Programming Challenge: Three Card Flush

This programming challenge takes the form of a "lab", kind of like a CS1 lab, only longer, and grounded in more advanced programming concepts and constructs.

Part 1: The Deck and the Shuffle

Please do the following tasks which pertain to the deck of cards that will be used in each play of a TCF game.

Note that `*deck*` is one of the featured global variable in this design. The `establish-shuffled-deck` function that you are asked to write will bind this featured global variable to a shuffled deck of cards.

1. `make-deck`

   (a) By analogy with the `make-notes` function written in class, write a `make-deck` function which returns a standard deck of 52 cards, where each card is represented as a cons cell with a rank drawn from {2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king, ace} and a suit is drawn from {club, diamond, heart, spade}. When I say "by analogy", I mean that this program must use `mapcan` and call a function that uses `mapcar`.

   (b) Type in the following program and use it to test your `make-deck` function.

   ```
   ( defun demo--make-deck ()
     ( format t ">>> Testing: make-deck~%" )
     ( setf deck ( make-deck ) )
     ( format t "--- deck = ~A~%" deck )
     ( format t "--- number of cards in deck = ~A~%" ( length deck ) )
     nil
   )
   ```

   (c) Run the test program. The output from the demo should look like (except for paragraphing) this:

   ```
   []> ( demo--make-deck )
   >>> Testing: make-deck
   --- deck ...
   ((2 . CLUB) (3 . CLUB) (4 . CLUB) (5 . CLUB) (6 . CLUB) (7 . CLUB) (8 . CLUB) (9 . CLUB)
    (10 . CLUB) (JACK . CLUB) (QUEEN . CLUB) (KING . CLUB) (ACE . CLUB) (2 . DIAMOND) (3 . DIAMOND
    (4 . DIAMOND) (5 . DIAMOND) (6 . DIAMOND) (7 . DIAMOND) (8 . DIAMOND) (9 . DIAMOND)
    (10 . DIAMOND) (JACK . DIAMOND) (QUEEN . DIAMOND) (KING . DIAMOND) (ACE . DIAMOND)
    (2 . HEART) (3 . HEART) (4 . HEART) (5 . HEART) (6 . HEART) (7 . HEART) (8 . HEART)
    (9 . HEART) (10 . HEART) (JACK . HEART) (QUEEN . HEART) (KING . HEART) (ACE . HEART)
    (2 . SPADE) (3 . SPADE) (4 . SPADE) (5 . SPADE) (6 . SPADE) (7 . SPADE) (8 . SPADE)
    (9 . SPADE) (10 . SPADE) (JACK . SPADE) (QUEEN . SPADE) (KING . SPADE) (ACE . SPADE))
   --- number of cards in deck = 52
   NIL
   []>
   ```

2. `establish-shuffled-deck`

(a) Add the following code to your file:

```
( defun establish-shuffled-deck ()
  ( setf *deck* ( shuffle ( make-deck ) ) )
  nil
)
```

(b) Refine the `shuffle-deck` function by defining the `shuffle` program by **refining the following recursive pseudocode**, keeping in mind that the deck is represented as a list of cards:

```
to SHUFFLE a given DECK do
    if the DECK is empty then
        return it (that is, return the empty deck)
    else
        let CARD = a card randomly selected from the DECK
        remove the CARD from the DECK
        add the CARD to the beginning of the SHUFFLE of the SLIGHTLY SMALLER DECK
    end
end
```

(c) Type in the following program and use it to test your `make-deck` function.

```
( defun demo--establish-shuffled-deck ()
  ( format t ">>> Testing: shuffle-deck~%" )
  ( shuffle-deck )
  ( format t "--- *deck* ... ~A~%" *deck* )
  ( format t "--- number of cards in *deck* = ~A~%" ( length *deck* ) )
  nil
)
```

(d) Run the test program. The output from the demo should look something like (except for paragraphing) this:

```
[]> ( demo--establish-shuffled-deck )
>>> Testing: shuffle-deck
--- *deck* ...
((8 . SPADE) (7 . DIAMOND) (KING . CLUB) (9 . SPADE) (9 . HEART) (8 . HEART)
 (QUEEN . HEART) (5 . CLUB) (4 . HEART) (8 . DIAMOND) (3 . CLUB) (2 . HEART)
 (4 . CLUB) (2 . DIAMOND) (2 . SPADE) (JACK . HEART) (JACK . SPADE) (KING . DIAMOND)
 (10 . SPADE) (ACE . DIAMOND) (QUEEN . SPADE) (10 . CLUB) (KING . SPADE) (JACK . CLUB)
 (6 . HEART) (KING . HEART) (JACK . DIAMOND) (6 . CLUB) (9 . CLUB) (ACE . SPADE)
 (10 . HEART) (6 . SPADE) (6 . DIAMOND) (9 . DIAMOND) (7 . HEART) (4 . SPADE)
 (8 . CLUB) (3 . DIAMOND) (7 . SPADE) (5 . DIAMOND) (5 . SPADE) (QUEEN . DIAMOND)
 (2 . CLUB) (QUEEN . CLUB) (ACE . CLUB) (3 . HEART) (7 . CLUB) (ACE . HEART) (5 . HEART)
 (3 . SPADE) (4 . DIAMOND) (10 . DIAMOND))
--- number of cards in *deck* = 52
NIL
[]>
```

3. Save your work!

Save the functions that your wrote for this part of the program, including the demo functions, to the section of your solution document dedicated to this part of the program. Also, save the demos that you generated to the to the section of your solution document dedicated to this part of the program.

Part 2: The Hands, the Deal and the Discard

Please do the following tasks which pertain to the two three card hands, which we will loosely refer to as "hand 1" and "hand 2", and the deal of three cards to each hand from a fresh deck of cards.

Note that `*hand1*` and `*hand2*` are among the featured global variables in this design.

1. `deal-hands`

   (a) Please type in the following program which will serve, once it is refined, to establish the initial hands for a game of TCF. Significantly, it binds the global variables representing the hands appropriately.

   ```
   ( defun deal-hands ()
     ( establish-shuffled-deck )
     ( setf *hand1* () )
     ( setf *hand2* () )
     ( deal-card-to-hand1 )
     ( deal-card-to-hand2 )
     ( deal-card-to-hand1 )
     ( deal-card-to-hand2 )
     ( deal-card-to-hand1 )
     ( deal-card-to-hand2 )
     nil
   )
   ```

   (b) Refine the `deal-hands` function by defining the `deal-card-to-hand1` function and the `deal-card-to-hand1` function:

      i. Define the `deal-card-to-hand1` function which removes the **first** card from the deck (denoted by `*deck*`) and adds it as the **last** element of hand 1 (denoted by `*hand1*`).

      ii. Define the `deal-card-to-hand2` function which removes the **first** card from the deck (denoted by `*deck*`) and adds it as the **last** element of hand 2 (denoted by `*hand2*`).

   (c) Type in the following program and use it to test your `deal-hands` function.

   ```
   ( defun demo--deal-hands ()
     ( format t ">>> Testing: deal-hands~%" )
     ( deal-hands )
     ( format t "--- *hand1* = ~A~%" *hand1* )
     ( format t "--- *hand2* = ~A~%" *hand2* )
     ( format t "--- number of cards in *deck* = ~A~%" ( length *deck* ) )
     nil
   )
   ```

   (d) Run the demo at least twice, in order to assure that the program is working properly. The output from the demo should look something like (except for specific cards) this:

   ```
   []> ( demo--deal-hands )
   >>> Testing: deal-hands
   --- *hand1* = ((10 . CLUB) (10 . DIAMOND) (10 . HEART))
   --- *hand2* = ((8 . SPADE) (5 . SPADE) (QUEEN . DIAMOND))
   --- number of cards in *deck* = 46
   ```

```
NIL
[]> ( demo--deal-hands )
>>> Testing: deal-hands
--- *hand1* = ((KING . DIAMOND) (7 . DIAMOND) (6 . CLUB))
--- *hand2* = ((5 . DIAMOND) (5 . CLUB) (10 . SPADE))
--- number of cards in *deck* = 46
NIL
[]>
```

2. `randomly-discard-cards`

   (a) Please type in the following program which will serve, once it is refined, to change the two hands by simulating the random discard of one card from each hand. You might like to imagine two players, one responsible for each hand, discarding cards by means of random selection.

```
( defun randomly-discard-cards ()
  ( randomly-discard-card-from-hand1 )
  ( randomly-discard-card-from-hand2 )
  nil
)
```

   (b) Refine the `randomly-discard-cards` function by defining the `randomly-discard-card-from-hand1` function and the `randomly-discard-card-from-hand2` function:

      i. Define the `randomly-discard-card-from-hand1` function by randomly replaceing one of the three cards in hand 1 (denoted by `*hand1*`) with the empty list. Note that this function must actually replace the old value of `*hand1*` with a new value.

      ii. Define the `randomly-discard-card-from-hand2` function by randomly replaceing one of the three cards in hand 2 (denoted by `*hand2*`) with the empty list. Note that this function must actually replace the old value of `*hand2*` with a new value.

   (c) Type in the following program and use it to test the `randomly-discard-cards` function (and hence the two functions that you have been asked to write.

```
( defun demo--randomly-discard-cards ()
  ( format t ">>> Testing: randomly-discard-cards~%" )
  ( deal-hands )
  ( format t "--- *hand1* = ~A~%" *hand1* )
  ( format t "--- *hand2* = ~A~%" *hand2* )
  ( randomly-discard-cards )
  ( format t "--- *hand1* = ~A~%" *hand1* )
  ( format t "--- *hand2* = ~A~%" *hand2* )
  nil
)
```

   (d) Run the demo a number of times, in order to assure that the program is working properly. The output from the demo should look something like (except for specific cards) this:

```
[]> ( demo--randomly-discard-cards )
>>> Testing: randomly-discard-cards
--- *hand1* = ((7 . CLUB) (8 . SPADE) (9 . CLUB))
--- *hand2* = ((ACE . HEART) (4 . DIAMOND) (ACE . SPADE))
--- *hand1* = (NIL (8 . SPADE) (9 . CLUB))
--- *hand2* = ((ACE . HEART) NIL (ACE . SPADE))
NIL
[]> ( demo--randomly-discard-cards )
```

```
>>> Testing: randomly-discard-cards
--- *hand1* = ((10 . DIAMOND) (3 . DIAMOND) (QUEEN . DIAMOND))
--- *hand2* = ((8 . CLUB) (KING . SPADE) (JACK . SPADE))
--- *hand1* = ((10 . DIAMOND) (3 . DIAMOND) NIL)
--- *hand2* = ((8 . CLUB) (KING . SPADE) NIL)
NIL
[]> ( demo--randomly-discard-cards )
>>> Testing: randomly-discard-cards
--- *hand1* = ((ACE . DIAMOND) (QUEEN . SPADE) (JACK . SPADE))
--- *hand2* = ((5 . SPADE) (3 . CLUB) (3 . DIAMOND))
--- *hand1* = ((ACE . DIAMOND) (QUEEN . SPADE) NIL)
--- *hand2* = ((5 . SPADE) NIL (3 . DIAMOND))
NIL
[]> ( demo--randomly-discard-cards )
>>> Testing: randomly-discard-cards
--- *hand1* = ((9 . CLUB) (5 . SPADE) (JACK . DIAMOND))
--- *hand2* = ((10 . CLUB) (QUEEN . HEART) (JACK . CLUB))
--- *hand1* = (NIL (5 . SPADE) (JACK . DIAMOND))
--- *hand2* = ((10 . CLUB) (QUEEN . HEART) NIL)
NIL
[]> ( demo--randomly-discard-cards )
>>> Testing: randomly-discard-cards
--- *hand1* = ((ACE . HEART) (10 . DIAMOND) (JACK . DIAMOND))
--- *hand2* = ((7 . SPADE) (8 . CLUB) (4 . DIAMOND))
--- *hand1* = ((ACE . HEART) NIL (JACK . DIAMOND))
--- *hand2* = ((7 . SPADE) NIL (4 . DIAMOND))
NIL
[17]>
```

3. Save your work!

   Save the functions that your wrote for this part of the program, including the demo functions, to the section of your solution document dedicated to this part of the program. Also, save the demos that you generated to the to the section of your solution document dedicated to this part of the program.

Part 3: Replacing Cards in Hands, Taking Turns

Please do the following tasks which pertain to the hands. The first has to do with **filling the void** in each hand. The second encapsulates, for each hand, a discard function with a corrpesonding fill the void function to **perform a turn**.

Please keep in mind that `*hand1*` and `*hand2*` are among the featured global variables in this design.

1. `replace-cards`

   (a) Please type in the following program which will serve, once it is refined, fill the void that was left in each hand as a result of the discard operators. Significantly, it re-binds the global variables representing the hands appropriately.

   ```
   ( defun replace-cards ()
   ```

```
  ( replace-card-in-hand1 )
  ( replace-card-in-hand2 )
  nil
)
```

(b) Refine the `replace-cards` function by defining the `replace-card-in-hand1` function and the `replace-card-in-ha` function:

   i. Define the `replace-card-in-hand1` function which removes the **first** card from the deck (denoted by `*deck*`) and uses it to fill the void in hand 1 (denoted by `*hand1*`).

   ii. Define the `replace-card-in-hand2` function which removes the **first** card from the deck (denoted by `*deck*`) and uses it to fill the void in hand 2 (denoted by `*hand2*`).

(c) Type in the following program and use it to test your `replace-cards` function.

```
( defun demo--replace-cards ()
  ( format t ">>> Testing: replace-cards~%" )
  ( deal-hands )
  ( format t "--- *hand1* = ~A~%" *hand1* )
  ( format t "--- *hand2* = ~A~%" *hand2* )
  ( randomly-discard-cards )
  ( format t "--- *hand1* = ~A~%" *hand1* )
  ( format t "--- *hand2* = ~A~%" *hand2* )
  ( replace-cards )
  ( format t "--- *hand1* = ~A~%" *hand1* )
  ( format t "--- *hand2* = ~A~%" *hand2* )
  nil
)
```

(d) Run the demo a sufficient number of times to see each of the three slots being filled, in order to assure that the program is working properly. The output from the demo should look something like this:

```
[2]> ( demo--replace-cards )
>>> Testing: replace-cards
--- *hand1* = ((8 . SPADE) (KING . CLUB) (9 . HEART))
--- *hand2* = ((7 . DIAMOND) (9 . SPADE) (8 . HEART))
--- *hand1* = ((8 . SPADE) (KING . CLUB) NIL)
--- *hand2* = ((7 . DIAMOND) (9 . SPADE) NIL)
--- *hand1* = ((8 . SPADE) (KING . CLUB) (QUEEN . HEART))
--- *hand2* = ((7 . DIAMOND) (9 . SPADE) (5 . CLUB))
NIL
[]> ( demo--replace-cards )
>>> Testing: replace-cards
--- *hand1* = ((10 . CLUB) (10 . DIAMOND) (10 . HEART))
--- *hand2* = ((8 . SPADE) (5 . SPADE) (QUEEN . DIAMOND))
--- *hand1* = (NIL (10 . DIAMOND) (10 . HEART))
--- *hand2* = ((8 . SPADE) NIL (QUEEN . DIAMOND))
--- *hand1* = ((6 . DIAMOND) (10 . DIAMOND) (10 . HEART))
--- *hand2* = ((8 . SPADE) (KING . SPADE) (QUEEN . DIAMOND))
NIL
[]> ( demo--replace-cards )
>>> Testing: replace-cards
--- *hand1* = ((ACE . SPADE) (8 . CLUB) (9 . HEART))
--- *hand2* = ((6 . CLUB) (7 . DIAMOND) (5 . SPADE))
--- *hand1* = ((ACE . SPADE) (8 . CLUB) NIL)
--- *hand2* = ((6 . CLUB) (7 . DIAMOND) NIL)
--- *hand1* = ((ACE . SPADE) (8 . CLUB) (8 . HEART))
--- *hand2* = ((6 . CLUB) (7 . DIAMOND) (ACE . DIAMOND))
NIL
[]> ( demo--replace-cards )
>>> Testing: replace-cards
--- *hand1* = ((10 . SPADE) (KING . CLUB) (JACK . HEART))
--- *hand2* = ((3 . SPADE) (5 . DIAMOND) (4 . HEART))
--- *hand1* = ((10 . SPADE) (KING . CLUB) NIL)
--- *hand2* = (NIL (5 . DIAMOND) (4 . HEART))
--- *hand1* = ((10 . SPADE) (KING . CLUB) (ACE . CLUB))
--- *hand2* = ((10 . CLUB) (5 . DIAMOND) (4 . HEART))
NIL
[]>
```

2. `players-each-take-a-turn`

   (a) Please type in the following program which will serve to simulate the action of taking a turn by both players.

```
( defun players-each-take-a-turn ()
  ( randomly-discard-cards )
  ( replace-cards )
  nil
)
```

   (b) Type in the following program and use it to test the `players-each-take-a-turn` function.

```
( defun demo--players-each-take-a-turn ()
  ( format t ">>> Testing: players-each-take-a-turn~%" )
  ( deal-hands )
  ( format t "--- The hands ...~%" )
  ( format t "--- *hand1* = ~A~%" *hand1* )
  ( format t "--- *hand2* = ~A~%" *hand2* )
  ( players-each-take-a-turn )
  ( format t "--- Each player takes a turn ...~%" )
  ( format t "--- *hand1* = ~A~%" *hand1* )
  ( format t "--- *hand2* = ~A~%" *hand2* )
  ( players-each-take-a-turn )
  ( format t "--- Each player takes a turn ...~%" )
  ( format t "--- *hand1* = ~A~%" *hand1* )
  ( format t "--- *hand2* = ~A~%" *hand2* )
  ( players-each-take-a-turn )
  ( format t "--- Each player takes a turn ...~%" )
  ( format t "--- *hand1* = ~A~%" *hand1* )
  ( format t "--- *hand2* = ~A~%" *hand2* )
  ( players-each-take-a-turn )
  ( format t "--- Each player takes a turn ...~%" )
  ( format t "--- *hand1* = ~A~%" *hand1* )
  ( format t "--- *hand2* = ~A~%" *hand2* )
  nil
)
```

   (c) Just run the program once, but be sure to reflect upon the four turns that are taken by the demo. The output from the demo should look something like this:

```
>>> Testing: players-each-take-a-turn
--- The hands ...
--- *hand1* = ((6 . CLUB) (10 . SPADE) (ACE . SPADE))
--- *hand2* = ((KING . SPADE) (10 . HEART) (ACE . CLUB))
--- Each player takes a turn ...
--- *hand1* = ((6 . CLUB) (ACE . HEART) (ACE . SPADE))
--- *hand2* = ((KING . SPADE) (10 . HEART) (7 . CLUB))
--- Each player takes a turn ...
--- *hand1* = ((5 . HEART) (ACE . HEART) (ACE . SPADE))
--- *hand2* = ((JACK . CLUB) (10 . HEART) (7 . CLUB))
--- Each player takes a turn ...
--- *hand1* = ((2 . HEART) (ACE . HEART) (ACE . SPADE))
--- *hand2* = ((3 . HEART) (10 . HEART) (7 . CLUB))
--- Each player takes a turn ...
--- *hand1* = ((2 . HEART) (ACE . HEART) (2 . DIAMOND))
--- *hand2* = ((8 . CLUB) (10 . HEART) (7 . CLUB))
NIL
```

3. Save your work!

   Save the functions that your wrote for this part of the program, including the demo functions, to the section of your solution document dedicated to this part of the program. Also, save the demos that you generated to the to the section of your solution document dedicated to this part of the program.

Part 4: Hand Analysis

The analysis of a hand which is **not** a flush will simply be the symbolic atom BUST. For a hand which **is** a flush, the syntax for hand analysis will be the following **list** of items:

1. The rank of the high card in the hand
2. The word HIGH
3. The words STRAIGHT if the hand happens to be a straight
4. The suit of the flush
5. The world FLUSH

For example:

1. The analysis of ((5 . SPADE) (3 . DIAMOND) (4 . SPADE)) will be: BUST
2. The analysis of ((5 . CLUB) (9 . CLUB) (4 . CLUB)) will be: (9 HIGH CLUB FLUSH)
3. The analysis of ((QUEEN . HEART) (ACE . HEART) (KING . HEART)) will be: (ACE HIGH STRAIGHT HEART FLUSH)

For the record, this is the linear ordering, from low to high, of the cards in a deck: (2 . CLUB) (2 . DIAMOND) (2 . HEART) (2 . SPADE) (3 . CLUB) (3 . DIAMOND) (3 . HEART) (3 . SPADE) (4 . CLUB) (4 . DIAMOND) (4 . HEART) (4 . SPADE) (5 . CLUB) (5 . DIAMOND) (5 . HEART) (5 . SPADE) (6 . CLUB) (6 . DIAMOND) (6 . HEART) (6 . SPADE) (7 . CLUB) (7 . DIAMOND) (7 . HEART) (7 . SPADE) (8 . CLUB) (8 . DIAMOND) (8 . HEART) (8 . SPADE) (9 . CLUB) (9 . DIAMOND) (9 . HEART) (9 . SPADE) (10 . CLUB) (10 . DIAMOND) (10 . HEART) (10 . SPADE) (JACK . CLUB) (JACK . DIAMOND) (JACK . HEART) (JACK . SPADE) (QUEEN

. CLUB) (QUEEN . DIAMOND) (QUEEN . HEART) (QUEEN . SPADE) (KING . CLUB) (KING . DIAMOND) (KING . HEART) (KING . SPADE) (ACE . CLUB) (ACE . DIAMOND) (ACE . HEART) (ACE . SPADE)

Please do the following tasks which pertain to the analysys of a TCF hand. The first has to do with checking to see if the hand is a **flush**. The second has to do with determining the **high card** in a hand. The third has to do with checking to see if the hand is a **straight**. And the fourth makes use of these three in performing hand analysis for a hand in the game of TCF.

1. `flush-p`
   (a) Please define the function `flush-p` which takes one argument, a hand of cards (of any length) where each card is represented as a dotted pair, and returns true if the hand is a flush, false if not. **Constraint**: Use the approach (involving `mapcar`) suggested in class.
   (b) Type in the following program and use it to test your `flush-p` function.

```
( defun demo--flush-p ( &aux hand )
  ( format t ">>> Testing: flush-p~%" )
  ( setf hand '( ( 2 . club ) ( ace . club ) ( 10 . club ) ) )
  ( format t "~A " hand )
  ( if ( flush-p hand )
    ( format t "is a flush~%" )
    ( format t "is not a flush~%" )
  )
  ( setf hand '( ( jack . diamond ) ( 9 . diamond ) ( 5 . diamond ) ) )
  ( format t "~A " hand )
  ( if ( flush-p hand )
    ( format t "is a flush~%" )
    ( format t "is not a flush~%" )
  )
  ( setf hand '( ( jack . heart ) ( 10 . heart ) ( 9 . heart ) ) )
  ( format t "~A " hand )
  ( if ( flush-p hand )
    ( format t "is a flush~%" )
    ( format t "is not a flush~%" )
  )
  ( setf hand '( ( 2 . spade) ( 3 . spade ) ( ace . spade ) ) )
  ( format t "~A " hand )
  ( if ( flush-p hand )
    ( format t "is a flush~%" )
    ( format t "is not a flush~%" )
  )
  ( setf hand '( ( 10 . spade) ( 5 . diamond ) ( ace . spade ) ) )
  ( format t "~A " hand )
  ( if ( flush-p hand )
    ( format t "is a flush~%" )
    ( format t "is not a flush~%" )
  )
  ( setf hand '( ( 8 . club) ( 9 . diamond ) ( 10 . heart ) ) )
  ( format t "~A " hand )
  ( if ( flush-p hand )
    ( format t "is a flush~%" )
    ( format t "is not a flush~%" )
  )
)
```

(c) Run the demo one time, in order to assure that the program is working properly. The output from the demo should look something like this:

```
>>> Testing: flush-p
((2 . CLUB) (ACE . CLUB) (10 . CLUB)) is a flush
((JACK . DIAMOND) (9 . DIAMOND) (5 . DIAMOND)) is a flush
((JACK . HEART) (10 . HEART) (9 . HEART)) is a flush
((2 . SPADE) (3 . SPADE) (ACE . SPADE)) is a flush
((10 . SPADE) (5 . DIAMOND) (ACE . SPADE)) is not a flush
((8 . CLUB) (9 . DIAMOND) (10 . HEART)) is not a flush
NIL
```

2. `high-card`

(a) Please define the function `high-card` which takes one argument, a hand of cards (of any length) where each card is represented as a dotted pair, and returns the high card of the hand.

(b) Type in the following program and use it to test your `high-card` function.

```
( defun demo--high-card ()
  ( format t ">>> Testing: high-card~%" )
  ( setf hand '( ( 10 . heart ) ( 5 . club ) ( queen . spade ) ( 7 . heart ) ) )
  ( format t "~A is the high card of~%  ~A~%" ( high-card hand ) hand )
  ( setf hand '( ( 2 . diamond ) ( 2 . club ) ( 10 . heart ) ( 4 . diamond )
    ( ace . club ) ) )
  ( format t "~A is the high card of~%  ~A~%" ( high-card hand ) hand )
  ( setf hand '( ( ace . diamond ) ( ace . club ) ( 5 . spade ) ) )
  ( format t "~A is the high card of~%  ~A~%" ( high-card hand ) hand )
  nil
)
```

(c) Run the demo one time, in order to assure that the program is working properly. The output from the demo should look something like this:

```
>>> Testing: high-card
(QUEEN . SPADE) is the high card of
  ((10 . HEART) (5 . CLUB) (QUEEN . SPADE) (7 . HEART))
(ACE . CLUB) is the high card of
  ((2 . DIAMOND) (2 . CLUB) (10 . HEART) (4 . DIAMOND) (ACE . CLUB))
(ACE . DIAMOND) is the high card of
  ((ACE . DIAMOND) (ACE . CLUB) (5 . SPADE))
NIL
```

3. `straight-p`

(a) Please define the function `straight-p` which takes one argument, a hand of cards (of any length) where each card is represented as a dotted pair, and returns true if the hand is a straight, false if not.

(b) Type in the following program and use it to test your `straight-p` function.

```
( defun demo--straight-p ()
  ( format t ">>> Testing: straight-p~%" )
  ( setf hand '( ( 5 . spade) ( 3 . diamond ) ( 4 . spade ) ( 6 . club )) )
  ( format t "~A " hand )
  ( if ( straight-p hand )
    ( format t "is a straight~%" )
    ( format t "is not a straight~%" )
  )
  ( setf hand '( ( 5 . spade) ( 7 . diamond ) ( 4 . spade ) ( 8 . club ) ) )
  ( format t "~A " hand )
  ( if ( straight-p hand )
    ( format t "is a straight~%" )
    ( format t "is not a straight~%" )
  )
  ( setf hand '( ( king . heart ) ( queen . diamond ) ( ace . spade ) ( 10 . club )
    ( jack . diamond ) ) )
  ( format t "~A " hand )
  ( if ( straight-p hand )
    ( format t "is a straight~%" )
    ( format t "is not a straight~%" )
  )
  ( setf hand '( ( ace . club ) ( 2 . diamond ) ( 3 . spade ) ) )
  ( format t "~A " hand )
  ( if ( straight-p hand )
    ( format t "is a straight~%" )
    ( format t "is not a straight~%" )
  )
  nil
)
```

(c) Run the demo one time, in order to assure that the program is working properly. The output from the demo should look something like this:

```
>>> Testing: straight-p
((5 . SPADE) (3 . DIAMOND) (4 . SPADE) (6 . CLUB)) is a straight
((5 . SPADE) (7 . DIAMOND) (4 . SPADE) (8 . CLUB)) is not a straight
((KING . HEART) (QUEEN . DIAMOND) (ACE . SPADE) (10 . CLUB) (JACK . DIAMOND)) is a straight
((ACE . CLUB) (2 . DIAMOND) (3 . SPADE)) is not a straight
NIL
```

4. `analyze-hand`

(a) Please define the function `analyze-hand` which takes one argument, a **three-card** hand where each card is represented as a dotted pair, and returns the analysis of the hand, consistent with the aforementioned syntax. **Hint**: Be sure to make good use of `flush-p` and `high-card` and `straight-p`.

(b) Type in the following program and use it to test your `analyze-hand` function.

```
( defun demo--analyze-hand ()
  ( format t ">>> Testing: analyze-hand~%" )
  ( setf hand '( ( 5 . spade) ( 3 . diamond ) ( 4 . spade ) ) )
  ( format t "~A is a ~A~%" hand ( analyze-hand hand ) )
  ( setf hand '( ( 5 . club) ( 9 . club ) ( 4 . club ) ) )
  ( format t "~A is a ~A~%" hand ( analyze-hand hand ) )
  ( setf hand '( ( queen . heart ) ( ace . heart ) ( king . heart ) ) )
  ( format t "~A is a ~A~%" hand ( analyze-hand hand ) )
  nil
)
```

(c) Run the demo one time, in order to assure that the program is working properly. The output from the demo should look something like this:

```
>>> Testing: analyze-hand
((5 . SPADE) (3 . DIAMOND) (4 . SPADE)) is a BUST
((5 . CLUB) (9 . CLUB) (4 . CLUB)) is a (9 HIGH CLUB FLUSH)
((QUEEN . HEART) (ACE . HEART) (KING . HEART)) is a (ACE HIGH STRAIGHT HEART FLUSH)
NIL
```

5. Save your work!

   Save the functions that your wrote for this part of the program, including the demo functions, to the section of your solution document dedicated to this part of the program. Also, save the demos that you generated to the to the section of your solution document dedicated to this part of the program.

Part 5: Game State and End of Game Reporting

The variable `*game-state*` is a featured global variable in this design. It will be given a value just prior to testing to see if the game is over, and it will be used as a means by which to report the result of the game.

The "game state" is considered, quite simply, to be a list of length two consisting of the current analysis of hand 1 and the current analysis of hand 2. For example, the following lists represent possible game states:

1. (bust bust)
2. ((10 high club flush) bust)
3. (bust (queen high straight heart flush))
4. ((6 high straight club flush) (ace high spade flush))
5. ((king high club flush) (9 high straight diamond flush))
6. ((10 high straight spade flush) (10 high straight heart flush))
7. ((queen high club flush) (queen high diamond flush))

1. `analyze-game`

   (a) Please define the parameterless function `analyze-game` which binds the global variable `*game-state*` to the state of the game, using the syntax that was just provided. **Hint**: This should be a simple one line program that makes good use of the `analyze-hand` function.

   (b) Type in the following program and use it to test your `analize-game` function.

```
( defun demo--analyze-game ()
  ( format t ">>> Testing: analyze-game~%" )
  ; a couple of busts
  ( format t "Game 1 ... ~%" )
  ( setf *hand1* '( ( 2 . diamond ) ( 4 . diamond ) ( jack . heart ) ) )
  ( setf *hand2* '( ( 10 . spade ) ( king . heart ) ( queen . heart ) ) )
  ( analyze-game )
  ( format t "*hand1* = ~A~%" ( write-to-string *hand1* ) )
  ( format t "*hand2* = ~A~%" *hand2* )
  ( format t "*game-state* = ~A~%" *game-state* )
  ; an ordinary flush and a straight flush
  ( format t "Game 2 ... ~%" )
  ( setf *hand1* '( ( 10 . diamond ) ( jack . diamond ) ( 2 . diamond ) ) )
  ( setf *hand2* '( ( 3 . spade ) ( 5 . spade ) ( 4 . spade ) ) )
  ( analyze-game )
  ( format t "*hand1* = ~A~%" ( write-to-string *hand1* ) )
  ( format t "*hand2* = ~A~%" *hand2* )
  ( format t "*game-state* = ~A~%" *game-state* )
  nil
)
```

(c) Run the demo one time, in order to assure that the program is working properly. Save the output to a file called `demo_analyze_game.text`. The output from the demo should look something like this:

```
>>> Testing: analyze-game
Game 1 ...
*hand1* = ((2 . DIAMOND) (4 . DIAMOND) (JACK . HEART))
*hand2* = ((10 . SPADE) (KING . HEART) (QUEEN . HEART))
*game-state* = (BUST BUST)
Game 2 ...
*hand1* = ((10 . DIAMOND) (JACK . DIAMOND) (2 . DIAMOND))
*hand2* = ((3 . SPADE) (5 . SPADE) (4 . SPADE))
*game-state* = ((JACK HIGH DIAMOND FLUSH) (5 HIGH STRAIGHT SPADE FLUSH))
NIL
```

2. `report-the-result`

(a) Please type in the following function, `report-the-result`, which reports the result of a game, presumed to come to completion, which reports the result of the game by analyzing the `*game-state*` variable.

```
( defun report-the-result ()
  ( cond
    ( ( equal *game-state* '( bust bust ) )
      ( format t "--> The game is a draw. The deck is dead.~%")
    )
    ( ( and ( not ( equal ( first *game-state* ) 'bust ) )
            ( equal ( second *game-state* ) 'bust )
      )
      ( format t "--> Player 1 wins with ~A~%" ( first *game-state* ) )
    )
    ( ( and ( equal ( first *game-state* ) 'bust )
            ( not ( equal ( second *game-state* ) 'bust ) )
      )
      ( format t "--> Player 2 wins with ~A~%" ( second *game-state* ) )
    )
    ( ( and ( straight-p *hand1* ) ( not ( straight-p *hand2* ) ) )
      ( format t "!!! Both players found their way to a flush~%" )
      ( format t "--> Player 1 wins with ~A~%" ( first *game-state* ) )
    )
    ( ( and ( not ( straight-p *hand1* ) ) ( straight-p *hand2* ) )
      ( format t "!!! Both players found their way to a flush~%" )
      ( format t "--> Player 2 wins with ~A~%" ( second *game-state* ) )
    )
    ( ( card-greater ( high-card *hand1* ) ( high-card *hand2* ) )
      ( format t "!!! Both players found their way to a flush~%" )
      ( format t "--> Player 1 wins with ~A~%" ( first *game-state* ) )
    )
    ( ( card-greater ( high-card *hand2* ) ( high-card *hand1* ) )
      ( format t "!!! Both players found their way to a flush~%" )
      ( format t "--> Player 2 wins with ~A~%" ( second *game-state* ) )
    )
  )
  nil
)
```

(b) Write `card-greater` function that is used in the `report-the-result`.

(c) Write a test program called `demo--report-the-result` which gives the `report-the-result` program a good going over by arranging for it to be run 7 times, one time corresponding to each of the cases in the cond of the program.

(d) Run your `demo--report-the-result` program and look closely at the results. If they are not as they should be, fix the problem.

3. Save your work!

   Save the functions that your wrote for this part of the program, including the demo functions, to the section of your solution document dedicated to this part of the program. Also, save the demos that you generated to the to the section of your solution document dedicated to this part of the program.

Part 6: Play Game

This first task of this part involves mapping the internal represetation of a hand (list of dotted pairs) to a fixed width external representation.

1. The rank mapping: 2 → 2, 3 → 3, 4 → 4, 5 → 5, 6 → 6, 7 → 7, 8 → 8, 9 → 9, 10 → X, JACK → J, QUEEN → Q, KING → K, ACE → A.

2. The suit mapping: CLUB → C, DIAMOND → D, HEART → H, SPADE → S.

3. Examples:

   (a) ((5 . CLUB) (QUEEN . DIAMOND) (10 . SPADE)) → ((5 . C) (Q . D) (X . S))

   (b) ((KING . DIAMOND) (10 . DIAMOND) (5 . DIAMOND)) → ((K . C) (X . D) (5 . S))

The second task of this part consist of implementing the main game playing program, and two crucial support programs, one for making moves, the other for determining when the game is over.

1. `hand-rep`

   (a) Write the function `hand-rep` taking one argument which presents the internal representation of a hand and returns as its value the external fixed witdth representation of the hand.

   (b) Type in the following program and use it to test your `hand-rep` function.

```
( defun demo--hand-rep ( &aux hand )
  ( establish-shuffled-deck )
  ( setf internal ( list ( pop *deck* ) ( pop *deck* ) ( pop *deck* ) ) )
  ( setf external ( hand-rep internal ) )
  ( format t "~A --> ~A~%" internal external )
  nil
)
```

(c) Run the demo a half dozen times, in order to assure that the program is working properly. Save the output to a file called `demo_hand_rep.text`. The output from the demo should look something like this:

```
[]> ( demo--hand-rep )
((8 . DIAMOND) (QUEEN . SPADE) (JACK . DIAMOND)) --> ((8 . D) (Q . S) (J . D))
NIL
[]> ( demo--hand-rep )
((6 . SPADE) (3 . CLUB) (5 . SPADE)) --> ((6 . S) (3 . C) (5 . S))
NIL
[]> ( demo--hand-rep )
((7 . CLUB) (KING . DIAMOND) (6 . CLUB)) --> ((7 . C) (K . D) (6 . C))
NIL
[]> ( demo--hand-rep )
((10 . DIAMOND) (5 . HEART) (8 . SPADE)) --> ((X . D) (5 . H) (8 . S))
NIL
[]> ( demo--hand-rep )
((2 . DIAMOND) (KING . HEART) (2 . HEART)) --> ((2 . D) (K . H) (2 . H))
NIL
[]> ( demo--hand-rep )
( 6 . SPADE) (9 . SPADE) (10 . SPADE)) --> ((6 . S) (9 . S) (X . S))
NIL
```

2. `play-game, make-moves, game-over-p`

(a) Carefully type in the following three function definitions, doing your best to study what they do seperately and collectively:

```
( defun play-game ()
  ( deal-hands )
  ( make-moves )
  ( report-the-result )
)

( defun make-moves ()
  ( format t "~A      ~A~%" ( hand-rep *hand1* ) ( hand-rep *hand2* ) )
  ( if ( not ( game-over-p ) )
    ( let ()
      ( players-each-take-a-turn )
      ( make-moves )
    )
  )
  nil
)

( defun game-over-p ()
  ( analyze-game )
  ( or
    ( not ( equal *game-state* '( bust bust ) ) )
    ( null *deck* )
  )
)
```

(b) Type in the following program and use it to test your hand-rep function.

```
( defun demo--play-game ()
  ( format t ">>> Testing: play-game~%" )
  ( play-game )
)
```

(c) Run the demo a half dozen times, in order to assure that the program is working properly. The output from the demo should look something like this:

```
[]> ( demo--play-game )
>>> Testing: play-game
((5 . C) (6 . H) (3 . C))      ((3 . H) (K . C) (Q . H))
((5 . C) (4 . S) (3 . C))      ((3 . H) (7 . H) (Q . H))
--> Player 2 wins with (QUEEN HIGH HEART FLUSH)
NIL
[]> ( demo--play-game )
>>> Testing: play-game
((9 . D) (3 . H) (3 . C))      ((J . S) (6 . S) (8 . S))
--> Player 2 wins with (JACK HIGH SPADE FLUSH)
NIL
[]> ( demo--play-game )
>>> Testing: play-game
((9 . C) (A . S) (3 . C))      ((2 . H) (K . C) (X . C))
((5 . S) (A . S) (3 . C))      ((4 . S) (K . C) (X . C))
((5 . S) (J . S) (3 . C))      ((J . D) (K . C) (X . C))
((5 . S) (J . S) (Q . D))      ((J . D) (K . C) (2 . D))
((K . D) (J . S) (Q . D))      ((J . D) (X . D) (2 . D))
--> Player 2 wins with (JACK HIGH DIAMOND FLUSH)
NIL
[]> ( demo--play-game )
>>> Testing: play-game
((2 . H) (X . S) (A . S))      ((9 . S) (2 . C) (5 . C))
((2 . H) (K . H) (A . S))      ((9 . S) (2 . C) (K . C))
((2 . H) (6 . C) (A . S))      ((6 . D) (2 . C) (K . C))
((2 . H) (6 . C) (3 . H))      ((6 . D) (8 . D) (K . C))
((X . D) (6 . C) (3 . H))      ((A . H) (8 . D) (K . C))
((X . D) (Q . S) (3 . H))      ((A . H) (2 . S) (K . C))
((X . D) (Q . S) (8 . S))      ((5 . S) (2 . S) (K . C))
((X . D) (X . H) (8 . S))      ((8 . H) (2 . S) (K . C))
((K . D) (X . H) (8 . S))      ((8 . H) (2 . S) (4 . C))
((K . D) (5 . H) (8 . S))      ((4 . H) (2 . S) (4 . C))
((K . D) (5 . H) (J . H))      ((4 . H) (2 . S) (7 . C))
((A . C) (5 . H) (J . H))      ((4 . H) (2 . S) (J . C))
((A . C) (6 . S) (J . H))      ((4 . H) (2 . S) (9 . H))
((3 . D) (6 . S) (J . H))      ((Q . H) (2 . S) (9 . H))
((3 . D) (6 . S) (A . D))      ((Q . H) (3 . C) (9 . H))
((3 . D) (6 . S) (9 . D))      ((Q . H) (3 . C) (5 . D))
((3 . D) (7 . D) (9 . D))      ((Q . D) (3 . C) (5 . D))
```

```
--> Player 1 wins with (9 HIGH DIAMOND FLUSH)
NIL
[]> ( demo--play-game )
>>> Testing: play-game
((K . D) (6 . C) (K . H))      ((K . C) (Q . H) (5 . D))
((3 . S) (6 . C) (K . H))      ((K . C) (Q . H) (A . D))
((3 . S) (6 . C) (2 . C))      ((6 . S) (Q . H) (A . D))
((3 . S) (6 . C) (4 . S))      ((6 . S) (2 . H) (A . D))
((3 . S) (6 . C) (7 . D))      ((6 . S) (3 . H) (A . D))
((X . H) (6 . C) (7 . D))      ((6 . S) (3 . H) (X . C))
((8 . D) (6 . C) (7 . D))      ((6 . S) (3 . H) (5 . S))
((8 . D) (J . D) (7 . D))      ((6 . S) (J . H) (5 . S))
--> Player 1 wins with (JACK HIGH DIAMOND FLUSH)
NIL
[]> ( demo--play-game )
>>> Testing: play-game
((K . H) (Q . C) (5 . S))      ((5 . H) (9 . H) (X . H))
--> Player 2 wins with (10 HIGH HEART FLUSH)
NIL
[]> ( demo--play-game )
>>> Testing: play-game
((6 . S) (2 . S) (5 . S))      ((6 . H) (9 . D) (K . D))
--> Player 1 wins with (6 HIGH SPADE FLUSH)
NIL
[]> ( demo--play-game )
>>> Testing: play-game
((5 . S) (J . H) (4 . D))      ((5 . D) (6 . H) (8 . D))
((5 . S) (J . H) (5 . H))      ((5 . D) (8 . S) (8 . D))
((8 . H) (J . H) (5 . H))      ((J . S) (8 . S) (8 . D))
--> Player 1 wins with (JACK HIGH HEART FLUSH)
NIL
[]> ( demo--play-game )
>>> Testing: play-game
((3 . H) (6 . S) (3 . S))      ((A . H) (7 . S) (J . D))
((3 . H) (J . S) (3 . S))      ((A . H) (7 . S) (9 . D))
((5 . D) (J . S) (3 . S))      ((A . H) (7 . S) (X . S))
((2 . H) (J . S) (3 . S))      ((A . H) (4 . D) (X . S))
((K . C) (J . S) (3 . S))      ((A . H) (4 . D) (7 . C))
((K . C) (5 . H) (3 . S))      ((A . H) (6 . D) (7 . C))
((K . C) (5 . H) (K . D))      ((A . H) (A . D) (7 . C))
((K . C) (5 . H) (8 . C))      ((2 . S) (A . D) (7 . C))
((K . C) (5 . S) (8 . C))      ((2 . S) (A . D) (Q . H))
((K . C) (5 . S) (J . C))      ((6 . C) (A . D) (Q . H))
((K . C) (9 . C) (J . C))      ((6 . C) (A . D) (8 . S))
--> Player 1 wins with (KING HIGH CLUB FLUSH)
NIL
[47]> ( demo--play-game )
>>> Testing: play-game
((7 . C) (X . C) (8 . C))      ((J . S) (9 . H) (J . C))
--> Player 1 wins with (10 HIGH CLUB FLUSH)
NIL
[]>
```

3. Save your work!

Save the functions that your wrote for this part of the program, including the demo functions, to the section of your solution document dedicated to this part of the program. Also, save the demos that you generated to the to the section of your solution document dedicated to this part of the program.

4. Save your work!

   Save the functions that your wrote for this part of the program, including the demo functions, to the section of your solution document dedicated to this part of the program. Also, save the demos that you generated to the to the section of your solution document dedicated to this part of the program.

Part 7: Computing Statistics

This part will incorporate a number of counters in the service of gathering some statistics for the Three Card Flush. These counters will be implemented as global variables. In particular, these are the counters:

- *game-count* - number of games played
- *turn-count* - number of turns taken, where a turn, for the purposes of this statistic, consists of both players discarding a card and accepting another from the dealer
- *win1-count* - number of games won by Player 1
- *win2-count* - number of games won by Player 2
- *draw-count* - number of games that resulted in a draw
- *f1f2-count* - number of games where both players found their way to flush

1. `Counter Initialization, Incrementation, and Infrastructure`

   (a) Please read and study the following code, and then type it into your program.

```
; Counter initialization -- initialize once so the game can be played regardless of
; whether or not statistics are being computed

( defun init-counters ()
  ( setf *win1-count* 0 )
  ( setf *win2-count* 0 )
  ( setf *draw-count* 0 )
  ( setf *turn-count* 0 )
  ( setf *f1f2-count* 0 )
  ( setf *game-count* 0 )
  nil
)

( init-counters )

; Flexible counter incrementation

( defun increment (name)
  ( set name ( + ( eval name ) 1 ) )
  nil
)

; The main statistics computation program

( defun compute-statistics ( n )
  ( init-counters )
  ( play-game-n-times n )
  ( format t "*game-count* = ~A~%" *game-count* )
  ( format t "*turn-count* = ~A~%" *turn-count* )
  ( format t "*win1-count* = ~A~%" *win1-count* )
  ( format t "*win2-count* = ~A~%" *win2-count* )
  ( format t "*draw-count* = ~A~%" *draw-count* )
  ( format t "*f1f2-count* = ~A~%" *f1f2-count* )
  nil
)

; Program to play the game n times

( defun play-game-n-times ( n )
  ( cond
    ( ( > n 0 )
      ( play-game )
      ( play-game-n-times ( - n 1 ) )
    )
  )
)
```

2. Place calls to the `increment` function in just the right places:
   (a) Place `( increment '*game-count* )` in the `play-game` function, as its first line.
   (b) Place `( increment '*turn-count* )` in the `make-moves` function, as its first line.
   (c) Carefully sprinkle forms to increment the remaining counters within the cases of the `cond` in the `report-the-result` method so that the proper counts will be made as the program to compute statistics runs.

3. Run the `compute-statistics` function, giving it 10. Run it again, giving it 100. Run it once more, giving it 1000. Take a good look at the results.

4. Save your work!

   Save the functions that your wrote for this part of the program, including the demo functions, to the section of your solution document dedicated to this part of the program. Also, save the demos that you generated to the to the section of your solution document dedicated to this part of the program.

Part 8: The Heuristic Player

This part of your assignment changes the nature of Player 2. In particular, it transforms Player to from a random mover to a heuristic mover!

1. `players-each-take-a-turn`

   (a) Edit the `players-each-take-a-turn` function so that it looks like this:

   ```
   ( defun players-each-take-a-turn ()
     ( randomly-heuristically-discard-cards )
     ( replace-cards )
     nil
   )
   ```

   (b) Edit the `randomly-heuristically-discard-cards` function so that it looks like this:

   ```
   ( defun randomly-heuristically-discard-cards ()
     ( random-discard-card-from-hand1 )
     ( heuristic-discard-card-from-hand2 )
     nil
   )
   ```

   (c) Define the `heuristic-discard-card-from-hand2` function so that operates using one or more "if-situation-then-action" rules to make a move.

   (d) Type in the following test program for the `randomly-heuristically-discard-cards` function:

   ```
   ( defun demo--randomly-heuristically-discard-cards ()
     ( format t ">>> Testing: randomly-discard-cards~%" )
     ( deal-hands )
     ( format t "--- *hand1* = ~A~%" *hand1* )
     ( format t "--- *hand2* = ~A~%" *hand2* )
     ( randomly-heuristically-discard-cards )
     ( format t "--- *hand1* = ~A~%" *hand1* )
     ( format t "--- *hand2* = ~A~%" *hand2* )
     nil
   )
   ```

   (e) Run the `demo--randomly-heuristically-discard-cards` function enough times to get a feel for whether or not your `randomly-heuristically-discard-cards` function is working reasonably. Save the output to a file called `demo_randomly_heuristically_discard_cards.text`.

   (f) Run the `play-game` function enough times to get a feel for whether or not your `play-game` function is working reasonably.

2. Checking the statistics for Random vs Heuristic

   Run the `compute-statistics` function, giving it 10. Run it again, giving it 100. Run it once more, giving it 1000. Take a good look at the results.

3. Save your work!

Save the functions that your wrote for this part of the program, including the demo functions, to the section of your solution document dedicated to this part of the program. Also, save the demos that you generated to the to the section of your solution document dedicated to this part of the program.

Due Date

Friday, October 14, 2022