Kuncheng Feng

CSC 416

Solution document for :-

Programming Challenge: Three Card Flush

Part 1: The Deck and the Shuffle

1-1 Demo for make-deck

```
[]> (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
```

1-1 Code for make-deck

```
(defun make-deck ()
      (mapcan (function make-rank-suit) '(club diamond heart spade))
)

(defun make-rank-suit (suit &aux ranks suit-duplicates)
      (setf ranks '(2 3 4 5 6 7 8 9 10 jack queen king ace))
```

```
      (setf suit-duplicates (duplicate (length ranks) suit))
      (mapcar (function cons) ranks suit-duplicates)
)

(defun demo--make-deck (&aux 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
)
```

1-2 Demo for establish-shuffled-deck

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

1-2 Code for shuffle-deck

```
(setf *deck* (list))

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

(defun shuffle (deck &aux card)
      (cond
            ((null deck)
```

```
                deck
            )
            (t
                (setf card (pick deck))
                (setf deck (remove card deck :count 1))
                (cons card (shuffle deck))
            )
        )
)

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

Part 2: The Hands, the Deal and the Discard

2-1 Demo for deal-hands

```
[]> (demo--deal-hands)
>>> Testing: deal-hands
--- *hand1* = ((JACK . DIAMOND) (8 . SPADE) (4 . CLUB))
--- *hand2* = ((6 . HEART) (3 . DIAMOND) (7 . SPADE))
--- number of cards in *deck* = 46
NIL
[]> (demo--deal-hands)
>>> Testing: deal-hands
--- *hand1* = ((6 . DIAMOND) (8 . DIAMOND) (9 . DIAMOND))
--- *hand2* = ((QUEEN . DIAMOND) (3 . SPADE) (10 . SPADE))
--- number of cards in *deck* = 46
NIL
```

2-1 Code for deal-hands

```
; Global variables
(setf *hand1* (list))
(setf *hand2* (list))
```

```
(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
)

(defun deal-card-to-hand1 (&aux card)
      (setf card (car *deck*))
      (setf *deck* (cdr *deck*))
      (setf *hand1* (snoc card *hand1*))
)

(defun deal-card-to-hand2 ()
      (setf card (car *deck*))
      (setf *deck* (cdr *deck*))
      (setf *hand2* (snoc card *hand2*))
)

(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
)
```

2-2 Demo for randomly-discard-cards

```
[]> (demo--randomly-discard-cards)
>>> Testing: randomly-discard-cards
Before:
--- *hand1* = ((8 . SPADE) (2 . CLUB) (KING . HEART))
--- *hand2* = ((5 . HEART) (QUEEN . CLUB) (3 . SPADE))
```

```
After:
--- *hand1* = ((8 . SPADE) NIL (KING . HEART))
--- *hand2* = ((5 . HEART) (QUEEN . CLUB) NIL)
NIL
[]> (demo--randomly-discard-cards)
>>> Testing: randomly-discard-cards
Before:
--- *hand1* = ((3 . HEART) (6 . SPADE) (7 . CLUB))
--- *hand2* = ((KING . CLUB) (QUEEN . HEART) (6 . CLUB))
After:
--- *hand1* = ((3 . HEART) NIL (7 . CLUB))
--- *hand2* = ((KING . CLUB) (QUEEN . HEART) NIL)
NIL
[]> (demo--randomly-discard-cards)
>>> Testing: randomly-discard-cards
Before:
--- *hand1* = ((KING . CLUB) (6 . CLUB) (5 . CLUB))
--- *hand2* = ((ACE . DIAMOND) (5 . DIAMOND) (6 . HEART))
After:
--- *hand1* = ((KING . CLUB) NIL (5 . CLUB))
--- *hand2* = (NIL (5 . DIAMOND) (6 . HEART))
NIL
```

2-2 Code for randomly-discard-cards

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

(defun randomly-discard-card-from-hand1 (&aux number)
      (setf number (random (length *hand1*)))
      (setf (nth number *hand1*) (list))
)

(defun randomly-discard-card-from-hand2 (&aux number)
      (setf number (random (length *hand2*)))
      (setf (nth number *hand2*) (list))
)

(defun demo--randomly-discard-cards ()
      (format t ">>> Testing: randomly-discard-cards~%")
      (deal-hands)
```

```
      (format t "Before:~%")
      (format t "--- *hand1* = ~A~%" *hand1*)
      (format t "--- *hand2* = ~A~%" *hand2*)
      (randomly-discard-cards)
      (format t "After:~%")
      (format t "--- *hand1* = ~A~%" *hand1*)
      (format t "--- *hand2* = ~A~%" *hand2*)
      nil
)
```

Part 3: Replacing Cards in Hands, Taking Turns

3-1 Demo for replace-cards

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

```
--- *hand2* = ((6 . HEART) (10 . DIAMOND) (10 . HEART))
Cards randomly discarded:
--- *hand1* = (NIL (ACE . HEART) (5 . HEART))
--- *hand2* = (NIL (10 . DIAMOND) (10 . HEART))
Cards replaced:
--- *hand1* = ((3 . HEART) (ACE . HEART) (5 . HEART))
--- *hand2* = ((ACE . SPADE) (10 . DIAMOND) (10 . HEART))
NIL
```

3-1 Code for replace-cards

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

(defun replace-card-in-hand1 (&aux newCard)
      (setf newCard (car *deck*))
      (setf *deck* (cdr *deck*))
      (setf (nth (position nil *hand1*) *hand1*) newCard)
)

(defun replace-card-in-hand2 (&aux newCard)
      (setf newCard (car *deck*))
      (setf *deck* (cdr *deck*))
      (setf (nth (position nil *hand2*) *hand2*) newCard)
)

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

```
      nil
)
```

3-2 Demo for players-each-take-a-turn

```
[]> (demo--players-each-take-a-turn)
>>> Testing: players-each-take-a-turn
--- The hands ...
--- *hand1* = ((10 . CLUB) (KING . CLUB) (QUEEN . DIAMOND))
--- *hand2* = ((ACE . DIAMOND) (JACK . DIAMOND) (10 . DIAMOND))
--- Each player takes a turn ...
--- *hand1* = ((10 . CLUB) (KING . CLUB) (8 . DIAMOND))
--- *hand2* = ((2 . SPADE) (JACK . DIAMOND) (10 . DIAMOND))
--- Each player takes a turn ...
--- *hand1* = ((10 . CLUB) (9 . HEART) (8 . DIAMOND))
--- *hand2* = ((2 . HEART) (JACK . DIAMOND) (10 . DIAMOND))
--- Each player takes a turn ...
--- *hand1* = ((7 . DIAMOND) (9 . HEART) (8 . DIAMOND))
--- *hand2* = ((2 . HEART) (3 . DIAMOND) (10 . DIAMOND))
--- Each player takes a turn ...
--- *hand1* = ((7 . DIAMOND) (9 . HEART) (9 . DIAMOND))
--- *hand2* = ((10 . HEART) (3 . DIAMOND) (10 . DIAMOND))
NIL
```

3-2 Code for players-each-take-a-turn

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

(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
)
```

Part 4: Hand Analysis

4-1 Demo for flush-p

```
[]> (demo--flush-p)
>>> 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
```

4-1 Code for flush-p

```
(defun check-uniform (l)
      (if (<= (length l) 1)
            t
            (and
                  (equal (first l) (second l))
                  (check-uniform (rest l))
            )
      )
)

(defun flush-p (hand &aux suits suit)
      (setf suits (mapcar (function cdr) hand))
      (check-uniform suits)
```

```
)

( 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~%" )
      )
)
```

4-2 Demo for high-card

```
[]> (demo--high-card)
>>> 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
```

4-2 Code for high-card

```
; Set as global variable in case of future use
(setf *ranks* '(2 3 4 5 6 7 8 9 10 jack queen king ace))
(setf *suits* '(club diamond heart spade))

(defun high-card (hand)
       (highest-card (car hand) (cdr hand))
)

(defun highest-card (high hand &aux this)
       (setf this (car hand))
       (cond
              ((equal nil this)
                     high
              )
              ((> (position (car high) *ranks*) (position (car this) *ranks*))
                     (highest-card high (cdr hand))
              )
              ((< (position (car high) *ranks*) (position (car this) *ranks*))
                     (highest-card this (cdr hand))
              )
              ((> (position (cdr high) *suits*) (position (cdr this) *suits*))
                     (highest-card high (cdr hand))
              )
              (t
                     (highest-card this (cdr hand))
              )
       )
)

(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
)
```

4-3 Demo for straight-p

```
[]> (demo--straight-p)
>>> 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
```

4-3 Code for straight-p

```
(defun straight-p (hand &aux hand-ranks)
      (setf hand-ranks (mapcar #'car hand))
      (setf hand-ranks (mapcar #'rank-to-number hand-ranks))
      (setf hand-ranks (sort hand-ranks '<))
      (check-increment hand-ranks)
)

(defun rank-to-number (rank)
      (position rank *ranks*)
)

(defun check-increment (hand-ranks)
      (if (<= (length hand-ranks) 1)
            t
            (and
                  (equal (+ (first hand-ranks) 1) (second hand-ranks))
                  (check-increment (rest hand-ranks))
            )
      )
)

(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
)
```

4-4 Demo for analyze-hand

```
[]> (demo--analyze-hand)
>>> 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
```

4-4 Code for analyze-hand

```
(defun analyze-hand (hand &aux high rank suit)
	(cond
		((not (flush-p hand))
			'bust
		)
		(t
			(setf high (high-card hand))
			(setf rank (car high))
			(setf suit (cdr high))
			(if (straight-p hand)
				(setf result (list rank 'high 'straight suit 'flush))
				(setf result (list rank 'high suit 'flush))
			)
		)
	)
```

```
)

(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
)
```

Part 5: Game State and End of Game Reporting

5-1 Demo for analyze-game

```
[]> (demo--analyze-game)
>>> 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
```

5-1 Code for analyze-game

```
; More global variables
(setf *game-state* (list))
(defun analyze-game ()
      (setf *game-state* (list (analyze-hand *hand1*) (analyze-hand
*hand2*)))
)

(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
)
```

5-2 Demo for report-the-result

```
[]> (demo--report-the-result)
>>> Testing: report-the-result

Testing: (bust bust)
*hand1*: ((2 . CLUB) (3 . DIAMOND) (4 . HEART))
*hand2*: ((2 . DIAMOND) (3 . HEART) (4 . SPADE))
Game state: (BUST BUST)
--> The game is a draw. The deck is dead.

Testing: (not-bust bust)
*hand1*: ((2 . CLUB) (4 . CLUB) (6 . CLUB))
*hand2*: ((2 . DIAMOND) (3 . HEART) (4 . SPADE))
Game state: ((6 HIGH CLUB FLUSH) BUST)
--> Player 1 wins with (6 HIGH CLUB FLUSH)

Testing: (bust not-bust)
*hand1*: ((2 . CLUB) (4 . DIAMOND) (6 . HEART))
*hand2*: ((2 . DIAMOND) (3 . DIAMOND) (4 . DIAMOND))
Game state: (BUST (4 HIGH STRAIGHT DIAMOND FLUSH))
--> Player 2 wins with (4 HIGH STRAIGHT DIAMOND FLUSH)

Testing: (straight not-straight)
```

```
*hand1*: ((2 . CLUB) (3 . CLUB) (4 . CLUB))
*hand2*: ((2 . DIAMOND) (4 . DIAMOND) (5 . DIAMOND))
Game state: ((4 HIGH STRAIGHT CLUB FLUSH) (5 HIGH DIAMOND FLUSH))
!!! Both players found their way to a flush
--> Player 1 wins with (4 HIGH STRAIGHT CLUB FLUSH)

Testing: (not-straight straight)
*hand1*: ((2 . CLUB) (4 . CLUB) (6 . CLUB))
*hand2*: ((2 . DIAMOND) (3 . DIAMOND) (4 . DIAMOND))
Game state: ((6 HIGH CLUB FLUSH) (4 HIGH STRAIGHT DIAMOND FLUSH))
!!! Both players found their way to a flush
--> Player 2 wins with (4 HIGH STRAIGHT DIAMOND FLUSH)

Testing: (greater-flush flush)
*hand1*: ((2 . CLUB) (4 . CLUB) (6 . CLUB))
*hand2*: ((2 . DIAMOND) (3 . DIAMOND) (5 . DIAMOND))
Game state: ((6 HIGH CLUB FLUSH) (5 HIGH DIAMOND FLUSH))
!!! Both players found their way to a flush
--> Player 1 wins with (6 HIGH CLUB FLUSH)

Testing: (flush greater-flush)
*hand1*: ((2 . CLUB) (4 . CLUB) (6 . CLUB))
*hand2*: ((2 . DIAMOND) (3 . DIAMOND) (6 . DIAMOND))
Game state: ((6 HIGH CLUB FLUSH) (6 HIGH DIAMOND FLUSH))
!!! Both players found their way to a flush
--> Player 2 wins with (6 HIGH DIAMOND FLUSH)
NIL
```

5-2 Code for report-the-result

```
(defun report-the-result ()
       (cond
              ((equal *game-state* '(bust bust))
                     (increment '*draw-count*)
                     (format t "--> The game is a draw. The deck is dead.~%")
              )
              ((and
                     (not ( equal (first *game-state*) 'bust))
                     (equal (second *game-state*) 'bust))
                            (increment '*win1-count*)
                            (format t "--> Player 1 wins with ~A~%" (first *game-state*))
              )
              ((and
                     (equal (first *game-state*) 'bust)
                     (not (equal (second *game-state*) 'bust)))
                            (increment '*win2-count*)
                            (format t "--> Player 2 wins with ~A~%" (second *game-state*))
              )
              ((and
```

```
                        (straight-p *hand1*)
                        (not (straight-p *hand2*)))
                                (increment '*win1-count*)
                                (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*))
                                (increment '*win2-count*)
                                (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*))
                        (increment '*f1f2-count*)
                        (increment '*win1-count*)
                        ( 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*))
                        (increment '*f1f2-count*)
                        (increment '*win2-count*)
                        (format t "!!! Both players found their way to a flush~%")
                        (format t "--> Player 2 wins with ~A~%" (second *game-state*))
                )
        )
        nil
)

; Returns true is thisCard is greater than otherCard
(defun card-greater (thisCard otherCard)
        (if (equal (car thisCard) (car otherCard))
                (> (position (cdr thisCard) *suits*) (position (cdr otherCard) *suits*))
                (> (position (car thisCard) *ranks*) (position (car otherCard) *ranks*))
        )
)

(defun demo--report-the-result ()
        (format t ">>> Testing: report-the-result ~%")

        (format t "~%Testing: (bust bust) ~%")
        (setf *hand1* '((2 . club) (3 . diamond) (4 . heart)))
        (setf *hand2* '((2 . diamond) (3 . heart) (4 . spade)))
        (analyze-game)
        (format t "*hand1*: ~A~%" *hand1*)
        (format t "*hand2*: ~A~%" *hand2*)
        (format t "Game state: ~A ~%" *game-state*)
        (report-the-result)

        (format t "~%Testing: (not-bust bust) ~%")
        (setf *hand1* '((2 . club) (4 . club) (6 . club)))
        (setf *hand2* '((2 . diamond) (3 . heart) (4 . spade)))
        (analyze-game)
        (format t "*hand1*: ~A~%" *hand1*)
        (format t "*hand2*: ~A~%" *hand2*)
        (format t "Game state: ~A ~%" *game-state*)
        (report-the-result)

        (format t "~%Testing: (bust not-bust) ~%")
        (setf *hand1* '((2 . club) (4 . diamond) (6 . heart)))
        (setf *hand2* '((2 . diamond) (3 . diamond) (4 . diamond)))
        (analyze-game)
```

```
        (format t "*hand1*: ~A~%" *hand1*)
        (format t "*hand2*: ~A~%" *hand2*)
        (format t "Game state: ~A ~%" *game-state*)
        (report-the-result)

        (format t "~%Testing: (straight not-straight) ~%")
        (setf *hand1* '((2 . club) (3 . club) (4 . club)))
        (setf *hand2* '((2 . diamond) (4 . diamond) (5 . diamond)))
        (analyze-game)
        (format t "*hand1*: ~A~%" *hand1*)
        (format t "*hand2*: ~A~%" *hand2*)
        (format t "Game state: ~A ~%" *game-state*)
        (report-the-result)

        (format t "~%Testing: (not-straight straight) ~%")
        (setf *hand1* '((2 . club) (4 . club) (6 . club)))
        (setf *hand2* '((2 . diamond) (3 . diamond) (4 . diamond)))
        (analyze-game)
        (format t "*hand1*: ~A~%" *hand1*)
        (format t "*hand2*: ~A~%" *hand2*)
        (format t "Game state: ~A ~%" *game-state*)
        (report-the-result)

        (format t "~%Testing: (greater-flush flush) ~%")
        (setf *hand1* '((2 . club) (4 . club) (6 . club)))
        (setf *hand2* '((2 . diamond) (3 . diamond) (5 . diamond)))
        (analyze-game)
        (format t "*hand1*: ~A~%" *hand1*)
        (format t "*hand2*: ~A~%" *hand2*)
        (format t "Game state: ~A ~%" *game-state*)
        (report-the-result)

        (format t "~%Testing: (flush greater-flush) ~%")
        (setf *hand1* '((2 . club) (4 . club) (6 . club)))
        (setf *hand2* '((2 . diamond) (3 . diamond) (6 . diamond)))
        (analyze-game)
        (format t "*hand1*: ~A~%" *hand1*)
        (format t "*hand2*: ~A~%" *hand2*)
        (format t "Game state: ~A ~%" *game-state*)
        (report-the-result)
)
```

Part 6: Play Game

6-1 Demo for hand-rep

```
[]> (demo--hand-rep)
((6 . CLUB) (3 . HEART) (10 . SPADE)) --> ((6 . C) (3 . H) (X . S))
NIL
[]> (demo--hand-rep)
((ACE . DIAMOND) (4 . CLUB) (2 . CLUB)) --> ((A . D) (4 . C) (2 .
C))
NIL
[]> (demo--hand-rep)
```

```
((7 . HEART) (5 . DIAMOND) (4 . HEART)) --> ((7 . H) (5 . D) (4 .
H))
NIL
[]> (demo--hand-rep)
((JACK . SPADE) (9 . HEART) (8 . HEART)) --> ((J . S) (9 . H) (8 .
H))
NIL
[]> (demo--hand-rep)
((5 . DIAMOND) (ACE . CLUB) (8 . DIAMOND)) --> ((5 . D) (A . C) (8
. D))
NIL
[]> (demo--hand-rep)
((KING . HEART) (ACE . DIAMOND) (QUEEN . DIAMOND)) --> ((K . H) (A
. D) (Q . D))
```

6-1 Code for hand-rep

```
(defun map-rank (rank)
	(cond
		((equal rank 10)
			'x)
		((equal rank 'jack)
			'j)
		((equal rank 'queen)
			'q)
		((equal rank 'king)
			'k)
		((equal rank 'ace)
			'a)
		(t
			rank)
	)
)

(defun map-suit (suit)
	(cond
		((equal suit 'club)
			'c)
		((equal suit 'diamond)
			'd)
		((equal suit 'heart)
			'h)
		((equal suit 'spade)
			's)
		(t
			suit)
	)
)

(defun map-card (card)
	(cons (map-rank (car card)) (map-suit (cdr card)))
)
```

```
(defun hand-rep (hand)
       (mapcar #'map-card hand)
)

(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
)
```

6-2 Demo for play-game, make-moves, game-over-p

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

[]> (demo--play-game)
>>> Testing: play-game
((7 . C) (9 . S) (Q . D))     ((5 . S) (J . S) (Q . H))
((7 . C) (2 . D) (Q . D))     ((K . D) (J . S) (Q . H))
((A . C) (2 . D) (Q . D))     ((K . D) (J . S) (6 . S))
((6 . C) (2 . D) (Q . D))     ((Q . C) (J . S) (6 . S))
```

```
((8 . D) (2 . D) (Q . D))     ((Q . C) (J . S) (6 . H))
--> Player 1 wins with (QUEEN HIGH DIAMOND FLUSH)
NIL

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

6-2 Code for play-game, make-moves, game-over-p

```
(defun play-game ()
	(increment '*game-count*)
	(deal-hands)
	(make-moves)
	(report-the-result)
)

(defun make-moves ()
	(increment '*turn-count*)
	(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*))
)

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

Part 7: Computing Statistics

Demo

```
[]> (compute-statistics 10)
……
*game-count* = 10
*turn-count* = 140
*win1-count* = 3
*win2-count* = 5
*draw-count* = 2
*f1f2-count* = 0
NIL

[]> (compute-statistics 100)
……
*game-count* = 100
*turn-count* = 1012
*win1-count* = 46
*win2-count* = 42
*draw-count* = 12
*f1f2-count* = 1
NIL

[]> (compute-statistics 1000)
……
*game-count* = 1000
*turn-count* = 10218
*win1-count* = 453
*win2-count* = 444
*draw-count* = 103
*f1f2-count* = 21
NIL
```

Code

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

(init-counters)
```

```
; Flexible counter incrementation
(defun increment (name)
      (set name (+ (eval name) 1))
)

; 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)))
      )
)
```

Part 8: The Heuristic Player

Demo

```
[]> (compute-statistics 10)
*game-count* = 10
*turn-count* = 61
*win1-count* = 2
*win2-count* = 8
*draw-count* = 0
*f1f2-count* = 0
NIL

[]> (compute-statistics 100)
```

```
*game-count* = 100
*turn-count* = 491
*win1-count* = 18
*win2-count* = 82
*draw-count* = 0
*f1f2-count* = 2
NIL

[]> (compute-statistics 1000)
*game-count* = 1000
*turn-count* = 4907
*win1-count* = 208
*win2-count* = 792
*draw-count* = 0
*f1f2-count* = 33
NIL
```

Code

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

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

; 2). heuristic-discard-card-from-hand2
; I would first try to discard the card with the lone suit
; If all three cards have different suit, I would discard card with
smallest number
; It is assumes that hand 2 will always have three cards, otherwise
this won't work.
(defun heuristic-discard-card-from-hand2 (&aux rank1 rank2 rank3
suit1 suit2 suit3)
      (setf rank1 (position (car (first *hand2*)) *ranks*))
      (setf suit1 (cdr (first *hand2*)))

      (setf rank2 (position (car (second *hand2*)) *ranks*))
```

```
      (setf suit2 (cdr (second *hand2*)))

      (setf rank3 (position (car (third *hand2*)) *ranks*))
      (setf suit3 (cdr (third *hand2*)))

      (cond
            ; (same same different), discard the thrid card
            ((and
                  (equal suit1 suit2)
                  (not (equal suit1 suit3)))
                        (setf (third *hand2*) nil)
            )
            ; (same different same), discard the second card
            ((and
                  (equal suit1 suit3)
                  (not (equal suit1 suit2)))
                        (setf (second *hand2*) nil)
            )
            ; (different same same), discard the first card
            ((and
                  (equal suit2 suit3)
                  (not (equal suit1 suit2)))
                        (setf (first *hand2*) nil)
            )
            ; (lowest higer higer), discard the first card
            ((and
                  (<= rank1 rank2)
                  (<= rank1 rank3))
                        (setf (first *hand2*) nil)
            )
            ; (higer lowest higher), discard the second card
            ((and
                  (<= rank2 rank1)
                  (<= rank2 rank3))
                        (setf (second *hand2*) nil)
            )
            ; (higher higher lowest), discard the third card
            ((and
                  (<= rank3 rank1)
                  (<= rank3 rank2))
                        (setf (third *hand2*) nil)
            )
      )
      nil
)
```

```
(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
)
```