Your homework is due at 11.59pm on September 19, 2018. You will use the Learn OCaml platform to submit and automatically grade your homework. You can submit your homework as often as you like until the due date and immediately check your grade.

House of Cards

We continue the example of playing cards that we have seen in class (the file datatypes.ml on the myCourses schedule) and the material described in Chap. 1.3 from the course notes.

1. **Comparing Cards**

   Write a function dom_card of type card -> card -> bool which takes in two cards and tests whether the first card is greater than or equal to the second. To compare suits, use the relative ordering Spades > Hearts > Diamonds > Clubs (see the function dom we defined in class, adapted as dom_suit here). You will need to write an auxiliary function dom_rank : rank -> rank -> bool that will allow you to compare the rank of two cards using the relative ordering:
   Six < Seven < Eight < Nine < Ten < Jack < Queen < King < Ace.
   A card with rank r1 and suit s1 is considered greater than a card with rank r2 and suit s2, if suit s1 is greater than suit s2 or if they have the same suit and the rank r1 is greater than the rank r2.

   Note that you are free to modify the given function headers to use pattern matching as long as you do not change the type signatures of the functions. This applies to all questions!

2. **Sorting Cards in a Hand**

   Given a hand of cards, we want to sort them in decreasing order using the ordering described above. Implement a function sort which, given a hand of cards, uses **insertion sort** to return a sorted hand of cards.
   For example, given a hand with two distinct cards *A* and *B* with dom_card A B = true, the result should be Hand(A, Hand(B, Empty)).
   To do so, implement and use an auxiliary function insert : card -> hand -> hand that, given a card and a hand, returns a copy of the hand with the card inserted in its sorted place.
3. **Generating a Deck of Cards**

   To play a game, we first need to generate a new deck of cards. Here, we represent a deck as a list of cards. Write a function generate_deck which takes a list of suits and a list of ranks and generates a list of cards where each suit is paired with all the possible ranks. The resulting list should preserve the order of the given inputs. For example:
   generate_deck [Hearts; Clubs; Diamonds] [Seven; Eight; Nine; Ten; Jack; Queen]
   should return
   [(Seven, Hearts); (Eight, Hearts);
   (Nine, Hearts); (Ten, Hearts);
   (Jack, Hearts); (Queen, Hearts);

(Seven, Clubs); (Eight, Clubs);
(Nine, Clubs); (Ten, Clubs);
(Jack, Clubs); (Queen, Clubs);
(Seven, Diamonds); (Eight, Diamonds);
(Nine, Diamonds); (Ten, Diamonds);
(Jack, Diamonds); (Queen, Diamonds)]

4. **Shuffling Cards**

   Last, we want to shuffle a list of cards. For this exercise, you will implement a modified version of the Fisher-Yates shuffle as follows:

   - Iteratively call the function Random.int n, where n is the number of cards not yet used, to generate a random index
   - Append the card at that index of the list of cards not yet used to the return list

   Thus, this should return a list of cards placed in order of when they were picked (first picked is first in list).

   *Hint 1: We recommend writing a function split : card list -> int -> (card * card list) that extracts the card at an index from the deck, returning that card as well as the remaining deck in the original order without that card.*
   Hint 2: As you can see in the template, we also recommend writing a recursive function select : card list -> int -> card list that gets a random card from the given list and places it first in a list. Don't forget to figure out what happens with the remainder of the list in this function!

Sparse Representation of Binary Numbers

We can represent binary numbers as a list of increasing integers where each element is a power of two:

type nat = int list
For example: 5 = [1; 4] or 15 = [1; 2; 4; 8] or 17 = [1; 16]. We can represent 0 with the empty list.

The sparse representation can be a more useful way of representing numbers than using a dense representation (i.e. one using ones and zeroes), especially for human-readable arithmetic.

1. Implement a function inc : nat -> nat which increments a given sparse binary number.
2. Implement a function dec : nat -> nat which decrements (i.e., subtracts one from) a given sparse binary number. If given 0 as input, you should raise the exception Domain.
3. Implement a function add : nat -> nat which adds two sparse binary numbers.
4. Implement a function sbinToInt : nat -> int using a helper function toInt which translates a given sparse binary number to an integer tail-recursively using an accumulator.