CS61A Summer 2010

George Wang, Steven Tang, Jonathan Kotker, Seshadri Mahalingam, Eric Tzeng

Project 1

Due: July 2 2010 at noon

Note: The number of stars (★) besides a question represents the estimated difficulty of the problem: the more the number of stars, the harder the question.

1 How to Start and Submit

First, download `http://inst.eecs.berkeley.edu/~cs61a/su10/proj/twenty-one.scm` and use it as a template while filling out your procedures. You will submit the following files using `submit proj1`:

1. `twenty-one.scm`,
2. `joker.scm`, and
3. `twenty-one-count.scm`, if you choose to attempt the *Extra for Experts* section.

2 Introduction

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.)

3 Strategies

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 `twenty-one.scm` contains a definition of a function `twenty-one`. Invoking `(twenty-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. The instructions to generate a transcript are available in the Lab 1a document. **We suggest that you perform the exercises in order**.

4 Exercises

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 is called `best-total` because if a hand contains aces, it may have several different totals. The procedure should return the largest possible total that is 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 ace counts as 11 and the other as 1
21
```

   Write `best-total`.

2. (★) Define a strategy procedure `stop-at-15` that takes a card if and only if the total so far is less than 15.

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 are asking you to write a procedure that takes another procedure as an argument. This comment is also relevant to problems 6 and 7 below.

4. (★) Define a strategy named `dealer-sensitive` 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. (★★) Generalize problem 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 15)` is equivalent to the strategy in problem 2.

6. (★) On Earth Day, your local casino has a special deal: If you win a round of 21 with a spade in your hand, they pay double. You decide that if you have a spade in your hand, you should play more aggressively than usual. Write an `earth` strategy that stops at 17 unless you have a spade in your hand, in which case it stops at 19.

7. (★★) Generalize problem 6 above by defining a function `suit-strategy` that takes three arguments: a suit (`h`, `s`, `d`, or `c`), a strategy to be used if your hand doesn't include that suit, and a strategy to be used if your hand does include that suit. It should return a strategy that behaves accordingly. Show how you could use this function and the `stop-at` function from problem 5 to redefine the `earth` strategy of problem 6.

8. (★★) 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 problems 2, 4, and 6 as argument strategies, play a few games using the "majority strategy" formed from these three.

9. (★) 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 stand if the old strategy would stand on the `butlast` of the customer's hand.

10. (★★★) Copy your Scheme file to a new file, named `joker.scm`, 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.) You will submit both this new file and the original, nonjoker version for grading. You don't have to worry about making strategies optimal; just be sure nothing blows up and the hands are totaled correctly. (*Hint*: **Be sure to test `joker.scm`!** Try to think of as many test cases as you can, and try them out! It is easy to lose points because of a missed test case.)

5 Optional: Extra for Experts

This section is *optional*: you do not have to do it. Blackjack is virtually the only casino game where the player can have an edge over the house, utilizing a method called card counting. In `http://inst.eecs.berkeley.edu/~cs61a/su10/proj/twenty-one-count.scm`, we have provided a modified `twenty-one` procedure that incorporates the idea of card counting.

The rules of `twenty-one-count` are roughly the same as those of regular `twenty-one`, with a few changes and extra pieces of information. First, there are now 5 decks of cards, and they are not reshuffled after every play. There are also now three strategies that determine whether or not to take another card. In order to decide which strategy to use, we must first establish the notion of a *count*. The `twenty-one-count` procedure maintains a sentence of cards already played. For example, if you play four hands, every card that you played and every card the dealer has shown you is in that sentence. With this knowledge, you can obtain a special number that describes how much the deck is in your favor: this number is called the *count*. When the count is below a certain value, called the *low break-point*, the first strategy is used; when the count is between the low break-point and a *high break-point*, the second strategy is used; and when the count goes over the high break-point, the third strategy is used. If you are interested, you can read `http://en.wikipedia.org/wiki/Card_counting#Basics` to understand the logic and theory behind counting cards.

Your task is to find a specific set of strategies and conditions that maximizes the number of wins in a card-counting Blackjack game. You will submit your solution in a file called `twenty-one-count.scm`, which contains the following:

- Three strategies, called `lo-strategy`, `mid-strategy`, and `hi-strategy`,
- A card-counting procedure, called `count-scheme`, and
- Two numeric definitions, called `lo-break-point` and `hi-break-point`.

You do not need to worry about how to maintain the sentence of past cards; this is done for you. In terms of counting, all you need to make is a procedure, called `count-scheme`, that takes in the sentence of past cards and returns a number. This number will represent the count. For example,

```
(count-scheme past-cards-sent)
```

returns a number that signifies the count, where `past-cards-sent` is a sentence of cards; for example, it could look like `(ah jh 3h 10d)`. `lo-strategy` will be used when the count is below the `lo-break-point`, `mid-strategy` will be used when the count is between the `lo-break-point` and the `hi-break-point`, and `hi-strategy` will be used when the count is above the `hi-break-point`.

Thus, your submission in `proj1extra.scm` will look something like:

```
(define (lo-strategy customer-hand dealer-up-card)  ; Strategy code here)
(define (mid-strategy customer-hand dealer-up-card) ; Strategy code here)
(define (hi-strategy customer-hand dealer-up-card)  ; Strategy code here)
```

```
(define (count-scheme list-of-cards)                ; Return value is a number)
(define lo-break-point 5)                           ; 5 is only an example; any number here is fine.
(define hi-break-point 8)                           ; 8 is only an example; any number here is fine.
```

To run `twenty-one-count`, you would call

`(twenty-one-count lo-strategy mid-strategy hi-strategy lo-break-point hi-break-point count-scheme)`,

and this would output the number of wins minus the number of losses after 40 hands.

To run this code more than once, there is another procedure `play-n-count`, as follows:

`(play-n-count lo-strat mid-strat hi-strat lo-break-point hi-break-point count-scheme num)`

would play `twenty-one-count` `num` times, and give you the net result of adding these plays together.

The 5 deck of cards will go through 40 hands before being reshuffled. The list of past cards will reset. Your code will be tested through 40000 hands; i.e., 1000 play-throughs of `twenty-one-count`. The code with the most hand wins will get a prize!