University of Virginia, Department of Computer Science
cs150: Computer Science — Spring 2007

Problem Set 2: Procedurally Predicting Poker Probabilities

Due: Friday, 2 February
Beginning of class

Collaboration Policy - Read Carefully

For this problem set, **you are required to work with an assigned partner.** You will receive an email before Wednesday's class containing the partner assignments.

Before starting to work with your partner, you should go through questions 1 and 2 yourself on paper. When you meet with your partner, check if you made the same predicitions. If there are any discrepancies, try to decide which is correct *before* using DrScheme to evaluate the expressions.

You and your partner should work together on the rest of the assignment and turn in a single staped document containing both your answers to the questions 1 and 2, and one answer to the rest of the questions with both of your names on it. You should read the whole problem set yourself and think about the questions before beginning to work on them with your partner.

In addition to your partner, you may discuss this assignment with other students in the class and ask and provide help in useful ways. You may consult any outside resources you wish including books, papers, web sites and people except for materials from previous cs150 courses. If you use resources other than the class materials, indicate what you used along with your answer.

You are **strongly encouraged** to take advantage of the lab hours posted on the cs150 website.

Purpose

- Practice programming with procedures.
- Become familiar with cons cells and how they can be used to manage complex data.
- Understand recursive procedures.
- Create the beginnings of a poker bot that might win the 2007 World Poker Robot Championships

> *These robots are much better than the average player ... It would for sure make money online.*
> Phil Laak, Los Angeles Times, July 18, 2005.

In this problem set, you will develop procedures that can calculate odds for poker. Actually creating a poker bot involves making decisions based on incomplete information about what the other players have and how they behave. This is much harder than just calculating odds, but knowing the odds is important for any poker strategy.

In the game we will consider, Texas Hold 'Em (this is the poker variant that is used in most major tournaments) each player is dealt two hole cards. These are kept hidden from the other players and can only be used by the player who was dealt them. Then, five community cards are dealt. Every player may use these cards. There are betting rounds after the hold cards are dealt, and after the third, fourth, and final community cards.

At the end of the hand, each player makes their best five-card hand using as many of their own hole cards as they want and the remaining cards from the community cards. So, a player may make a hand using just the five community cards (and none of their hole cards), either of their hole cards and any four of the community cards, or both of their hold cards and any three of the community cards.

To calculate the odds a player will win a hand, we need to know all possible hands that player could get and how many of them beat the other player's hand. For example, when there is one community card left to be dealt, that means we need to consider how many of the remaining cards in the deck will allow the player to make a hand that is better than the other players hand.

Reading: Chapters 4 and 5 of the course book.

Becoming Pros at Cons

It never hurts for potential opponents to think you're more than a little stupid and can hardly count all the money in your hip pocket, much less hold on to it.
Amarillo Slim

The simplest compound data structure is a pair: a way of grouping two things into one. In Scheme, we can make pairs using `cons` cells. A `cons` cell has two elements. For silly historical reasons, these are known as the `car` and `cdr`:

`cons` cell

The elements in a `cons` pair can be anything, even another `cons` pair. By using `cons` pairs inside `cons` pairs inside `cons` pairs we can build up complex data structures to represent any data we want.

Question 1: For this question you should not use the Scheme interpreter. For each fragment below, either:

1. Explain why the fragment is not a valid Scheme expression; or,
2. Predict what value the expression will evaluate to. If the expression evaluates to a compound data structure, draw a box-and-pointer picture showing the structure (for example, see Figure 2.3 in SICP).

a. `(cons 100 50)`
b. `(cons (cons 1 2) 3)`
c. `(cons 1 (cons 2 3))`
d. `(car (cons (cons 1 2) null))`
e. `(cdr (car (cons (cons 1 2) null)))`
f. `(car (cdr (cons (cons 1 2) null)))`

Nullifying Null

A `list` is either the special value `null` (which represents the empty list), or a pair whose second element is a `list`.

Scheme provides many useful procedures for manipulating lists. The ones we use in this assignment include:

- `(list` *Expression**`)` — evaluates to a list containing the values of all the operand expressions. Evaluating `(list 1 2 3)` is equivalent to `(cons 1 (cons 2 (cons 3 (cons 4 null))))`.
- `(null?` *Expression*`)` — evaluates to `#t` if and only if the operand evaluates to `null`
- `(list?` *Expression*`)` — evaluates to `#t` if and only if the operand evaluates to a list.
- `(length` *Expression*`)` — evaluates to the number of elements in the value the operand evaluates to. Produces an error if the operand does not evaluate to a list. Note that `null` is a list of length 0.
- `(append` *Expression*$_1$ *Expression*$_2$`)` — evaluates to a list containing the elements of the list *Expression*$_1$ evaluates to followed by the elements of the list *Expression*$_2$ evaluates to. Produces an error if either operand does not evaluate to a list.
- `(map` *Expression*$_1$ *Expression*$_2$`)` — evaluates to a list that contains elements that result from applying the procedure *Expression*$_1$ evalutes to, to each elements in the list *Expression*$_2$ evaluates to. Produces an error if *Expression*$_1$ does not evaluate to a procedure or *Expression*$_2$ does not evaluate to a list.
- `(sort` *Expression*$_1$ *Expression*$_2$`)` — evaluates to a list that contains the elements that the list

Expression$_1$ evaluates to, but reordered according to the comparison procedure *Expression*$_2$ evaluates to. For `sort` to make sense, *Expression*$_2$ must evaluate to a *transitive* comparison procedure. That is, if `(`*Expression*$_2$ `x y)` evaluates to true and `(`*Expression*$_2$ `y z)` evaluates to true, then `(`*Expression*$_2$ `x z)` must also evaluate to true. Then, the sorted list will have the least element (according to *Expression*$_2$) at the front. (Note that `sort` takes its parameters in the opposite order of `map`: the first parameter to `map` is the procedure, the first parameter to `sort` is the list.)

Chapter 5 of the course book explains how `list?`, `length`, `append`, and `map` could be defined. We will explore several ways to define `sort` in later classes (but especially ambitious students will try to think of how to do this themselves).

Question 2: For this question you should not use the Scheme interpreter. For each expression below, either:

1. Explain why the fragment is not a valid Scheme expression; or,
2. Predict what value the expression will evaluate to. If the expression evaluates to a compound data structure, draw a box-and-pointer picture showing the structure (for example, see Figure 2.3 in SICP).

Assume the following definition is evaluated before each expression below is evaluated:

```
(define intsto3 (list 1 2 3))
```

a. `(list? (car intsto3))`
b. `(list? (cdr intsto3))`
c. `(length (car intsto3))`
d. `(length (cdr (cdr (cdr intsto3))))`
e. `(length (append intsto3 intsto3))`
f. `(length (append intsto3 4))`
g. `(length (append intsto3 null))`
h. `(map (lambda (x) x) intsto3)`
i. `(car (map (lambda (x) (> x 2)) (map (lambda (x) (+ x 3)) intsto3)))`
j. `(length (apply append (map (lambda (x) (list x (+ 1 x) (+ 2 x))) intsto3)))`

> The procedure `(apply` *Expression*$_1$ *Expression*$_2$`)` expects the first operand to evaluate to a procedure and the second operand to evaluate to a list. It evaluates to the value *Expression*$_1$ would evaluate to on operands that are the elements of the list *Expression*$_2$ evaluates to. For example, `(apply + (list 1 2 3))` means the same thing as `(+ 1 2 3)`.

k. `(car (sort intsto3 >))`

After you have predicted how each expression evaluates, remember to meet with your partner and discuss your results. After you have done this, try evaluating them in DrScheme to check your predictions. If any evaluate differently than you expected, explain why DrScheme evaluates the express the way it does.

Download: Download *ps2.zip* to your machine and unzip it into your home directory `J:\cs150\ps2` (it will take some time to download because it contains many tile images, so start the download and continue reading while it is finishing). See the Lab Guide document for instructions on unziping files and creating directories.

This file contains:

- *ps2.scm* - A template for your answers. You should do the problem set by editing this file.
- *poker.scm* - Provided code for this problem set. You should examine, but not need to modify, this file.

Representing Cards

To represent a card we need to keep track of both its rank (2 through Ace) and its suit (Hearts, Diamonds, Clubs, or Spades). We can do this using a `cons` cell: (these definitions are in *poker.scm*

```
(define (make-card rank suit) (cons rank suit))
(define card-rank car)
(define card-suit cdr)
```

We use the numbers 2 through 10 to represent the normal card ranks, and 11 (Jack), 12 (Queen), 13 (King), and 14 (Ace) to represent the special cards. The definitions in *poker.scm* allow us to use the names `Ace`, `King`, `Queen`, and `Jack` for these ranks.

Question 3: Define a procedure `higher-card?` that takes two cards are operands and evaluates to `#t` if and only if the first card has a higher rank than the second card. Note that the card suit does not matter in this comparison.

If your `higher-card?` procedure is correct, you should get the following interactions:

```
> (higher-card? (make-card Ace Diamonds) (make-card King Spades))
#t
> (higher-card? (make-card 3 Clubs) (make-card 2 Clubs))
#t
> (higher-card? (make-card Ace Diamonds) (make-card Ace Spades))
#f
```

Sorting Hands

In order to evaluate a hand, it will be useful to sort the hand according to the card values.

Question 4: Define a procedure `sort-hand` that takes one operand that is a list of cards, and evaluates to the cards sorted by decreasing card value (the highest card should be first in the list).

If your `sort-hand` procedure is correct, you should get the following interactions (the `display-cards` procedure provided in *poker.scm* makes it easier to see the cards; the sample hands are also defined in *poker.scm*):

```
> (display-cards (sort-hand royal-flush))
"Ah Kh Qh Jh 10h"
> (display-cards (sort-hand ace-high))
"Ad 10s 8c 7c 3s"
```

Hint: you should not need more than one line for your definition.

Since the rankings of poker hands don't just depend on the card values, but on having pairs, triples, and quads of a given card, it will be more useful to sort the cards according to first the number of duplicates of each rank, and then by rank. For example, if the hand is `Ac Js Jc 7d 4c` the procedure `sort-by-ranks` should produce `((Js Jc) (Ac) (7d) (4c))` since the pair of Jacks are more important than the single A. Note that instead of being just a list of cards, the result is now a list of lists of cards.

The procedure `sort-by-ranks` is defined:

```
(define (sort-by-ranks cards)
  ;;; sorts cards into lists of cards of each rank, ordered by most
  ;;; cards and highest cards within group
  (sort (combine-adjacent-matches  ;; combine them into lists of matching rank
         same-rank?
         (sort-hand cards))        ;; cards sorted by rank
        (lambda (r1 r2)
          (if (= (length r1) (length r2))
              (higher-card? (car r1) (car r2))
            (> (length r1) (length r2))))))
```

Ranking Hands

There are few things that are so unpardonably neglected in our country as poker. The upper class knows very little about it. Now and then you find ambassadors who have sort of a general knowledge of the game, but the ignorance of the people is fearful. Why, I have known clergymen, good men, kind-hearted, liberal, sincere, and all that, who did not know the meaning of a flush. It is enough to make one ashamed of the species.
Mark Twain

There are two ways that a hand could beat other hands:

1. It is in a better category of hands (e.g., a flush beats a straight)
2. It is in the same category of hands, but with higher cards

The ranking of the hand categories is shown next.

Category	Description	Example
Straight Flush	Five cards in sequence all of the same suit	Ks Qs Js 10s 9s
Four-of-a-Kind ("Quads")	Four cards of the same rank	3h 3d 3c 3s Jd
Full House	Three cards of matching rank and two different cards of matching rank	Ac As Ah 7d 7h
Flush	Five cards of the same suit	Qh 10h 8h 3h 2h
Straight	Five cards in sequence	9d 8h 7c 6h 5s
Wheel Straight	Ace-2-3-4-5 straight	5d 4h 3c 2h As
Three-of-a-Kind ("Trips")	Three cards of matching rank	5d 5h 5c Kh Js
Two Pair	Two different pairs of matching rank	Jd Jh 5c 5h As
Pair	Two cards of matching rank	8d 8h Ac 7h 3s
High Card	Anything else	Kd 9h 7c 5h 2s

(Note: we've listed *wheel straight* as a separate hand category, even though it is usually listed just as a straight. It is the only straight that can use A as a low card, and the lowest possible straight.)

We have provided the `higher-hand?` procedure that defines the poker rules:

```
(define (higher-hand? hand1 hand2)
  (cond
    ((straight-flush? hand1)      (or (not (straight-flush? hand2))
                                      (and (straight-flush? hand2)
                                           (higher-similar-hand? hand1 hand2))))
    ((four-of-a-kind? hand1)      (or (and (not (straight-flush? hand2))
                                           (not (four-of-a-kind? hand2)))
                                      (and (four-of-a-kind? hand2)
                                           (higher-similar-hand? hand1 hand2))))
    ((full-house? hand1)          (or (and (not (straight-flush? hand2))
                                           (not (four-of-a-kind? hand2))
                                           (not (full-house? hand2)))
                                      (and (full-house? hand2)
                                           (higher-similar-hand? hand1 hand2))))
    ((flush? hand1)               (and (not (beats-flush? hand2))
                                       (or (and (flush? hand2)
                                                (higher-similar-hand? hand1 hand2))
                                           (not (flush? hand2)))))
    ((wheel-straight? hand1)      (not (or (any-straight? hand2) (beats-straight? hand2))))
    ((three-of-a-kind? hand1)     (and (not (beats-trips? hand2))
                                       (or (and (three-of-a-kind? hand2)
                                                (higher-similar-hand? hand1 hand2))
                                           (not (three-of-a-kind? hand2)))))
    ((three-of-a-kind? hand1)     (and (not (beats-trips? hand2))
                                       (or (and (three-of-a-kind? hand2)
                                                (higher-similar-hand? hand1 hand2))
                                           (not (three-of-a-kind? hand2)))))
    ((two-pair? hand1)            (and (not (beats-two-pair? hand2))
                                       (or (and (two-pair? hand2)
                                                (higher-similar-hand? hand1 hand2))
                                           (not (two-pair? hand2)))))
    ((pair? hand1)                (and (not (beats-pair? hand2))
                                       (or (and (pair? hand2)
                                                (higher-similar-hand? hand1 hand2))
                                           (not (pair? hand2)))))
    (#t                           (and (not (beats-high-card? hand2))
                                       (higher-similar-hand? hand1 hand2)))))
```

Your job is to define the `higher-similar-hand?` procedure it uses to compare two hands in the same category.

The rules for comparing poker hands of the same category specify that the most important part of the hand should be compared first. The most important part is the highest card with the highest number of duplicates. If the most important parts are equal, than the next most important part of the hand determines the higher hand. Note that the `sort-by-ranks` procedure we defined sorts the cards in a hand according to importance, so you can determine the higher hand by considering each element of the list produced by `sort-by-ranks` in order until you find one that is unequal.

Here are a few examples:

- Jenny has Ad Js Jc 3h 3d. Mark has Ks Js Jc 4h 4d. Mark has the higher hand. Both hands are in the category two pair. The most important part of the hand is the higher of the two pairs. Both Jenny and Mark have Jacks. The next most important part of the hand is the lower of the two pairs. Mark has fours, which beat Jenny's threes. The unpaired card doesn't matter, since the lower pair is more important than a single card.
- Anne has Ac Ad As 4s 4c. Dave has Ac Ad 4d 4s 4c. Anne has the higher hand. Both hands are in the category full house. The most important part of the hand is the three-of-a-kind. Anne's Aces beat Dave's fours.
- Greg has Ac Ks 5d 3s 2h. Erin has Ad Ks 5d 4c 2h. Erin has the higher hand. Both players have no pairs or special hands, so the high card wins. The highest card is most important. Both have Aces. The next highest card is next most important, but again they both have Kings. Next, we compare the fives, which again match. The fourth highest card is next most important. Erin's four beats Greg's three, so Erin has the higher hand.
- Alyssa has Ac Kc Qc Jc 10c. Ben has Ad Kd Qd Jd 10d. Neither player has a higher hand. Both players have straight flushes with the highest card an Ace. The hands are equal.

Question 5: Define the `higher-similar-hand?` procedure, completing the template we have provided in `ps2.scm`. It should take two poker hands as operands, and can assume the hands are of the same category. It should evaluate to `#t` if and only if the first hand beats the second hand.

If your procedure is correct, you should get the following interactions:

```
> (higher-hand? pair-jacks pair-kings)
#f
> (higher-hand? pair-kings pair-jacks)
#t
> (higher-hand? queens-up queens-up)
#f
> (higher-hand? ace-high ace-higher)
#f
> (higher-hand? ace-higher ace-high)
#t
> (higher-hand? kings-full-of-aces kings-full-of-jacks)
#t
```

Finding the Monster

To find the best possible hand, we will need to consider all possibilities and use the `higher-hand?` procedure to identify the best hand among them. There are five community cards and two hole cards, so a player can make a hand by using both hole cards and choosing any three of the community cards, by choosing one of the hole cards and any four of the community cards, or by just using the five community cards.

To find possible hands, you will find this procedure (defined in *poker.scm* useful:

```
(define (choose-n n lst)
  ;; operands: a number n and a list (of at least n elements)
  ;; result: evaluates to a list of all possible was of choosing n elements from lst
  (if (= n 0)
      (list null)
      (if (= (length lst) n)
          (list lst) ; must use all elements
          (append
           (choose-n n (cdr lst)) ;; all possibilities not using the first element
           (map (lambda (clst) (cons (car lst) clst))
                (choose-n (- n 1) ;;; all possibilities using the first element
                          (cdr lst)))))))
```

Question 6: Define the `possible-hands` procedure, completing the template we have provided in `ps2.scm`. It should take two operands: the first is a list of two cards representing a player's hole cards; the second is a list of 5 cards representing the community cards. It should evaluate to a list of all possible hands that could be made using 0, 1, or 2 of the player's hole cards and enough of the community cards to make a five-card hand.

Note that your `possible-hands` procedure doesn't depend on the elements of the operands being cards. You may find it easier to test by using scalar values instead. For example, `(possible-hands (list 1 2) (list 'a 'b 'c 'd 'e))` should evaluate to a list containing these elements (in any order):

```
((a b c d e)
 (1 b c d e) (1 a c d e) (1 a b d e) (1 a b c e) (1 a b c d)
 (2 b c d e) (2 a c d e) (2 a b d e) (2 a b c e) (2 a b c d)
 (1 2 c d e) (1 2 b d e) (1 2 b c e) (1 2 b c d) (1 2 a d e)
 (1 2 a c e) (1 2 a c d) (1 2 a b e) (1 2 a b d) (1 2 a b c))
```

Now that we have a list of all possible hands, we can find the best hand using the `higher-hand?` procedure.

Question 7: Define the `find-best-hand` procedure, completing the template we have provided in `ps2.scm`. It should take two operands: the first is a list of two cards representing a player's hole cards; the second is a list of 5 cards representing the community cards. It should evaluate the best possible hands that could be made by the player using 0, 1, or 2 of the player's hole cards and enough of the community cards to make a five-card hand.

If your procedure is correct, you should get the following interactions:

```
> (display-cards (find-best-hand aces-in-hole trip-nines))
"9d 9c 9s Ah Ad"
> (display-cards (find-best-hand big-slick community-clubs4))
"Ac Qc Jc 7c 5c"
> (display-cards (find-best-hand big-slick royal-flush))
"Ah Kh Qh Jh 10h"
```

For the first hand, both hole cards are used with the three nines to make a full house. For the second hand, the Ace of clubs hole card is used with the four club community cards to make a flush. For the third hand, no hole cards are used and the five community cards make a straight flush.

Hint: as in Question 4, you should be able to define this using only one line. Don't worry about efficiency in your definition (for now).

After the Turn

After the fourth community card (known as turn) has been dealt, there is one more community card to come. To make good decisions, a player needs to know the likelihood that she will have the winning hand after the final card is dealt, but doesn't know what the final card is.

For simplicity, we will assume the player is playing against only one opponent, and has good enough card-reading skills to know the other player's exact hand. (Of course, no real poker player is that good, but in many cases players do have a reasonable guess what cards the other players are holding.)

To analyze a hand, we determine how many of the possible river cards would allow the player to win or draw. The `analyze-turn-situation` procedure is defined below (and in *poker.scm*):

```
(define (accumulate-outs lst)
  ; lst is a list of triples representing  cards
  (if (null? lst) (list null null null)
      (let ((rest-outs (accumulate-outs (cdr lst))))
        (list (append (car (car lst))
                      (car rest-outs))
              (append (car (cdr (car lst)))
                      (car (cdr rest-outs)))
              (append (car (cdr (cdr (car lst))))
                      (car (cdr (cdr rest-outs))))))))

(define (analyze-turn-situation hole1 hole2 community)
  ;; remove all known cards from the deck
  (let ((current-deck (remove-cards (append hole1 hole2 community)
                                    full-deck)))
    ;; we want to find out how many of the remaining cards produce each
    result
    (accumulate-outs
     (map (lambda (river-card)
            (let ((outcome
                   (compare-hands?
                    (find-best-hand hole1
                                    (cons river-card community))
                    (find-best-hand hole2
                                    (cons river-card community)))))
```

```
            (if (eq? outcome 'higher)
                (list (list river-card) null null)
                (if (eq? outcome 'equal)
                    (list null (list river-card) null) ; chop
                    (list null null (list river-card))))))
          current-deck))))
```

The `analyze-turn-situation` procedure determines the cards left in the deck by removing the known hole cards and community cards as `current-deck`. Then, it uses `map` to try each possible river card from the current deck. If the river card would allow player 1 to produce a better hand than player 2, it puts that card as the first element in the list; if the hands would be equal, it puts that card as the second element; if player 2 would win, it puts that card as the third element. The `accumulate-outs` procedure combines all the sublists to form a list of all the winning and chopping outs.

Here's an example:

```
> (show-analysis (analyze-turn-situation connect67 aces-in-hole straight-
draw4))
Winning outs (15): 2c 3c 5h 5d 5c 5s 9c 10h 10d 10c 10s Jc Qc Kc Ac
Chopping outs (0):
Losers (29): 2h 2d 2s 3h 3d 3s 4h 4d 4s 6h 6d 6s 7h 7d 7s 8h 8d 8s 9h 9d
Jd Js Qh Qd Qs Kh Kd Ks As three-clubs))
```

To consider the situation after the flop (the first three community cards have been dealt), we need to look at all possibilities for both the fourth and fifth card. The `analyze-flop-situation` does this:

```
(define (analyze-flop-situation hole1 hole2 community)
  ;; operands: hole cards for player 1 and play 2 and community cards
  ;;     there must be 2 cards in each players hole cards and 3 community cards
  ;; result: a list of three elements (winning-outs, chopping-outs, loser) showing
  ;;     the turn and river cards that will lead for the each outcome for player 1.
  (let ((current-deck (remove-cards (append hole1 hole2 community) full-deck)))
    ;; we want to find out how many of the remaining cards produce each
    result
    (map (lambda (turn-card)
           (analyze-turn-situation hole1 hole2 (cons turn-card
           community)))
         current-deck)))
```

Question 8: Predict how long it will take to evaluate an application of `analyze-flop-situation` (for example, `(analyze-flop-situation connect67 aces-in-hole straight-draw3)`). You should start by timing `analyze-turn-situation`. You can do this using the `time` procedure: `(time (show-analysis (analyze-turn-situation connect67 aces-in-hole straight-draw4)))`. This will print out something like `cpu time: 3523 real time: 4823 gc time: 203` (the numbers here are made up for the example). The number after real time gives the number of milliseconds it took to do the evaluation (in this case 4.823 seconds). If you want to try evaluation and timing `analyze-flop-situation` you can, but you should predict how long it will take first.

Question 9: This implementation is obviously too slow for many practical uses, including building a poker bot. Suggest some approaches you would use if you wanted to make a faster implementation.

Credits: This problem set was originally developed for UVA CS200 Fall 2005 by David Evans and tested and improved by Dan Upton.