MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Sample Programming Assignment

The Game of Twenty-one

Louis Reasoner took a course on game theory and became interested in the card game Twenty-One (also called Blackjack). Louis was also treasurer of his living group. By the end of the semester, he had managed to squander the term's dinner money at Atlantic City casinos in an attempt to perfect his "no-lose" strategy. Ben Bitdiddle, Louis's roommate, has decided to construct a general-purpose Twenty-One simulator to help discover what Louis has been doing wrong (in terms of playing Twenty-One, not in terms of moral or ethical conduct).

For our purposes, the rules of Twenty-One are as follows. There are two players, and the object of the game is to be dealt a set of cards that totals as close to 21 as possible without going over 21, where each number card as that number as its values, and face cards have a value of 10. Each player is dealt one card face up that the other player can see. Subsequent cards are dealt face down. One player plays first, asking for more cards one at a time (called a "hit") until he decides to "stay" with the total he has or until his total exceeds 21. If a player's total exceeds 21 he "busts", meaning he immediately loses the game. If the first player does not bust, the second player, called the *house*, then plays, asking for more cards until either losing by exceeding 21 or deciding to stay with the current total. After the house decides not to take additional cards both players expose their cards, and the player with the largest total wins. In the event of a tie, the house wins. This version of Twenty-One is simplified: we will not consider such things as "splitting" or the special treatment of aces. In fact, since we are interested only in the relative strengths of competing strategies, we will not simulate betting either.

A player's *strategy* determines when he wishes another card and when he would like to stay with what he has. For our Scheme simulation, each player will be modeled by a procedure that implements his strategy. Since a typical strategy for when to stay and when to hit involves both the player's current hand and the point value of the opponent's face-up card, we will represent a strategy as a procedure of two arguments: the player's hand and the point value of the opponent's face-up card. The procedure returns true if the player would want another card, and false if the player would stay. For example, the following (stupid) strategy procedure will always take a card if the opponent's up card is greater than 5:

```
(define (stupid-strategy my-hand opponent-up-card)
  (> opponent-up-card 5))
```

The following procedure `play-hand` takes as arguments a strategy, a hand, and the opponent's up card. It continues to accept cards for as long as the strategy requests, or until the total of the cards in the hand exceeds 21. `Play-hand` returns as a value, the full hand that was dealt.

```
(define (play-hand strategy my-hand opponent-up-card)
  (cond ((> (hand-total my-hand) 21) my-hand) ; I lose... give up
        ((strategy my-hand opponent-up-card) ; hit?
         (play-hand strategy
                    (hand-add-card my-hand (deal))
                    opponent-up-card))
        (else my-hand)))               ; stay
```

For the purposes of this simple simulation, a "hand" of cards will be represented by two numbers—the value of the up card and the total of all the cards in the hand.[1] We will represent this using a very simple form of *data abstraction*, of the kind that will be discussed in lecture on September 25—we have a *constructor* procedure `make-hand` that creates a hand from two numbers, and two *selectors* `hand-up-card` and `hand-total` that return the up card and total of a given hand:

```
(define (make-hand up-card total)
  (cons up-card total))

(define (hand-up-card hand)
  (car hand))

(define (hand-total hand)
  (cdr hand))
```

Don't worry right now about the details of the Scheme primitives `cons`, `car`, and `cdr`, which are used in implementing these procedures—we will be discussing this topic in detail over the next few weeks. For now, consider `make-hand`, `hand-up-card`, and `hand-total` to be simple "black boxes" that allow us to represent hands.

In terms of these basic procedures, we can implement some useful operations on hands. `Make-new-hand` takes as argument a first card and returns a hand containing only that card (i.e., the card is both the up card and the total):

```
(define (make-new-hand first-card)
  (make-hand first-card first-card))
```

`Hand-add-card` takes a hand and a new card and returns a hand with the same up-card as the original, but with the total augmented by the value of the new card:

```
(define (hand-add-card hand new-card)
  (make-hand (hand-up-card hand)
             (+ new-card (hand-total hand))))
```

Instead of modeling a real deck of cards, we simply deal cards at random from an infinite deck in which each card value from 1 to 10 is equally probable. (This does not, of course, correctly model real decks of cards, but since our focus is one strategies, we won't worry about the difference.) We represent dealing a card as simply returning a random number in the range 1 through 10:

```
(define (deal) (+ 1 (random 10)))
```

Finally, the top-level procedure in our simulation, `twenty-one`, simulates one game of Twenty-One. It takes strategy procedures for a player and for the house as its two arguments. It creates initial hands for the house and the player, then plays the player strategy, then plays the house strategy. `Twenty-one` returns 1 if the player wins the simulated game and 0 if the house wins.

[1] For tutorial, we will ask you think about how to keep track of additional information about a hand.

```
(define (twenty-one player-strategy house-strategy)
  (let ((house-initial-hand (make-new-hand (deal))))  ; set up house hand
        ; let is covered on pp.58-61 of text
    (let ((player-hand  ; set up initial hand, and play out
           (play-hand player-strategy  ; strategy to use
                      (make-new-hand (deal))  ; initial player hand
                      (hand-up-card house-initial-hand))))  ;
                            ;information about house hand available to player
      (if (> (hand-total player-hand) 21)
          0                                 ; ``bust'': player loses
          (let ((house-hand   ; play out house hand
                 (play-hand house-strategy
                            house-initial-hand
                            (hand-up-card player-hand))))
            (cond ((> (hand-total house-hand) 21)
                   1)                       ; ``bust'': house loses
                  ((> (hand-total player-hand)
                      (hand-total house-hand))
                   1)                       ; house loses
                  (else 0))))))             ; player loses
```

`Hit?` is a simple interactive strategy procedure that can be used with `twenty-one`. It displays on the screen the information available to the player it is simulating and asks whether it should take another card. It returns true if you type `y` and false if you type any other character. [2]

```
(define (hit? your-hand opponent-up-card)
  (newline)
  (princ "Opponent up card ")
  (princ opponent-up-card)
  (newline)
  (princ "Your Total: ")
  (princ (hand-total your-hand))
  (newline)
  (princ "Hit? ")
  (user-says-y?))
```

Problem 1

Load in the code for problem set 2 and try playing a few games of Twenty-One against yourself by evaluating:

```
(twenty-one hit? hit?)
```

Remember that the first set of questions you will be asked are for the player's hand and the second set of questions are for the house's hand. There is nothing to turn in for this problem.

Problem 2

Define a procedure `stop-at` that takes a number as argument and returns a strategy procedure. The strategy `stop-at` should ask for a new card if and only if the total of a hand less than the argument to `stop-at`. For example `(stop-at 16)` should return a strategy that asks for another

[2] `User-says-y?` is defined as `(define (user-says-y?)  (eq? (read-from-keyboard) 'y))`. This compares an expression read from the terminal to the symbol `y`. We will learn about symbols and expressions in section 2.2.3.

card if the hand total is less than 16, but stops as soon as the total reaches 16. To test your implementation of `stop-at`, play a few games by evaluating

```
(twenty-one hit? (stop-at 16))
```

Thus, you will be playing against a house whose strategy is to stop at 16. Turn in a listing of your procedure.

Problem 3

Define a procedure `test-strategy` that tests two strategies by playing a specified number of simulated Twenty-One games using the two strategies. `Test-strategy` should return the number of games that were won by the player (and thus lost by the house). For example,

```
(test-strategy (stop-at 16) (stop-at 15) 10)
```

should play ten games of Twenty-One, using the value returned by `(stop-at 16)` as the player's strategy and the value of `(stop-at 15)` as the house strategy. It should return a non-negative integer indicating how many games were won by the player. Turn in a listing of your procedure and some sample results.

Problem 4

When the simulated games in the previous Problem ran, it was impossible for us to tell what was going on. It would be nice if we could watch a strategy play by observing its inputs and the decisions it makes. Define a procedure called `watch-player` that takes a strategy as an argument and returns a strategy as its result. The strategy returned by `watch-player` should implement the same result as the strategy that was passed to it as an argument, but, in addition, it should print the information supplied to the strategy and the decision that the strategy returns. For example,

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

should play two simulated games and show what each player does at each step. Turn in a listing of your procedure and some sample runs using it.

Problem 5

Ben has finally gotten Louis to describe his Twenty-One strategy. Here is how Louis was playing. If his hand contained fewer than 12 points, he always asked for another card. If his hand had more than 16 points, he always stayed with what he had. If his hand had exactly 12 points, he took another card if his opponent's up card was less than 4. If his hand had exactly 16 points, he would stay if his opponent was showing 10. If none of the above conditions held (his hand had between 12 and 16 points, exclusive), Louis would take a card if his opponent's up card was greater than 6, otherwise he would stay with what he had. Define a procedure called `louis` that implements Louis's strategy. Try Louis's strategy against the strategies of stopping at 15, 16, and 17 by evaluating

```
(test-strategy louis (stop-at 15) 10)
(test-strategy louis (stop-at 16) 10)
(test-strategy louis (stop-at 17) 10)
```

Problem 6

Implement a procedure `both` that takes two strategies as arguments and returns a new strategy. This new strategy will call for a new card if and only if *both* strategies would ask for a new card. For example, using the strategy

```
(both (stop-at 19) hit?)
```

will ask for a new card only if the hand total is less than 19 and the user requests a hit. Turn in a listing of your procedure and an example showing that it works.

3. Preparing for tutorial

Tutorial exercise 1

The simulation above is very restricted in that it represents a hand simply as a pair of numbers. Suppose we also want to keep track of the actual cards in the hand, both their values and their suits. One way to do this is with a data abstraction `card` to represent a card, and, using `card`, to implement a data abstraction `card-set` to represent a set of cards. A `hand` in the above simulation would then be represented as a hand up-card together with a set of cards. Sketch a sample implementation that carries this out, using list structure to represent the cards and card sets. How do you need to change the procedures `make-hand, first-card`, `hand-up-card`, `hand-total`, and `hand-add-card`? What else do you need to change in order to get the simulation to work (other than perhaps devising new strategies that take advantage of this extra information about hands)?

Tutorial exercise 2

In our simulation, we used an infinite deck in which each type of card from 1 to 10 is equally probable. Ben Bitdiddle is perturbed by this. He complains, "With an infinite deck of equally probable card values, I can't use my winning card-counting strategy."

(a) How would you implement a procedure that generates a fresh deck of cards that looks like:

```
(1 1 1 1 2 2 2 2 3 3 3 3 ... 10 10 10 10)
```

How would you modify this procedure to generate a deck that also includes face cards (Kings, Queens, and Jacks) given that each face card has a value of 10?

(b) How would you implement a `shuffle` procedure that accepts a deck as argument and produces a new deck with the same cards as the argument deck but in permuted order? For simplicity, shuffle should permute a deck by first cutting the argument deck into two equal-length halves then combining these halves into a new deck by alternately choosing a card from each of the two halves.

(c) Note that the above shuffle would always produce the same deck given the same argument. This is called a deterministic procedure. A better shuffle would introduce an element of randomness in the shuffle. How would you implement `random-shuffle` that accepts a deck a argument and cuts it as before but that combines the two halves by alternately choosing some number of cards from each half where the number of cards chosen varies randomly from 1 to 5? (Warning: be careful what you do if you choose more cards than remain in the half-deck.)

(d) Finally, how would you change the simulation to use a finite deck? (Hint: pass around a finite deck as an additional argument to the appropriate procedures and consider implementing procedures for selecting the top card of a deck and for returning the rest of the deck.) Ideally, we could test strategies by starting with a fresh deck, shuffling it a few times, then playing games until we run out of cards. When the deck runs out, we can arbitrarily call that game a player win.