists of a five or more cards in consecutive order, excluding twos and jokers. Straights can only include 3, 4, 5, 6, 7, 8, 9, 10, 'Jack, 'Queen, 'King, or 'Ace in that order. Write a function `(straights hand)` that consumes a `Hand` and produces a `listof Hand` containing all the straights that can be constructed from that hand. The list should be sorted with duplicates removed.

(f) (5%): Three or more pairs can also form a straight of pairs. For example, given the hand `(list 3 3 3 4 5 5 6 6 6 7 7 7 8 8 8 8 9 'Ace 'Ace)` we can form the straight pairs `(list (list 5 5 6 6 7 7) (list 6 6 7 7 8 8) (list 5 5 6 6 7 7 8 8))`. Write a function `(straight-pairs hand)` that consumes a `Hand` and produces a `listof Hand` containing all the straight pairs that can be constructed from that hand. The list should be sorted with duplicates removed. Note that, unlike poker and some other games, two pairs is not a valid hand. A straight of pairs must contain three or more pairs of cards with different values.
In writing the function, consider how you can use `find-kind` as a helper. It might also be helpful to extend helper functions you may have written for `straights`. It should be possible to implement this function with very little extra code by making a small extension to your solutions for previous questions: 1) find all the pairs, 2) collapse the pairs into a list of cards, 3) find the straights of length 3 or more, and then 4) convert the cards back into pairs.

(g) (5%): An airplane is a straight of two or more trios. Write a function `(airplanes hand)` that consumes a `Hand` and produces a `(listof Hand)` containing all the airplanes that can be constructed from that hand. The list should be sorted with duplicates removed.

```scheme
(check-expect (airplanes (list 3 3 3 4 4 4 4)) (list (list 3 3 3 4 4 4)))

(check-expect
 (airplanes
  (list 'Queen 'Queen 'Queen 'Queen
       'King 'King 'King
       'Ace 'Ace 'Ace 2 2 2 ))
  (list (list 'Queen 'Queen 'Queen 'King 'King 'King))
  (list 'King 'King 'King 'Ace 'Ace 'Ace)
  (list 'Queen 'Queen 'Queen 'King 'King 'King
       'Ace 'Ace 'Ace))
```

While this ends the required implementation of Dou Dizhu for this assignment, the bonus question lets you put these pieces together to play a single turn of the game. In a future assignment you will write more of the game engine.

This concludes the list of questions for you to submit solutions, but see the following pages as well, especially the bonus. Don’t forget to always check the basic test results after making a submission.
4. **(Bonus 20%)**: We are now ready to put all the pieces together and play a single turn of Dou Dizhu. In the actual game there are some combinations of cards that we have not required you to implement above (e.g. a “full house” consisting of a trio and a pair) but these are enough combinations to play a basic version of the game.

In this question we assume that the previous player has already played some cards from their hand, and your job is to follow with a valid play. In Dou Dizhu, you can only follow with the same combination of cards (pairs, trio, airplane, etc.) or with a rocket or with a bomb. If you follow with the same combination of cards, they must have a higher value according to the sort order above.

For example, if the previous player has played \((\text{list } 5\ 5\ 5)\) and you are holding

\[
(\text{list } 3\ 3\ 3\ 3\ 4\ 4\ 6\ 6\ 6\ 6\ 7\ 7\ \text{'King}\ \text{'King}\ \text{'King}\ \text{'Black}\ \text{'Red})
\]

you can follow with:

- \((\text{list } 3\ 3\ 3\ 3)\) because it’s a bomb,
- \((\text{list } 6\ 6\ 6)\) because it’s a trio with higher value,
- \((\text{list } 6\ 6\ 6\ 6)\) because it’s a bomb,
- \((\text{list } \text{'King}\ \text{'King}\ \text{'King})\) because it’s a trio with higher value, or
- \((\text{list } \text{'Black}\ \text{'Red})\), the rocket.

In a real game, which one you pick depends on your strategy, the cards you are holding, other cards that have been played in previous turns, and any other information you might have available. You can’t play \((\text{list } 4\ 4\ 4)\), even though it’s a triple because it has lower value than \((\text{list } 5\ 5\ 5)\).

Write a function \((\text{follow previous holding})\) that consumes the previous play and the cards that you are holding and returns a list of all hands you could play to follow. The list should be sorted with duplicates removed. You can assume that \(\text{previous}\) is a valid card combination (e.g., a pair, straight, etc.).

Place your solution in \texttt{bonus-a05.rkt}.

\[
\text{(check-expect}
\quad (\text{follow } (\text{list } 5\ 5\ 5))
\quad (\text{list } 3\ 3\ 3\ 3\ 4\ 4\ 6\ 6\ 6\ 6\ 7\ 7\ \text{'King}\ \text{'King}\ \text{'King}\ \text{'Black}\ \text{'Red})
\quad (\text{list } 6\ 6\ 6) (\text{list } \text{'King}\ \text{'King})
\quad (\text{list } 3\ 3\ 3\ 3) (\text{list } 6\ 6\ 6\ 6) (\text{list } \text{'Black}\ \text{'Red}))
\text{)}
\]


**Enhancements:** Reminder—enhancements are for your interest and are not to be handed in.

There is a strong connection between recursion and induction. Mathematical induction is the proof technique often used to prove the correctness of programs that use recursion; the structure of the induction parallels the structure of the function. As an example, consider the following function, which computes the sum of the first $n$ natural numbers.

```racket
(define (sum-first n)
  (cond
    [(zero? n) 0]
    [else (+ n (sum-first (sub1 n)))]))
```

To prove this program correct, we need to show that, for all natural numbers $n$, the result of evaluating $(\text{sum-first } n)$ is $\sum_{i=0}^{n} i$. We prove this by induction on $n$.

**Base case:** $n = 0$. When $n = 0$, we can use the semantics of Racket to evaluate $(\text{sum-first } 0)$ as follows:

```
(sum-first 0) ; =>
(cond [(zero? 0) 0][else ...]) ; =>
(cond [true 0][else ...]) ; =>
0
```

Since $0 = \sum_{i=0}^{0} i$, we have proved the base case.

**Inductive step:** Given $n > 0$, we assume that the program is correct for the input $n - 1$, that is, $(\text{sum-first } (\text{sub1 } n))$ evaluates to $\sum_{i=0}^{n-1} i$. The evaluation of $(\text{sum-first } n)$ proceeds as follows:

```
(sum-first n) ; =>
(cond [(zero? n) 0][else ...]) ; (we know $n > 0$) =>
(cond [false 0][else ...]) ; =>
(cond [else (+ n (sum-first (sub1 n)))])) ; =>
(+ n (sum-first (sub1 n)))
```

Now we use the inductive hypothesis to assert that $(\text{sum-first } (\text{sub1 } n))$ evaluates to $s = \sum_{i=0}^{n-1} i$. Then $(+ n s)$ evaluates to $n + \sum_{i=0}^{n-1} i$, or $\sum_{i=0}^{n} i$, as required. This completes the proof by induction.

Use a similar proof to show that, for all natural numbers $n$, $(\text{sum-first } n)$ evaluates to $(n^2 + n)/2$.

**Note:** Summing the first $n$ natural numbers in imperative languages such as C++ or Java would be done using a `for` or `while` loop. But proving such a loop correct, even such a simple loop, is
considerably more complicated, because typically some variable is accumulating the sum, and its value keeps changing. Thus the induction needs to be done over time, or number of statements executed, or number of iterations of the loop, and it is messier because the semantic model in these languages is so far-removed from the language itself. Special temporal logics have been developed to deal with the problem of proving larger imperative programs correct.

The general problem of being confident, whether through a mathematical proof or some other formal process, that the specification of a program matches its implementation is of great importance in safety-critical software, where the consequences of a mismatch might be quite severe (for instance, when it occurs with software to control an airplane, or a nuclear power plant).