CSE 341 Assignment 2
Basra

January 13, 2006

For your second assignment we're going to complete the Basra game. After your implementation, you would be able to actually play the game! But first you should familiarize yourself with the rules of Basra.

Due: Thursday, January 19, at 11 pm. No late assignments will be accepted. Submit your assignment using the online turnin link on the course web.

Game Rules

Initially you have a shuffled deck of 52 distinct cards, which is called *Deck* in the actual program. You should never worry about checking whether they're duplicates because there shouldn't be any.

First, deal four cards from *Deck* to the player (denoted $Hand$). Whenever the player plays a card (each play is a *round*), another card is dealt from the *Deck* so that the player always has four cards.

Then deal four cards from *Deck* to the table (denoted $Floor$). If $Floor$ has less than four cards after the player plays a card, cards are dealt from *Deck* so that $Floor$ always has at least four cards.

At each round, the player looks at $Floor$ and decides which card to play. Two possibilities may result from this play:

- The played card doesn't make a "capture", and is added to $Floor$
- The played cards makes a "capture"

Given that the played card has rank x, a card or a number of cards can be "captured" from $Floor$ if:

- The captured card has the same rank x, or
- The sum of the rank of captured cards equals to x

If the played card captures all of the cards from $Floor$, it is called a "sweep" and extra points are given for each sweep. In each capture, the captured cards, along with the played card, are removed and put to the $Scored$ stack.

After each round, one card is dealt from *Deck* to $Hand$, and if $Floor$ has less than four cards, enough cards are dealt from *Deck* to $Floor$ to satisfy the four-card requirement.

The game ends when either of the following is true:

- Both $Floor$ and *Deck* is empty
- The player's $Hand$ is empty (which implies that *Deck* must be empty)
- $Floor$ contains six cards already, and none of the player's cards can make a capture

When the game is over, the score is calculated by summing earned points from *Scored* stack:

- Each face card (J, Q, K) is worth 2 points
- Each non-face card (A, 1 - 10) is worth 1 point
- Each sweep is worth 3 points

Gaming tip

A player is not obliged to play a card that makes a capture. As long as *Floor* has less than 6 cards, the player can always play a card that doesn't make a capture and adds it to *Floor*. However, if the played card can make a capture, the capture is always performed.

Sorting:

You will be making use of either a Mergesort or Quicksort algorithm in order to sort an array of cards in this assignment. The following is a description of the Mergesort algorithm:

- If the list is of length 1, return the list
- If the list is of length greater than 1, split the list into two parts in the middle. This results in two lists - left and right
- Call Mergesort on left, then call Mergesort on right
- Once the two parts are returned, merge them into the new array in the following way: While left and right both have elements in them, if the first element in left is less than the first element in right, place it on the front of a new list called result. Similarly, if the first element in right is less than the first element in left, place the first element in right on result. Remove the element that was used, and repeat the process while building result until either left or right is empty. Once this is the case, if left has elements remaing, append it to result. Otherwise, append right to result.
- Return result

Alternatively, you may elect to use the Quicksort algorithm to sort your list:

- If the list is of length 1, return the list
- If the list is of length greater than 1, pick a pivot from the center of the list
- Split the list into two parts, left and right, such that all elements in left are less than the pivot, and all elements in right are greater than the pivot. Include the pivot in the left list.
- Recursively call quicksort on the left and right lists
- Return the result of appending the lists in the following order: left, right.

Functions to implement:

Now that you've already learned about types and datatypes, we're using new types and datatypes to represent a single card and a list of cards:
datatype suit = Spade | Heart | Diamond | Club
type rank = int
*type card = suit * rank*

type cards = card list

Note that in the skeleton code provided to you, those functions that you worked with in homework 1 have been modified to adapt to this new type system.

You will be implementing the following functions. Since your functions will be used throughout the game, you must implement the parameters in the given order. You will always assume that the passed-in parameters are valid.

1. $capture(floor : cards, card : card)$ - given $floor$, which is a valid $Floor$, and $card$, which is a valid card played by the player, return a list with a sublist of cards such that each sublist is a valid capture from $Floor$. For this function, the sublists should not overlap each other. That is, if a card exists in one sublist, it should not exist in any other sublist returned by this function.
2. $isGameOver(deck : cards, floor : cards, hand : cards)$ - given $deck$, which is a valid $Deck$, and $floor$, which is a valid $Floor$, and hand, which is a valid $Hand$, return $true$ if the end-game condition is satisfied (see game rule), and $false$ otherwise.
3. $produceShuffledDeck()$ - For this function, you should generate a shuffled deck of cards. You will be provided with the function $produceUnshuffledDeck()$, which will produce a deck that is in perfect order by suit and rank. In addition to that function, you will also be provided with the function $getRandomInt(max)$ which will produce a random integer between 1 and max inclusive. Using these two functions and any helper functions that you require, create a deck that has been shuffled randomly.
4. $sort(cardList : cards)$ - Given a valid list of cards $cardList$, utilize a Mergesort or Quicksort algorithm to sort the cards by suit and rank. Cards are ranked first by suit, and then by their rank. The resulting list should contain the cards in order of their suits starting with Spades, followed by Hearts, Diamonds, and finally Clubs. Within a suit, cards should be ordered by rank starting with Aces and going up to Kings. For example, if you were asked to sort the list [(Diamond, 4), (Spade, 2), (Club, 7), (Spade, 5), (Diamond, 1)], you should return the list [(Spade, 2), (Spade, 5), (Diamond, 1), (Diamond, 4), (Club, 7)]. In order to implement this function, you will need to have some way of comparing two cards. You may either use the provided functions for comparison or write your own.

You should use pattern matching in your functions.

Bonus

As an extension to the above parts of the project, it is possible to offer the user a choice of which set of cards he/she would like to capture from the floor in ambiguous situations. To lay the groundwork for such functionality, you can write the following function:

$produceAllCaptures(floor : cards, aCard : card)$ - This function is similar in behavior to *capture*, but it must return all possible sublists that are valid captures, including overlapping sets of cards. For example, if the floor provided consists of the following cards: [(Spade, 1), (Heart, 1), (Diamond, 1), (Diamond, 2)] and the card played is (Diamond, 3), you should return a list with the following combinations: [[(Spade, 1), (Diamond, 2)],[(Heart, 1), (Diamond,2)], [(Diamond,1),(Diamond,2)],[(Spade,1),(Heart,1),(Diamond,1)]]. This result occurs because you can pair up any of the three cards with rank 1 with the card with rank 2, and also because you could use all three cards with rank 1 to make a total of 3. The order of the cards in the lists and the order in which the lists are returned is not important.

How to write and compile the game:

You will be given a file named **basra_full.sml**, which contains the entire game code, with the exception that the three functions you are to implement are left empty. To test your own four functions, you should

first write them in a separate **basra_work.sml** file, and after testing it, copy the three functions to **basra_full.sml**.

Then, in sml, type
use "basra_full.sml"

You should see under the same directory a compiled stand-alone executable **Basra.**$< arch - type >$, where $< arch - type >$ depends on the machine you're using (ex: under linux it would be **Basra.x86-linux**).

To run the executable, in your command line (or shell), type
sml @SMLload=Basra.$< arch - type >$
and start playing!

To turn in your homework, submit **basra_full.sml**.

Sample Log:

Below is an sample execution log:

- val x: cards = [(Spade, 1), (Spade, 2), (Spade, 3), (Spade, 7)];
val x = [(Spade,1),(Spade,2),(Spade,3),(Spade,7)] : cards
- val y: card = (Heart, 3);
val y = (Heart,3) : card
- capture(x, y);
val it = [[(Spade,3)],[(Spade,1),(Spade,2)]] : cards list
- val x: cards = [(Spade, 5), (Heart, 3), (Heart, 11), (Club, 1)];
val x = [(Spade,5),(Heart,3),(Heart,11),(Club,1)] : cards
- capture(x, (Club, 2));
val it = [] : cards list
- val floor=[(Spade, 1), (Club, 3), (Spade, 6), (Spade, 13), (Diamond, 9), (Diamond, 11)];
val floor = [(Spade,1),(Club,3),(Spade,6),(Spade,13),(Diamond,9),(Diamond,11)] : (suit * int) list
- val hand = [(Spade, 5), (Heart, 5), (Diamond, 5), (Club, 2)];
val hand = [(Spade,5),(Heart,5),(Diamond,5),(Club,2)] : (suit * int) list
- isGameOver(nil, floor, hand); #note: Deck is set to nil for testing, not possible in reality
val it = true : bool
- sort(floor);
val it = [(Spade,1),(Spade,6),(Spade,13),(Diamond,9),(Diamond,11),(Club,3)] : cards

Type Summary:

A correct solution of your four implemented functions will cause these bindings to be printed in the read-eval-print loop:

val capture = fn : cards * card → cards list
val isGameOver = fn : card list * card list * card list → bool
val produceShuffledDeck = fn : unit → cards
val sort = fn: cards → cards

Keep in mind that getting the right bindings does not necessarily indicate that your code is correct. You need to test your functions to verify that your code performs the way it is supposed to.

Assessment:

The homework that you turn in will be graded based on the following:

- Correctness - The code should perform the functions described above
- Style - You should exhibit good style in your coding. Pay particular attention to indentation and commenting. If you are doing something that may be unclear to someone reading the code, you should comment that section and explain what you are doing.
- Do not use material not covered in class. You should make good use of pattern matching, in addition to material covered earlier, to complete this assignment.