Programming Project 1: Twenty-One

This project must be done INDIVIDUALLY and is due Tuesday 7/8 at 11:59pm. Please check the course website for updates and errata.

For our purposes, the rules of twenty-one ("blackjack") are as follows. There are two players: the "customer" and the "dealer". The object of the game is to be dealt a set of cards that comes as close to 21 as possible without going over 21 ("busting"). A card is represented as a word, such as `10s` for the ten of spades. (Ace, jack, queen, and king are `a`, `j`, `q`, and `k`.) Picture cards are worth 10 points; an ace is worth either 1 or 11 at the player's option. We reshuffle the deck after each round, so strategies based on remembering which cards were dealt earlier are not possible. Each player is dealt two cards, with one of the dealer's cards face up. The dealer always takes another card ("hits") if he has 16 or less, and always stops ("stands") with 17 or more. The customer can play however s/he chooses, but must play before the dealer. If the customer exceeds 21, s/he immediately loses (and the dealer doesn't bother to take any cards). In case of a tie, neither player wins. (These rules are simplified from real life. There is no "doubling down," no "splitting," etc.)

The customer's *strategy* of when to take another card is represented as a function. The function has two arguments: the customer's hand so far, and the dealer's card that is face up. The customer's hand is represented as a sentence in which each word is a card; the dealer's face-up card is a single word (not a sentence). The strategy function should return a true or false output, which tells whether or not the customer wants another card. (The true value can be represented in a program as `#t`, while false is represented as `#f`.)

The file `~cs61a/lib/twenty-one.scm` contains a definition of function `twenty-one`. Invoking

$$(\texttt{twenty} - \texttt{one strategy})$$

plays a game using the given strategy and a randomly shuffled deck, and returns 1, 0, or -1 according to whether the customer won, tied, or lost.

For each of the steps below, you must provide a transcript indicating enough testing of your procedure to convince the readers that you are really sure your procedure works. These transcripts should include trace output where appropriate.

1. The program in the library is incomplete. It lacks a procedure `best-total` that takes a hand (a sentence of card words) as argument, and returns the total number of points in the hand. It's called *best*-total because if a hand contains aces, it may have several different totals. The procedure should return the largest possible total that's less than or equal to 21, if possible. For example:

```
> (best-total '(ad 8s))    ; in this hand the ace counts as 11
19
> (best-total '(ad 8s 5h)) ; here it must count as 1 to avoid busting
14
> (best-total '(ad as 9h)) ; here one counts as 11 and the other as 1
21
```

   Write `best-total`.

2. Define a strategy procedure `stop-at-17` that's identical to the dealer's, i.e., takes a card if and only if the total so far is less than 17.

3. Write a procedure `play-n` such that

   `(play-n strategy n)`

   plays `n` games with a given strategy and returns the number of games that the customer won minus the number that s/he lost. Use this to exercise your strategy from problem 2, as well as strategies from the problems below. To make sure your strategies do what you think they do, `trace` them when possible.

   Don't forget: a "strategy" is a procedure! We're asking you to write a procedure that takes another procedure as an argument. This comment is also relevant to parts 6 and 7 below.

4. Define a strategy that "hits" (takes a card) if (and only if) the dealer has an ace, 7, 8, 9, 10, or picture card showing, and the customer has less than 17, or the dealer has a 2, 3, 4, 5, or 6 showing, and the customer has less than 12. (The idea is that in the second case, the dealer is much more likely to "bust" (go over 21), since there are more 10-pointers than anything else.)

5. Define the best strategy that you can think of. (No, we won't grade you on how mathematically optimal it is. Just try to have fun.)

6. Generalize part 2 above by defining a function `stop-at`. `(stop-at n)` should return a strategy that keeps hitting until a hand's total is `n` or more. For example, `(stop-at 17)` is equivalent to the strategy in part 2.

7. Define a function `majority` that takes three strategies as arguments and produces a strategy as a result, such that the result strategy always decides whether or not to "hit" by consulting the three argument strategies, and going with the majority. That is, the result strategy should return `#t` if and only if at least two of the three argument strategies do. Using the three strategies from parts 2, 4, and 5 as argument strategies, play a few games using the "majority strategy" formed from these three.

8. Some people just can't resist taking one more card. Write a procedure `reckless` that takes a strategy as its argument and returns another strategy. This new strategy should take one more card than the original would. (In other words, the new strategy should stand if the old strategy would stand on the `butlast` of the customer's hand.)

9. **Print out a complete listing of your procedures before you begin this problem.** We are going to change the rules by adding two jokers to the deck. A joker can be worth any number of points from 1 to 11. Modify whatever has to be modified to make this work. (The main point of this exercise is precisely for you to figure out which procedures must be modified.) The reason you printed everything first is that otherwise the readers would have trouble grading the earlier steps, since you might mess up earlier work in attempting this one.

Submission instructions: In a directory named 'proj1', create files called 'twenty-one.scm' (which should contain your solutions for questions 1 through 8) and 'twenty-one-joker.scm' (which should contain your solutions for questions 1 through 9). Please make sure that each file is a fully playable copy of the project (not just a listing of the procedures that you changed). In addition, you must create a file called 'testing.txt' which contains an explanation of your testing and your transcripts – please surround your transcripts with real English words so that the readers can figure out what you are trying to do. Make sure that every file has your name and login on it and type 'submit proj1' at the shell prompt from within this directory. Also, please submit paper copies of both these files to the homework box.

By the way, the online submissions system is built so that you can submit as many times as you like with no penalty. Beware, however, that only your last submission will be graded, so don't do anything silly like re-submitting after the deadline.