IT-2105 Homeworks, Autumn 2006

*All code for all homework sets must be written in Scheme.

1 Homework Set I

1. (a) Write a simple procedure to calculate the hypotenuse of a right triangle when given the lengths of the two right-angle sides.

   (b) Write a similar procedure to calculate the length of one of the right-angle sides when given the lengths of the other right-angle side and the hypotenuse.

2. Write a procedure that takes two input side lengths, x and y, and a flag. When flag > 0, the input sides are assumed to be those of the right angle of a right triangle and thus the hypotenuse needs to be calculated. When flag ≤ 0, one of the two inputs is the hypotenuse and thus the other right-angle side needs to be calculated.

3. Write procedure **(my-expt a b)** which uses recursive calls to itself to compute a^b. You can assume that b is an integer, but it may be positive, negative or zero. Handle all 3 cases using Scheme's **cond** statement. Also write procedure **(pterm c a b)** that uses my-expt to compute ca^b.

4. Write a procedure that takes real numbers x and y as inputs and uses pterm to produce $7x^5+12y^4+8xy$ as output.

5. (a) Given two input numbers, A and B, use nested **if** statements to simply print out "A wins", "B wins" or "A and B tie" depending upon the values of A and B. Use Scheme's **write** or **display** primitives for the printing.

   (b) Write the same procedure as in part a, but now use nested **and** and **or** statements instead of **if**s. You may need to read up on the semantics of **and** and **or** in Scheme.

6. Write the procedure (rps-winner a b), whose inputs a and b are one of the keywords *scissors*, *rock*, or *paper* for the classic rock-paper-scissors (RPS) game (www.worldrps.com). Output is then 0 for a tie, 1 if a wins, and 2 if b wins. Use a sensible mixture of **if** and **cond** statements for the decision making.

7. Write at least 3 rps-player routines of this form **player-name my-last-move opponent-last-move)**. Each routine takes the RPS move made by itself and that of the opponent during the last round and computes its RPS move for the current round.

8. Write the (somewhat complex) recursive procedure **(play-rps p1 p2 num-rounds p1-last p2-last p1-wins p2-wins)**. The inputs p1 and p2 are the names of rps-player routines written above, num-rounds is the number of rounds to be played, p1-last and p2-last are the last moves of the two players, and p1-wins and p2-wins keep track of the number of wins for each player. After the final round, this procedure should print out the number of wins for each player.

   Write **(rps p1 p2 num-rounds)** as the main routine for rps. It should simply call play-rps with sensible initial values to begin several rounds of play.

   Hint: To avoid redundant calculations in **play-rps**, you may want to use Scheme's **let** or **let*** (see Scheme manual) constructs to create and bind local variables. For example:

   (let ((x 20) (y 40)) (any-func x y) (any-other-func x y) ...)

   This declares and binds local variables x and y to 20 and 40, respectively, before executing the code in any-func and any-other-func.

   When one local variables initial value is dependent upon the value of another local variable, use let*. For example:

   (let* ((x 20) (y (+ x 50))) ...any code ...)

This binds x to 20 and y to x+50. The key difference is that let binds its variables in parallel, while let* binds them serially. You will learn more about these later in the course.

9. Consider the following procedures:

   ```
   (define (dec x) (- x 1))
   (define (inc x) (+ x 1))
   (define (funny+ a b) (if (= a 0) b (inc (funny+ (dec a) b))))
   ```

   Show an applicative-order substitution model to describe Scheme's evaluation of the procedure call **(funny+ 4 5)**.

10. Now consider an iterative process:

    ```
    (define (funny+2 a b) (if (= a 0) b (funny+2 (dec a) (inc b))))
    ```

    Show an applicative-order substitution model for the call **(funny+2 4 5)**.

2 Homework Set II

1. Write two procedures, one recursive and one iterative, where both compute $\frac{b^n}{n!}$ when given nonnegative integers b and n as inputs.

2. Consider the function g(n) defined as follows:

$$g(n) = \begin{cases} n & \text{if } n < 4 \\ g(n-1) + 4g(n-2) + 8g(n-3) + 4g(n-4) & \text{otherwise} \end{cases} \quad (1)$$

   Write two procedures, one recursive and one iterative, to compute g(n) when given n as input.

3. Examine exercise 1.12 on page 42 of the textbook. Write a recursive procedure **(pascal row column)** to compute the element of pascal's triangle in the specified row and column.

4. Begin by typing in whatever prime-number-test code from the textbook that you prefer. Then use that code as the basis for two new recursive procedures:

   (a) **(primes-in-range a b)** - This prints out all prime numbers between a and b (inclusive).

   (b) **(first-n-primes a n)** - This prints out the sequence of n consecutive primes greater than or equal to a.

5. The famous Goldbach conjecture states that every even integer greater than 2 is the sum of two primes. Write the procedure **(goldbach k)** that takes any positive even number k and finds and prints out two primes p_1 and p_2 such that $k = p_1 + p_2$. Feel free to write additional supporting procedures as needed. Verify that your code works on k as large as 1 billion.

6. Write two versions of a general accumulator, **(accum combiner null-value term a next b)**, one recursive and the other iterative. Accum begins with element a and uses (term a) as the first item to be collected. The combiner function takes two inputs, an item and the currently accumulated result; it then updates the accumulated result to incorporate the new item. The null-value is simply the initial accumulated amount. The next function determines the next value in the sequence.

   For example, to sum the squares of 1 to 10: **(accum + 0 square 1 increment 10)**(sum, where increment is simply **(define (increment x) (+ x 1))**.

7. Use the accum procedure to compute the following:

(a) The sum of all primes between a and b (inclusive). Here, you will probably need a special **next** function to increment to the next prime number.

(b) n! (hint: a = 1, b = n)

(c)

$$\sum_{k=a}^{b} \frac{1}{k} \tag{2}$$

(d)

$$\sum_{k=a}^{b} \frac{k!}{k^k} \tag{3}$$

8. Write procedure **(mm-y f a b delta)**, which takes function f and an input range [a, b]. It then checks a discrete number of values, using delta as the step size, between a and b, and records the maximum and minimum values of f(x) in that range. Implement this using two separate calls to **accum**, one for the maximum and the other for the minimum. Both values should be printed out by your procedure.

9. Write procedure **(mm-x f a b delta)**, which is similar to **mm-y** above. However, it finds the **values of x** that yield the maximum and minimum values of f(x). Again, this should involve two calls to **accum**. Hint: Any local functions that you define in mm-x can be passed to accum.

3 Homework Set III

1. Write a new general accumulator, **(filter-accum combiner filter null-value term a next b)**, that works similar to **accum** but only applies the combiner to terms that pass the test provided by the **filter** function. Write two versions, one recursive and one iterative.

   Test filter-accum on the problem of summing up the cubes of all primes between 10000 and 20000. Also test it on the problem of computing the product of the square roots of all composite (i.e., non-prime) integers between 100 and 150.

2. Write the procedure **(digit-count n d)**, which counts the number of occurrences of digit d in the base-10 representation of integer n. Also write the procedure **(sum-digits n)** to sum all digits in the base-10 representation of integer n.

   For example (digit-count 1022321 2) $\rightarrow$ 3, and (sum-digits 9987) $\rightarrow$ 33.

3. Write the procedure **(special-primes a b m d)** that uses filter-accum to compute the sum of all primes between a and b that have at least m occurrences of the digit d in their base-10 representation. For the filter argument to filter-accum, use an unnamed procedure created by Scheme's **lambda** primitive.

   For example, an unnamed procedure to accept only integers that are multiples of 7 would look like this: (lambda (x) (= (modulo x 7) 0)).

   Use filter-accum to compute two other combinations of your choosing, but in both cases, use a lambda to express the filter condition.

4. Write three different procedures to calculate $f(x) = x^8 + x^6 + x^4 + x^2 + x$. Each procedure should reduce the total number of multiplications by reusing the values of x^4 and x^2.

   The first version should use let* and bind the values of x^2 and x^4 to local variables, where x^4 is computed as (square x^2).

   The second version should do the same computations but use nested let's instead of let*.

   The third version should use nested lambdas instead of let's.

5. Each of the following expressions should the value 42. Your job is to define the procedure textbffunky-k in each case such that 42 is indeed the return value of the expression. In each case, the funky-k function cannot simply ignore it's input arguments (if it has any). These inputs must be used to compute funky-k's output.

   For example, if the expression is (funky-10 7) $\rightarrow$ 42, then one reasonable definition for funky-10 would be (define (funky-10 x) (* x 6))

   (a) (funky-0) $\rightarrow$ 42
   (b) ((funky-1)) $\rightarrow$ 42
   (c) ((funky-2) 10) $\rightarrow$ 42
   (d) (((funky-3))) $\rightarrow$ 42
   (e) (((funky-4 2) 3) 7) $\rightarrow$ 42

6. These are similar to the previous exercise, but now, in all cases where functions take arguments, those arguments must also be functions. In each case, provide definitions for the (really funky) functions rfunk-k such that the output of the expression is 42.

   For example, given the expression (rfunk-11 (rfunk-10 (rfunk-9))) $\rightarrow$ 42, the following definitions would work:

   (define (rfunk-9) (lambda () 2))
   (define (rfunk-10 f) (lambda () (* 3 (f))))
   (define (rfunk-11f) (* 7 (f)))

   (a) (rfunk-1 (rfunk-2)) $\rightarrow$ 42
   (b) (rfunk-3 (rfunk-4) (rfunk-4)) $\rightarrow$ 42
   (c) ((rfunk-5 (rfunk-5 (rfunk-5 (lambda () 1))))) $\rightarrow$ 42 (Hint: ((rfunk-5 (lambda () 1))) $\rightarrow$ 2).

7. Do exercise 1.37 on page 71 of the textbook. Remember to write both the recursive and iterative versions. Hint: the recursive version builds the fraction from the top down, while the iterative version builds it from the bottom up.

   Now define **(cont-frac3 n d target error)**, which continues to compute a continuous fraction, frac, until $\mid target - frac \mid < error$.

   Use cont-frac3 to find an approximation of the golden ratio that is accurate to within .00001 (as described in exercise 1.37). Finally, use cont-frac3 to find an approximation to the golden ratio that is within one trillionth of the actual value.

8. Type in code from the textbook for computing the derivative of a function (pg. 74). Then define a 2nd derivative function based on deriv.

   Next, type in the code for computing zero crossing and fixed points.

   Your task is to write the procedure **(find-extremum f first-guess)** whose goal is to find an extremum (i.e., max or min) of a function. From calculus, remember that the zero crossings of the derivative of a function are potential local mins and maxs of that function. Once a zero crossing such as x* is found, the 2nd derivative of f at x* can reveal whether x* is a maximum, minimum or possible inflection point: a positive (negative) 2nd derivative indicates a minimum (maximum), while a zero 2nd derivative is a necessary but not sufficient condition for an inflection point.

   find-extremum should use first-guess as a seed for the fixed-point procedure to begin the search for a zero crossing of the derivative. If found, the zero crossing should then be subjected to the 2nd derivative test. Both the point and an indication of whether it is a min, max or potential inflection point should then be printed out by find-extremum.

   Test your code with several different functions of your choosing.

9. Pam Picky and Quinn Quicky are long-time partners in a software development firm. Quinn writes a lot of quick and dirty code that **normally** works fine but occasionally fails on certain special cases. Pam is much more diligent and tends to close all the holes in her programs. Unfortunately, Quinn often takes a little too much pride in his work and does not want anyone to touch his masterpieces.

   Pam and Quinn have come to the amicable agreement that Pam will not modify Quinn's code, but she has the right to create **wrappers** around it. In short, all the inputs to Quinn's procedures and outputs from them can be analyzed and possibly modified by Pam's code.

   To save work, Pam wants to write a few general purpose wrappers that can be applied to many of Quinn's creations, since, after all these years, she knows what steps he tends to skip!

   For example, if Quinn writes the following division procedure:

   (define (divide x y) (/ x y))

   Then Pam could use a wrapper that checks for divide-by-zero.

   (define (wrap-for-zero-divide f) (lambda (x y) (if (= y 0) (display "Divide by zero error") (f x y))))

   The call (wrap-for-zero-divide divide) would then create a more secure version of divide.

   Help Pam out with 3 different wrappers, all using lambda to create unnamed, more secure, versions of Quinn's code.

   (a) Quinn never bothers to acquire much domain knowledge before writing routines, so he never knows the correct ranges of input values for his procedures. Pam needs to write **(range-wrap f a b)** to insure that when f (a single-argument function) is called, its argument is within the range [a, b]. Otherwise, an out-of-range error should be printed and f should not be called.

   (b) Again, Quinn's lack of domain knowledge often leads him to write programs that work with improperly scaled inputs. Write **(scale-input-wrap f factor)** that divides the single argument to f by **factor** before calling f.

   (c) Outputs from Quinn's functions often need to be scaled also. Write a wrapper to scale single-number outputs.

   (d) Write **(super-wrap f in-scale out-scale range-a range-b)** to create all 3 of the above wrappers in one shot. Super-wrap should call the other wrappers in nested fashion. Input values should be scaled before they are range tested.

4 Homework Set IV

1. Use combinations of car and cdr to retrieve B, C, E , F, I and J from the following list X:

   (define X '(A B (C D) ((E F G H) I) J))

   For example, to retrieve A, (car X), and to retrieve D, (car (cdr (car (cdr (cdr X))))), which is equivalent to (cadr (caddr X)) in Scheme.

2. Write a recursive procedure **(nth n elements)** to return the nth (0-based indexing) element from a list of elements.

3. Redo exercise 1 using only nested calls to nth. For example, to retrieve D, (nth 1 (nth 2 X)).

4. Use nested calls to cons to produce the following lists:

   (a) (((55)))

   (b) (55 33)

   (c) ((55 . 44) 33 22)

   (d) (55 33 44)

(e) ((55 . 33) (44 . 11))

(f) (1 (2 (3 (4 5))))

For example (cons (cons 10 20) (cons (cons 30 40) *nil*)) → ((10 . 20) (30 . 40)). Be aware of the difference between a list and a dotted pair, both of which are created in this exercise.

5. Type in the code for **(map proc items)** (page 105, section 2.2.1 of the textbook). Use it to write the following:

   (a) (factor-map elems k) - multiplies every element in the list by k and returns a list of the new elements.

   (b) (prime-map elems) - For each number in elements, if it is prime, return it. Otherwise, return a *nil*.

   Provide at least 2 sample runs of each mapping procedure that you design.

   Next, write the general mapping procedure **(appmap f elems)** that is identical to map except that it combines results using **append** instead of **cons**.

   Use appmap to write a new version of prime-map such that the final list of primes contains no *nils*.

   For example (prime-map2 '(3 4 5 6 7 8)) → (3 5 7).

6. As the basis for a card-playing program, the basic data abstraction for a playing card needs to be determined. Define two different abstractions, one called **scard** (simple card) and the other **pcard** (pretty card), where the constructors for each are:

   (define (make-scard suit value) (cons 'scard (cons suit value)))

   (define (make-pcard pretty-suit pretty-value) (list 'pcard pretty-suit pretty-value))

   The first element in each data structure is the card type. Suit is one of 1(spade), 2(heart), 3(club) and 4(diamond), whereas pretty-suit is simply the name of the suit. The card value is a number between 2 and 14 (ace), while the pretty-value is a number between 2 and 10 or one of the words jack, queen, king, or ace.

   For each of the two card types, define procedures to:

   (a) Check whether or not a card is of that type.

   (b) Return the suit.

   (c) Return the value.

   (d) Return the card's name: a two-element list (pretty-suit pretty-value)

   Next, define three generic selector procedures named **card-suit, card-value** and **card-name** that use the type of the card to determine whether to call scard or pcard procedures to determine the suit, value or name.

   Finally, define 4 simple generic predicates (**jack?, queen?, king?, ace?**) at the **card** level that test whether a card is a jack, queen, king or ace, respectively, via a call to **card-value**. At this level, no knowledge of the card type should be necessary. That is, the detail of card type should be *abstracted away* or *hidden beneath the abstraction barrier.*

7. Begin by writing a general recursive procedure **(gen-numlist a b increment)** that produces a list of elements from a to b (inclusive) with a step size of **increment**. For example:

   (gen-numlist 1 10 3) → (1 4 7 10).

   Next, use gen-numlist, map and appmap to build the procedure **(gen-card-deck)**, which generates a standard deck of 52 cards, represented as a simple list of 52 cards. Feel free to use the scard or pcard type.

   Finally, write a simple routine to return the card-names of all cards in a deck (or any other list of cards). Using **map** allows you to do so with very little code.

8. Write the general procedure **(insert-nth n item elems)** which inserts **item** as the nth (0-based indexing) element of the list **elems**. For example:

   (insert-nth 3 88 '(0 1 2 3 4 5 6)) → (0 1 2 88 3 4 5 6).

   Next, write **(remove-nth n elems)** to remove the nth element from elems.

9. Write a procedure that uses repeated calls to **insert-nth** to generate a shuffled deck of cards. As a basic algorithm, begin with two cards lists, A and B, where A is a sorted list of 52 cards and B is an empty list. Repeatedly remove the first element of A and insert it into a random position in B. Use Scheme's **random** procedure to determine random locations.

10. Write a procedure that uses repeated calls to **remove-nth** to generate a shuffled deck of cards. In this case, remove cards from randomly-chosen locations in A and push them onto the front (car) of B.

    Which of the two shuffling techniques appears to produce a more randomly shuffled deck of cards?

5 Homework Set V

1. Write the code for a general sequence generator, **(generator start step end-test filter term)**, where start is anything from a Scheme atom to a complex data structure, step is a function that takes one sequence element and produces the next one, end-test is a predicate that is true when the final sequence item has been generated, filter is a test that new elements must pass (otherwise, they are omitted from the sequence), and term is applied to each element before it is added to the sequence.

   For example: (generator 1 (lambda (x) (+ x 11)) (lambda (x) (> x 100)) (lambda (x) (= 0 (modulo x 2))) square) → (144 1156 3136 6084 10000)

   Use **generator** to:

   (a) create procedure **(gen-intlist2 a b s)** to generate integers from a to b with step size s.

   (b) create procedure **(gen-primelist2 a b s)** to generate integers from a to be with step size s but to only collect those that are prime.

   (c) create **(gen-intlists a b s)**, which produces a list of lists, where each sublist is a subsequence. For example:
   (gen-intlists 1 10 2) → ((1) (1 3) (1 3 5) (1 3 5 7) (1 3 5 7 9))

2. Write the procedure **(for-each proc elems)**, which applies the procedure, proc, to each element in elems but does not collect the results of each such application.

   Use for-each to write **(special-primes2 a b m d)**, which prints out (but does not collect) all primes between a and b that have m or more occurrences of the digit d (when written in base-10). For example:

   (special-primes2 100 10000 3 1) → 1117 1151 1171 1181 1511 1811 2111 4111 8111

3. Write the procedure **(deep-reverse elems)**, which reverses a list of elems and any nested sublists. For example:

   (deep-reverse '(a b ((c (d e))) (f (g (h i j))))) → ((((j i h) g) f) (((e d) c)) b a)

   Verify that it works on 3 nested lists of your choice.

4. Write the procedure **(tree-map proc tree)**, which applies the procedure, proc, to all elements of a tree (i.e., a list with any level of nesting) and returns the results in a tree of the exact same form. For example:

   (tree-map square '(1 ((2 3 (4))))) → (1 ((4 9 (16))))

   Use tree-map to write procedures that:

   (a) Compute the square root of the absolute value of every element in a tree.

    (b) Add a constant to every element of a tree.
    (c) Multiply each element of a tree by a constant.

5. Write the procedure **(merge-lists l1 l2 key test)** to merge two lists of items, each of which is assumed to already be sorted according to the criteria specified by key and test. For example:

   (merge-lists '((c 1) (b 4) (a 12)) '((d 3) (f 5) (g 10)) cadr <) → ((c 1) (d 3) (b 4) (f 5) (g 10) (a 12))

6. Use merge-lists as the basis for writing **(merge-sort elems key test)**, the classic sorting routine.

7. Use merge-sort as the basis for the general procedure **(basic-card-sort cards)**, which sorts any list of playing cards (use either scards or pcards) such that all spades come before all hearts, which come before all clubs, which come before all diamonds. Within each suit group, the cards must be sorted by ascending values.

   Verify that it works with two different lists of cards.

8. Write the procedure **(partition elems key eq-test)**, which partitions the elems into groups such that for any two elements x and y, they are in the same group if and only if (eq-test (key x) (key y)) is true. For example:

   (define x (gen-intlist2 1 20 1))

   (partition x (lambda (elem) (modulo elem 4)) =) →

   ((17 13 9 5 1) (18 14 10 6 2) (19 15 11 7 3) (20 16 12 8 4)) , i.e., integers that are equivalent modulo 4.

6 Homework Set VI

1. Write the procedure **(n-of n f)**, which calls the no-argument function f a total of n times and collects the results of each call in a list of length n.

   In addition, write the procedure **(do-times n f)**, which, again, calls f n times but does not collect the results.

2. There are hundreds, if not thousands, of variations on the card game of poker. However, most share a common ranking of hands:

   (a) Highest card
   (b) One pair
   (c) Two pair
   (d) 3 of a kind
   (e) Straight
   (f) Flush
   (g) Full House
   (h) 4 of a kind
   (i) Straight Flush - with Royal Flush being the highest of these

   Many web sites, such as www.pagat.com/vying/pokerrank.html#standard, explain these rankings in detail. Although many poker games, such as Texas Hold 'Em and 7-card stud, give players access to 6, 7 or more cards, it is only the best 5 of those cards that compose the actual **hand**. Hence, all of the standard poker rankings are based on a 5-card hand.

Your task is to write the procedure **(calc-cards-power cards)**, which takes a group of cards, normally 5, 6 or 7 (but your procedure should work with any number of cards) and finds the best 5-card poker hand within them. It should then return a power ranking of some sort.

One suggestion for the power rating is a list consisting of the power index, which is 1 for a hand with nothing (but a high card), 2 for a pair, 3 for 2 pairs,...and 9 for a straight flush. The remaining elements of the list are values used to break ties in cases where, for example, two players have a full house, one with aces and 10's, the other with queens and jacks. The complete power rating should reflect all potential tie-breaking information in the 5-card hand. So for the full houses above, the rating that covers all 5 cards would be (7 14 10) and (7 12 11), since the 3 aces (14) and 2 10's (10) are the only 5 cards used. However, for a weaker hand, such as a pair of kings with the next 3 highest cards being 8 6 and 5, the power rating would be longer: (2 13 8 6 5). Conversely, a royal flush would have a simple, but dominating power rating of (9 14), where the 14 indicates the value of the highest card in the straight.

Alternate rating systems are clearly possible, but all should incorporate the standard rules for comparing poker hands. Be sure to explain your rating system (in complete sentences) as an extensive comment to your code.

Finally, define the procedure **(power-test n)** which creates a deck of cards, shuffles it, and then pops n cards off the top of the deck. Those cards should then be sent to calc-cards-power to compute a power rating. The pretty-printed cards and their power rating should then be displayed. Run this at least 10 times, with n = 7 or greater, to verify that calc-cards-power is working correctly.

Feel free to use as many auxiliary procedures as necessary in designing calc-cards-power.

3. Write the procedure **(gen-active-deck)**, which creates a procob (procedural object) with one local state variable, cards (or deck). The dispatcher should then allow two messages to be send to the procob;

   (a) init - This should cause the cards to be set to a newly shuffled 52-card deck.

   (b) get - This should pop the top card off of the deck and return it. The popped card should be deleted from the deck. If there are no cards, then this should return nil or false.

4. Write the procedure **(gen-card-player)**, which should produce a procob that represents a card player. The local variables for the player are hole-cards, shared-cards, and power. The hole cards are those that the player possesses alone, while the shared-cards are those that all players share, such as the **flop** in Texas Hold 'Em. The power is the power rating of the complete hand, which consists of the hole-cards and the shared-cards.

   This procob should respond to several different messages:

   (a) reset - to prepare for a new round.

   (b) receive - to receive a new card.

   (c) share - to receive the list of shared cards.

   (d) power - to compute and return the power rating of its hand.

   (e) pp-hand - to pretty-print its complete hand.

5. Write the procedure **(gen-card-dealer n)**, which should produce a procob that represents a card player. Its state variables should be a) players, a list of n player procobs, b) deck, a card-deck procob, and c) flop, the shared cards for Texas Hold 'Em.

   The call to gen-card-dealer should initialize the players variable to a list of n newly-generated player procobs, and it should initialize the deck to a shuffled, 52-card deck.

   The dealer procob should respond to the following messages:

   (a) reset - This sends the reset message to each player, along with setting the deck to a newly shuffled set of 52 and setting the flop to the empty list.

(b) texas - This causes a round of Texas Hold 'Em to be simulated.

To simulate a round of Texas Hold 'Em, do the following:

(a) Deal two cards to each player. These are the hole cards.
(b) Deal 3 cards to the flop, then send these 3 cards to each player using a **share** message.
(c) Deal another card to the flop, then send all 4 flop cards to each player.
(d) Deal a final card to the flop, then send all 5 flop cards to each player.
(e) Sort the players by the power ratings of their hands.
(f) Print out the players and their cards in sorted order, with the best hands first.

In a real round of Hold 'Em, there would be several rounds of betting. This will come later in the semester!