

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.081—Introduction to EECS I
Fall Semester, 2006

6.188 Programming Assignment for week of Oct. 23

Issued: October 23, 2006

Due: October 31, 2006

See the 6.188 web page (linked off mit.edu/6.081) for the complete assignment for this week

The goal of this problem set is to give you some programming practice by asking you to work on a small project that requires reading a moderate amount of code. The project also asks you to work with a few higher-order procedures as a way to review the idea of higher-order procedures as way of capturing common patterns—something we already saw in 6.081, with Python.

The Game of Twenty-one (Blackjack)

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 Foxwood's 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 has that number as its value, 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 (computationally challenged) 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) ; should I hit?
         (let ((new-card (infinite-deal)))
           (play-hand strategy
                      (hand-add-card my-hand new-card)
                      opponent-up-card)))
        (else my-hand))) ; stay
```

For the purposes of this simple simulation, a “hand” of cards will be represented by two things—the value of the up card and all the cards in the hand. We will represent this using a very simple form of *data abstraction*, of the kind that will be discussed in lecture on February 15—we have a *constructor* procedure `make-hand` that creates a hand from two numbers, and two *selectors* `hand-up-card` and `hand-set` that return the up card and total card set of a given hand:

```
(define (make-hand up-card card-set)
  (list up-card card-set))

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

(define (hand-set hand)
  (cadr hand))
```

Don’t worry right now about the details of the Scheme primitives `list`, `car`, and `cadr`, 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-set` to be simple “black boxes” that allow us to represent hands.

We also have a data abstraction for the card set, which again we can treat as a black box:

```
(define (card-set-create first-card)
  (list first-card))

(define (card-set-add set card)
  (cons card set))
```

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 entire card set):

```
(define (make-new-hand first-card)
  (make-hand first-card (card-set-create 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 card set incremented by the card:

```
(define (hand-add-card hand new-card)
  (make-hand (hand-up-card hand)
             (card-set-add (hand-set hand) new-card)))
```

We can get the value of the hand by using `hand-total`:

```
(define (hand-total hand)
  (card-set-total (hand-set hand)))

(define (card-set-total set)
  (apply + set))
```

Again, don't worry about the details of these procedures. What matters is that you can get access to components of the data structure, primarily by using `hand-total` and `hand-up-card`.

Here are a couple of other useful procedures, which print out the cards in a hand:

```
(define (hand-print hand)
  (card-set-print (hand-set hand)))

(define (card-set-print set)
  (cond ((null? set)
        nil)
        (else (display (car set))
                (display " ")
                (card-set-print (cdr set)))))
```

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 on 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 (infinite-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.

```
(define (twenty-one player-strategy house-strategy)
  (let ((house-initial-hand (make-new-hand (infinite-deal)))) ; set up house hand
    ; let is covered on pp.58-61 of text
    (newline)
    (newline)
    (display "Playing player's hand") ; print out some useful data
    (let ((new-card (infinite-deal))) ; get new card
```

```

(let ((player-hand ; set up initial hand, and play out
      (play-hand player-strategy ; strategy to use
                  (make-new-hand new-card) ; initial player hand
                  (hand-up-card house-initial-hand)
                  ;information about house hand available to player
      )))
  (cond ((> (hand-total player-hand) 21)
        (newline)
        (display "Outcome: 0")
        (newline)
        0) ; ‘bust’: player loses
        (else
         (newline)
         (newline)
         (display "Playing house hand")
         (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)
                 (newline)
                 (display "Outcome: 1")
                 1) ; ‘bust’: house loses
                 ((> (hand-total player-hand)
                     (hand-total house-hand))
                  (newline)
                  (display "Outcome: 1")
                  (newline)
                  1) ; house loses
                 (else
                  (newline)
                  (display "Outcome: 0")
                  (newline)
                  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.¹

```

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

```

¹User-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 2.3 of the text.

Lab exercise 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 playing the player's hand and the second set of questions are for playing the house's hand. There is nothing to turn in for this problem.

Lab exercise 2 Define a procedure `stop-at` that takes a number as argument and returns a strategy procedure. Remember that if `stop-at` is to return a strategy (which is itself a procedure), then the overall form for `stop-at` should be something like:

```
(define (stop-at cutoff)
  (lambda (my-hand opp-up-card)
    ;; this lambda will create a procedure to be returned
    .....))
```

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 card if the hand total is less than 16, but stops as soon as the total reaches 16 or more. 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.

Lab exercise 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.

Lab exercise 4 When the simulated games in the previous Lab exercise 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.

Lab exercise 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)
```

Lab exercise 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.

Lab exercise 7 This problem is a bit more open-ended than the others. So far, we have seen three procedures that can be considered as primitive strategies: `hit?`, `stop-at` and `louis`. We have also seen one means of combination for strategies: `both`. We want you to implement at least three new primitive strategies, and at least three new means of combination for strategies.

For example, a new primitive strategy might work out the expected value of the next card to be drawn, and use that to decide if it is safe to hit. Another primitive strategy might be purely random. A new combination might take three primitive strategies and hit if a majority of them say to, or might hit if at least one of them says to hit.

Turn in a description of each primitive strategy and means of combination, together with the procedure that implements it. Also demonstrate that your primitives and means of combination actually work.

Now use your means of combination to invent three new compound strategies, formed by combining primitive strategies (or other compound strategies). Call these new strategies `st-1`, `st-2`, and `st-3`. Turn in a description of each strategy, the code that implements it, and examples showing it working.

Lab exercise 8 To analyze your efforts to bail out Louis, organize a small tournament. That is, run `test-strategy` some number of times (say 100 times), to compare two different primitive or compound strategies. Make a chart like the one below, that records how many times the first strategy beats the second. We use `st-1`, `st-2` and `st-3` to refer to the new strategies you created in exercise 7. You can add more strategies to the tournament if you like, but you should at least compare Louis's strategy with the ones you invented in exercise 7.

What can you deduce from this chart about which are the best and worst strategies for playing Blackjack?

	louis	st-1	st-2	st-3
louis				
st-1				
st-2				
st-3				