Problem 2

```
(define (stop-at n)
  (lambda (my-hand opponent-up-card)
    (< (hand-total my-hand) n)))
```

Problem 3

```
(define (test-strategy player-strategy house-strategy games)
  (if (= games 0)
      0
      (+ (twenty-one player-strategy house-strategy)
         (test-strategy player-strategy house-strategy (- games 1)))))
```

Problem 4

```
(define (watch-player strategy)
  (lambda (my-hand opponent-up-card)
    (newline)
    (display "Opponent up card: ")
    (display opponent-up-card)
    (newline)
    (display "Total: ")
    (display (hand-total my-hand))
    (newline)
    (let ((pass? (strategy my-hand opponent-up-card)))
      (display "Strategy: ")
      (display (if pass? "Hit." "Stay."))
      (newline)
      pass?)))
```

Evaluating

```
(test-strategy (watch-player (stop-at 16))
               (watch-player (stop-at 15))
               2)
```

gives us

```
Opponent up card: 2
Total: 4
Strategy: Hit.

Opponent up card: 2
Total: 11
Strategy: Hit.

Opponent up card: 2
Total: 18
Strategy: Stay.

Opponent up card: 4
Total: 2
Strategy: Hit.

Opponent up card: 4
Total: 8
Strategy: Hit.
```

```
Opponent up card: 4
Total: 9
Strategy: Hit.

Opponent up card: 4
Total: 17
Strategy: Stay.

Opponent up card: 9
Total: 6
Strategy: Hit.

Opponent up card: 9
Total: 8
Strategy: Hit.

Opponent up card: 9
Total: 12
Strategy: Hit.

Opponent up card: 9
Total: 19
Strategy: Stay.

Opponent up card: 6
Total: 9
Strategy: Hit.

Opponent up card: 6
Total: 13
Strategy: Hit.

Opponent up card: 6
Total: 17
Strategy: Stay.
;Value: 2
```

Problem 5

`louis` by direct translation from his description.

```
(define (louis my-hand opponent-up-card)
  (cond ((< (hand-total my-hand) 12) true)
        ((> (hand-total my-hand) 16) false)
        ((and (= (hand-total my-hand) 12) (< opponent-up-card 4)) true)
        ((and (= (hand-total my-hand) 16) (= opponent-up-card 10)) false)
        ((> opponent-up-card 6) true)
        (else false)))
```

Problem 6

```
(define (both strategy1 strategy2)
  (lambda (my-hand opponent-up-card)
    (and (strategy1 my-hand opponent-up-card)
         (strategy2 my-hand opponent-up-card))))
```

Tutorial exercise 1

Earlier they mentioned `cons`, `car`, and `cdr` as being black boxes. I assume the reader is meant to understand them at this point, including lists.

```
(define (make-card rank suit)
  (cons rank suit))
(define (card-rank card)
  (car card))
(define (card-suit card)
  (cdr card))

(define (make-hand up-card card-set)
  (cons card-set up-card))
(define (make-new-hand first-card)
  (make-hand first-card (list first-card)))
(define (hand-up-card hand)
  (cdr hand))
(define (hand-card-set hand)
  (car hand))

(define (hand-total hand)
  (define (count cards)
    (if (null? cards)
        0
        (+ (card-rank (car cards))
           (count (cdr cards)))))
  (count (hand-card-set hand)))
(define (hand-add-card hand new-card)
  (make-hand (hand-up-card hand) (cons new-card (hand-card-set hand))))

(define (deal)
  (make-card (+ 1 (random 10))
             (+ 1 (random 4))))
```

We wish to avoid changing `twenty-one` and `play-hand`. So we make a small change to strategies. For `louis` we must get the card rank of the opponent's up card.

```
(define (louis my-hand opponent-up-card)
  (cond ((< (hand-total my-hand) 12) true)
        ((> (hand-total my-hand) 16) false)
        ((and (= (hand-total my-hand) 12) (< (card-rank opponent-up-card) 4)) true)
        ((and (= (hand-total my-hand) 16) (= (card-rank opponent-up-card) 10)) false)
        ((> (card-rank opponent-up-card) 6) true)
        (else false)))
```

Tutorial exercise 2

`generate-deck` without face cards.

```
(define (generate-deck)
  (define (gen-deck rank suit)
    (cond ((> rank 10) '())
          ((= suit 4) (cons (make-card rank suit)
                            (gen-deck (+ rank 1) 1)))
          (else (cons (make-card rank suit)
                      (gen-deck rank (+ suit 1))))))
  (gen-deck 1 1))
```

`generate-deck` with face cards. To not change `gen-deck` I call it multiple times, appending the values.

```
(define (generate-deck)
  (define (gen-deck rank suit)
    (cond ((> rank 10) '())
          ((= suit 4) (cons (make-card rank suit)
                            (gen-deck (+ rank 1) 1)))
          (else (cons (make-card rank suit)
                      (gen-deck rank (+ suit 1))))))
  (append (gen-deck 1 1)                              ; 1 through 10
          (append (gen-deck 10 1)                     ; jacks
                  (append (gen-deck 10 1)             ; queens
                          (append (gen-deck 10 1))))) ; kings
```

I would rather not split lists like this. If we were to include randomness at this point I would have `cut` form a list containing every other element. Then I would have two halves, one with all odd-positioned elements and one with all even-positioned elements. Without randomness, `alternate-choose` would reconstruct the sorted deck.

```
(define first-card car)
(define rest-cards cdr)
(define empty-deck? null?)

(define (shuffle deck)
  (define (cut2 deck n)
    (if (= n 0)
        deck
        (cut2 (rest-cards deck) (- n 1))))
  (define (cut1 deck n)
    (if (= n 0)
        '()
        (cons (first-card deck)
              (cut1 (rest-cards deck) (- n 1)))))
  (define (alternate-choose half1 half2 n)
    (cond ((empty-deck? half1) half2)
          ((empty-deck? half2) half1)
          ((= n 0) (cons (first-card half1)
                         (alternate-choose (rest-cards half1)
                                           half2
                                           1)))
          (else (cons (first-card half2)
                      (alternate-choose half1
                                        (rest-cards half2)
                                        0)))))
  (alternate-choose (cut1 deck 26) (cut2 deck 26) 0))
```

I chose to not change the formal parameters of `alternate-choose`. Instead I changed the meaning of the procedure. Awkwardly, the `n` parameter does not mean anything on the initial call, besides to not choose any cards. I could have written an internal procedure within `alternate-choose` to remove it. To keep things simple I swap between `half1` and `half2` rather than choose which half to pick from. That is, I choose from whichever half `half1` is. I also did not implement the change I wanted in `shuffle`.

```
(define (random-shuffle deck)
  (define (cut2 deck n)
```

```
  (if (= n 0)
      deck
      (cut2 (cdr deck) (- n 1))))
(define (cut1 deck n)
  (if (= n 0)
      '()
      (cons (first-card deck)
            (cut1 (cdr deck) (- n 1)))))
(define (alternate-choose half1 half2 n)
  (cond ((empty-deck? half1) half2)
        ((empty-deck? half2) half1)
        ((> n 0) (cons (first-card half1)
                       (alternate-choose (rest-cards half1)
                                         half2
                                         (- n 1))))
        (else (let ((cards-to-pick (+ 1 (random 5))))
                (alternate-choose half2 half1 cards-to-pick)))))
(alternate-choose (cut1 deck 26) (cut2 deck 26) 0))
```

For the last part (d), we could completely change `play-hand` and `twenty-one`. That seems poor. I would instead modify `deal` to mutate a deck. It is intuitive to me to use assignment for this part. However at this point I assume we do not know about `set!`.