Assignment 2
COMP 302 Programming Languages and Paradigms

Brigitte Pientka
MCGILL UNIVERSITY: School of Computer Science

Due Date: Oct 3, 2013

Q1. 15 points Inductive proof (**Hand in your proof as a pdf file q1.pdf**).

Recall the mobile from HW 1 where we defined a mobile a follows.

```
type mobile = Object of int | Wire of mobile * mobile
```

Below we give two functions which both compute the overall size of a mobile by counting the wires and objects.

```
let rec size m = match m with
  | Obj w -> 1
  | Wire (m1, m2) -> 1 + size m1 + size m2

let rec size' m acc = match m with
  | Obj w -> acc + 1
  | Wire (m1, m2) -> size' m1 (size' m2 (1 + acc))
```

Prove that for all `m:mobile`, `size m = size' m 0.`

Q2. 10 points Given a function `f:'a -> 'a` and a positive integer n, we want to define `repeated` as a function that returns `f` composed with itself n times. The type of the function `repeated` is `int -> ('a -> 'a) -> 'a -> 'a` . Stage your function properly such that *it generates code s.t.* `f` *is composed with itself* n *times*, i.e. this generated code is independent of the function `repeated`.

```
# let foo = repeated 2 square;;
val foo : int -> int = <fun>
# foo 2;;
- : int = 16
# foo 3;;
- : int = 81
#
```

```
# let t8 = repeated 3 (fun x ->
    x*2);;
val t8 : int -> int = <fun>
# t8 2;;
- : int = 16
# t8 3;;
- : int = 24
# t8 4;;
- : int = 32
#
```

Q3 30 points **Maximum Likelihood.**

Let's say we have a total of 10 marbles in a urn. The marbles are black and white, but we do not know how many black and white marbles are in the urn.

We draw 3 marbles at the same time. Based on the result of this draw, we want to make an educated guess as to how many black marbles are in the urn.

For example: Let us assume we draw 2 black and 1 white marbles. Hence, we know with certainty that the number of black marbles is between 2 and 9. In fact, we know that some values are more plausible than others.

We can compute the probability that given N black marbles we draw 2 black marbles follows:

$$P_N(X_1 = 2) = \frac{\binom{N}{2}\binom{10-N}{1}}{\binom{10}{3}} = \frac{N * (N-1) * (10-N)}{240}$$

Recall:

$$\binom{n}{k} = \frac{n!}{k! * (n-k)!}$$

Our goal is to estimate the number of black marbles in the urn. To put it differently, we are interested in finding out what number of black marbles in the urn is most likely.

To compute the maximum likelihood, we repeat the above experiment k times. Each time we draw 3 marbles, and we record how many black marbles where among the 3. For example when repeating the experiment 3 times we get [2 ; 0 ; 1].

This means that there were 2 black marbles out of the 3 marbles drawn the frist time, there were no black marbles the second time, and there was one black marble the third time.

We then compute the *distribution matrix*, i.e. a matrix where the i-th row corresponds to the distribution table of the i-th experiment. The distribution matrix for the above experiment is.

```
[[0.0,0.0,0.0667,0.175,0.300,0.417,0.500,0.525,0.467, 0.3,0.0]
 [1.0,0.7,0.467, 0.292,0.167,0.083,0.033,0.008,0.000, 0.0,0.0]
 [0.0,0.3,0.467, 0.525,0.500,0.417,0.300,0.175,0.067, 0.0,0.0]
 ]
```

To compute the most likely number of how many black marbles are in the urn, we need to compute the *combined distribution*. Since these were independent experiments, the i-the entry in the combined distribution table is

$$\prod_{x=0}^{k} x_{ji} \quad \text{where } 0 \leq j \leq k$$

We have already implemented basic functions for computing the binomial coefficient, finding the maximum in a list, and computing the probability that given N black marbles in the urn, we draw X black marbles. Your task is to implement the missing functionality to allow us to find out how many black marbles are most likely to be in the urn.

To implement the missing parts, use built-in higher-order functions such as `map` and `fold_right` to write elegant and short code! You can also use `tabulate` which is given in the template code.

Q3.1 (5 points) Compute the distribution table `t` containing the probability that given `n` black marbles are in the urn when we get `x` black marbles among the marbles drawn. For example, the distribution table when we have 10 marbles in the urn, draw 3 marbles, and encounter 2 black marbles among the marbles picked, looks as follows.

```
[0.0; 0.0; 0.067; 0.175; 0.3; 0.417; 0.5; 0.525; 0.467; 0.3; 0.0]
```

For better readability, we truncated the floating point numbers showing only up to three decimals. The `i`-th element in the list describes the probability that given we have `i` black marbles in the urn, we draw 2 black marbles.

Compute `dist_table (marblesTotal, marblesDrawn) x = t`, where `marblesTotal` describes the total number of marbles in the urn, `marbeslDrawn` denotes the number of marbles drawn, and `x` is the number of black marbles drawn.

Q3.2 (5 points) Implement the function `emptyMatrix` which when given a matrix returns true if it is empty. An empty matrix contains rows of empty lists.

Q3.3 (5 points) A distribution matrix `m` is represented as a list of lists. Row `i` is the distribution table for the `i`-th experiment in `resultList`. Compute the distribution matrix

`dist_matrix (marblesTotal, marblesDrawn) resultList = m`.

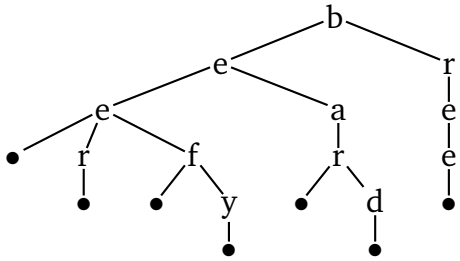
Q3.4 15 points Compute the combined distribution table from a distribution matrix. Implement the function `combined_dist_table matrix = t`

Here is the combined distribution table of the distribution matrix given earlier:

```
# combined_dist_table (dist_matrix (10, 3) [2; 0; 1]);;
- : float list =
[0.; 0.; 0.0311111111111111102; 0.0510416666666666657;
 0.0499999999999999958; 0.0347222222222222238;
 0.0166666666666666664; 0.004375; 0.; 0.; 0.]
```

Q4. 55 points Tries are important data-structure in computer science. In this exercise, we use a trie to store efficiently words by sharing of prefixes of strings. We represet a dictionary using a list of tries.

To illustrate, consider the following problem of storing the words "bee", "beer", "beef", "beefy", "bear", "beard", and "bree". We will use a trie to store these words in such a way that prefixes are shared. Every node contains one character, and every path in the trie corresponds to exactly one word. The leftmost path for example corresponds to the word "bee".



The trie is a tree, where each node contains one character and has `n` children. We use the following data-structure to represent a trie in OCaml.

```
type 'a trie = Node of 'a * ('a trie) list | Empty
```

`Empty` marks the end of a word and corresponds to • in the picture above. For a representation of the tree above using the template file provided.

Q4.1 10 points Before we begin, we implement two functions to simplify the manipulation of words:

```
string_explode : string -> char list
string_implode : char list -> string
```

`string_explode` turns a string into a list of characters and `string_implode` turns a list of characters back into a string.

To implement these two functions using a selection of the following higher-order functions : `List.map`, `List.fold_right`, `List.fold_left` and `tabulate`.

You may also find the following functions from the OCaml string and char library useful.

```
(* String.get s n returns character number n in string s. *)
String.get : string -> int -> char

(* Return the length (number of characters) of the given string.*)
String.length : string -> int

(* Return a string representing the given character *)
Char.escaped : char -> string
```

Q4.1 15 points Complete the function `insert` which allows us to insert a string `w` into a list of tries `t`, by writing a function

```
ins:char list -> (char trie) list -> (char trie) list
```

You may insert a given word multiple times, and we allow the trie to contain duplicates. Insert your new element to the left in the list.

Q4.2 15 points Complete the function `lookup` which allows us to check whether a given string `w` is in the trie `t`. For this task write a function

```
lkp: char list -> (char trie) list -> bool
```

The function `lkp char_list trie_list` returns true, if there exists a trie `t` in `trie_list` s.t. the trie `t` has a path which corresponds to `char_list`, i.e. the word corresponding to `char_list` is in the list of tries `trie_list`. The function returns false otherwise.

Q4.4 15 points Complete the function `findAll prefix trie_list` which returns a list of all words with `prefix` in `trie_list`.

```
findAll: string -> (char trie) list -> string list
```

For this task write an auxiliary function `findAll'` which takes as input a list of characters corresponding to `prefix` and a trie list. The auxiliary function `findAll'` raises an exception `Error` if no path matching the `prefix` can be found.

Here are a few hints:

- Think how you would need to modify the function `lkp` you wrote earlier, to achieve what you want.
- Use higher-order functions in appropriate places.