CS3 Scheme Practical: Simplified Blackjack

E H Blake

1. Introduction

This project is your practical exercise for the Functional Programming section of CSC3002F. Please note that while you may discuss ideas with others you may not copy or share any code. You will see below that this project has to be done with the supplied skeleton code in "**`cs3-black-jack.scm`**" and standard scheme only. Do not use or load the file "**`simply.scm`**": that is only for doing the exercises from the book. Details of what to hand in are in Section 6 below. Please pay careful attention to this. If you do not comply you will lose significant marks.

2. The Game

This project is about creating strategies to play a simplified version of the card game Blackjack or Twenty-One.

- There are two players: the "player" and the "dealer".
- The aim is to get a hand of cards that comes as close to 21 as possible without getting more than 21.
- This game is played with the normal card deck of four suites (Hearts, Diamonds, Spades and Clubs) of 52 cards, and this is reshuffled after every round.
- The picture cards (jack, queen and king) are worth 10.
- The ace can be either 1 or 11 depending on what works best for the player.
- Other cards simply have their nominal value (2–9).
- The players each get 2 cards.
- The dealer's first card is face up and known to the player.
- The dealer has a house rule to take another card if the dealer total is under 17 and stands at 17 or higher.
- The player can choose to take more cards or stand, but must do so before the dealer.
- If the player goes over 21 the player loses and the dealer does not play.
- The player with the highest total wins and if they are equal it is a draw.

3. Strategies

The dealer's strategy has been explained above: if the total value of the hand is less than 17 then take another card. The outcome is either a total between 17 and 21 or something more, known as bust.

Your task is to program the player's tactics. This is a function which takes two arguments: the player's hand so far and the face-up card of the dealer. This function returns true if the payer wants another card or false if not.

4. Skeleton Program

Please use the guidelines on Scheme (Lisp) style when you write your code. Load the file **`cs3-black-jack.scm`** to provide the skeleton of the game. Load the file **`my-twenty-one.scm`** to see an outline of your code hand in. The functions for the book "Simply Scheme" (called *`simply.scm`*) are *not* needed and *should not be used*[1].

An individual card is represented by a list, like **`'(2 c)`** for the two of clubs, and the picture cards are **`A, J, Q, K`**: so **`'(A s)`** is the ace of spades. A hand (or the whole deck) is represented as a list of cards, for example: **`'((Q d) (6 h) (3 h) (6 c))`** represents the hand Queen of Diamonds, six of Hearts, three of Hearts and six of Clubs. Since it adds up to 25 it represents a bust[2].

The skeleton contains two strategies as examples. One is called **`stupid`** for obvious reasons and the other is called **`stop-at-17`** which is exactly the same as the dealer strategy.

[1] "cs3-black-jack.scm" provides a function random which is used internally and which is taken from "simply.scm", this is only in fact needed for Gambit but does no harm in Racket.

[2] BTW: the system adds the latest cards to the front of the list as a hand builds up.

The game function is a higher-order function that is called with the desired payer strategy

(black-jack strategy)

The function **black-jack** returns 1 for player win, 0 for draw and -1 for dealer win. It can also display the hands in a round via display statements that are commented out (look for comments with **<comment out>** in them).

5. Your Assignment: Black Jack (Total 100 Marks)

There are 8 sub questions here. Please read them carefully and then submit the answers as stated below (Section 6).

1. Aces can count either 1 or 11 depending on what works best for the player. There is currently a function called **best-hand** that returns values based on the assumption that aces have the value 1, fix this so that **best-hand** picks the best combination of values for the aces in a hand so that the player to get as close to 21 without going over, where possible. [15]

   So for example, once it is fixed:

```
> (best-hand '((A d) (8 s)))        ; The ace counts 11
19
> (best-hand '((A d) (8 s) (5 h)))  ; Here the ace counts 1
14
> (best-hand '((A d) (A s) (8 s)))  ; Here the one ace is 11 and
                                    ; the other is 1
20
```

2. There is already a strategy **stop-at-17**. Create a function **stop-at** that takes an argument **n** that determines a **strategy** where a card is taken only if the best hand so far is less than **n**. So **(stop-at 17)** will return a function that will be the same as **stop-at-17**. [6]

3. Write a high order function **repeat-game** that takes a strategy and a number as arguments such that: [6]

(repeat-game strategy n)

   plays **n** games with the **strategy** specified and returns the number of games won minus the number of games lost (draws don't count either way).

4. Create a strategy function **clever** that takes into account both the player's current total and what the dealer's up card is. Look at **stupid** for ideas. [15]

   - If the player has 11 or less you always take a hit.
   - If it is 17 or higher you always stand.
   - If the player has 12 you stand iff the dealer's up card is 4, 5 or 6, otherwise hit.
   - If the player has ≥13 and ≤16 then take a hit iff the dealer has an ace or any card of value 7 or higher showing, otherwise stand.

   Hint: you are likely to have to create a predicate **member?** that takes two arguments and checks to see if an element can be found in a list.

5. Define a function **majority** that takes three strategies as arguments and produces a strategy as a result. This resulting strategy decides whether or not to "hit" by checking 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 return #t. [12]

6. Write a data gathering extension of **repeat-game** (Question 3) called **get-stats** which gathers the statistics of several repeats of the game, in a list. Every data point represents a large number of repeats of the game and there will be several such data points in the output list. So **get-stats** takes three arguments, a **strategy**, a **repeat-count** for each data point and a count of how many **data-points** there should be in the list it creates. For example, [6]

(get-stats stop-at-17 100 10)

   yields a list of 10 data points each corresponding to 100 repeats of the game, something like:

```
(-1 -3 -18 3 13 -15 -5 1 -40 -2)
```

7. Run **`get-stats`** on several different strategies and gather lots of data on the performance of your various strategies. Be sure to include at least, **`clever`**, **`stop-at-17`** and **`majority`**. Put these results into a spread sheet (or other system) and calculate the averages and standard deviations. Plot graphs of the results, either by hand or using the spread sheet or some other system like **`gnuplot`**. You may create your own extra strategies if you wish, see if they are better than the others you have been asked to do.

   Write a brief report (1–2 pages or 300-600 words) and include the results and the graph(s). Explain what you did and what your test system was and what the tested strategies were. Present the results clearly and fully (using tables and graphs). Discuss key features of what you found. Consider the variation in the data and whether you have enough samples (and do more if needed). Consider the implications of the results in a conclusion: Which is the best strategy? How well do any of them do against the dealer? Anything surprising? You might say if you think the interactive system (Question 8) would outperform the programmed strategies but consider how you would support such assertions with objective evidence. [30]

8. Write an interactive version where you choose to take a hit or not. The display functions in the function black-jack point the way. You will write a predicate **`hit?`** this returns true if the user wants a hit. Of course you will have to provide the necessary information to the player, including the current total of the player's hand. The following function will be useful to get the input: [10]

```
(define (hit-me?)
  (eq? (read) 'y))
```

   Play the game **`(black-jack hit?)`** a few times. Give an example where the standing before 17 resulted in a win for you (in other words supply a printout of the game from start to finish).

6. What to Submit – Please follow exactly.

Your Code: Insert your code into a file called "**`<stdnam001>-twenty-one.scm`**" (where you **`<stdnam001>`** represents your own student number. An outline of this file is available as "**`my-twenty-one.scm`**". All you code must comply with acceptable Scheme style rules (if in doubt refer to your slides). Failure to choose appropriate names, create useful comments, etc. can lead to a loss of up to 20% per question. It is vital that you keep to the function names specified above and ensure that your code works (*without* needing the file "**`simply.scm`**"). Code that does not work can get at most 40% and most likely will get much less.

In addition to your code, or perhaps as comments inside the code, you need to give evidence of thorough testing of the code. Only if you chose to have a separate file of these test results, transcripts and traces then submit that as a pdf (preferred) or word file (with headings for the various parts).

Unit Tests: Include unit test cases (tests for every function and even parts of code, inputs and acceptable outputs), we are not asking for formal unit test using some package, instead you can have the unit tests and expected results in the comments. Examples of such tests are in the header comments for many of the functions in **`black-jack.scm`**, see **`play-dealer`**, etc.

Interpreter Transcript: For each of the questions (1–6 & *especially* 8) provide a dump of the session where you demonstrate that your code works. You may copy and paste this dump of the interactive interpreter session into a report. Make sure that you add clear headings so the tutors can see what question is being demonstrated. Make sure that this is a convincing demonstration of all types of conditions: label them to avoid confusion (in the way that the various cases for **`best-hand`** were labelled in Question 1).

Traces: Where appropriate include traces of the functions with the transcript to show what was called and with what parameters. You will need to provide proof that the various strategies perform as required under various conditions.

Report for Question 7: The report for Question 7 is to be done as a properly formatted document, preferably as an Adobe acrobat file (**`.pdf`**) but a word document is also acceptable.