CS:4420 Artificial Intelligence
Spring 2018

Homework 1

Due: Friday, Feb 9 by 11:59pm

This is a programming assignment in F# to be done *individually*. Download the accompanying F# source file `hw1.fsx` and enter your solutions in it where indicated. When you are done, submit it through the Assignments section of ICON as a *plain text file* with the same name. Make sure you *write your name in that file* where indicated.

Each of your answers *must be free of static errors*. You may receive no credit for problems whose code contains syntax or type errors.

Pay close attention to the specification of each problem and the restrictions imposed on its solution. Solutions ignoring the restrictions may receive only partial credit or no credit at all.

In the following problems *do not* use any mutable variables (i.e., variables declared with `let mutable`) or any mutable predefined types such as references, mutable records, arrays, and so on. Define your functions recursively as needed and take advantage of the `match` operator.

You do not need, and *should not use*, any F# library functions unless instructed otherwise. However, you can use your own auxiliary functions if you prefer.

Do not worry about the efficiency of your solution. Strive instead for cleanness and simplicity. *Unnecessarily complicated code may not receive full credit.*

1 Programming Functional Style with Lists

Do not use the `if` *operator* in Problems 1–4 below.

1. Write a function `sum: float list -> float` which takes as input a list l of floats and returns the sum of all the elements in l.
   For example, `sum [1.4; 3.1; 3.1]` is `7.6`, while `sum []` is `0.0`.

2. Write a function `concatAll: list -> string` which, given any list l of strings, returns the concatenation of all the strings in l.
   For example, `concatAll ["un"; "be"; "liev"; "a"; "ble"]` is `"unbelievable"`.

3. Write a function `squareAll: int list -> int list` which, given an integer list l, "squares" every element in it, that is, returns a new list containing in the same order the square of each element of l.
   For example, `squareAll [1; 2; -3; 0]` is `[1; 4; 9; 0]`.

4. Write a parametric function `remove: 'a -> 'a list -> 'a list` which takes as input a value x of type $'a$ and an $'a$ list l, and "removes" all the occurrences of x from l, that is, returns a list containing, in the same order, all and only the elements of l that differ from x. For example,

   `remove 2 [2; 1; 3; 3; 2; 1]` is `[1; 3; 3; 1]`,
   `remove 5 [2; 1; 3; 3; 2; 1]` is `[2; 1; 3; 3; 2; 1]`.

5. Write a parametric function `replaceAll: 'a -> 'a -> 'a list -> 'a list` which, given values x and y of type $'a$ and an $'a$ list l, returns a list identical to l except that every occurrence of x in l is replaced by y in the new list.
   For example, `replaceAll 2 22 [2; 1; 3; 2; 1]` is `[22; 1; 3; 22; 1]`.

6. Write a parametric function `removeDuplicates: 'a list -> 'a list` which, removes all duplicates from the input list l, that is, returns a new list consisting with all elements of l, in the same order, but with no repetitions.
   For example, `removeDuplicates [4; 3; 4; 5; 5; 3; 2]` is `[4; 3; 5; 2]`.

7. Write a parametric function `pair: 'a list -> 'b list -> 'a * 'b list` which takes two lists of the same length and produces a list of pairs of corresponding elements in the two input lists.
   For example, `pair [4; 2; 5; 9] ["a"; "b"; "c"; "d"]` is
   `[(4, "a"); (2, "b"); (5, "c"); (9, "d")]`.[1]

8. Write a function `move: 'a list -> 'a list -> 'a list` which takes two lists of the same type and returns the result of inserting the values of the first list into the second, in reverse order.
   For example, `move [1; 2; 3] [7; 8]` is `[3; 2; 1; 7; 8]`.

9. Use function `move` to implement the function `reverse: 'a list -> 'a list` that returns the result of reversing its input list.
   For example, `reverse [1; 2; 3; 4]` is `[4; 3; 2; 1]`.

10. Write a function `middle: 'a list -> 'a` which takes a list with an odd number of elements and returns its middle element.
    For example, `middle [4]` is `4` and `middle ['A'; 'B'; 'C'; 'D', 'E']` is `'C'`.

2 Programming Functional Style with Trees

Consider a variant of the binary tree datatype seen in class, this time parametrized by the type of element stored in a node:

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

1. Write a parametric function `size: 'a tree -> int` which, given a tree t, returns the number of (non-empty) nodes in it.

[1] Here and later use `failwith` to raise an exception when the input does satisfy the assumptions.

2. Write a parametric function `leftmost: 'a tree -> 'a option` which takes as input an $'a$ tree t, and returns `None` if t is empty and otherwise returns `(Some` x`)` where x is the *leftmost* value in t.

   Observe that the leftmost value of a tree of the form `Node (`x`,` t_1`,` t_2`)` is the same as the leftmost value of t_1 if t_1 is nonempty and is x otherwise.

3. Write a parametric function `lreplace: 'a -> 'a tree -> 'a tree` which, given a value x of type $'a$ and an $'a$ tree t, returns a tree identical to t except that the *leftmost* value of t, if any, is replaced by x in the new tree. For example,

```
lreplace 9 (Node (3, Node (1, Empty, Empty),
                     Node (4, Empty,
                              Node (7, Empty, Empty))))
```

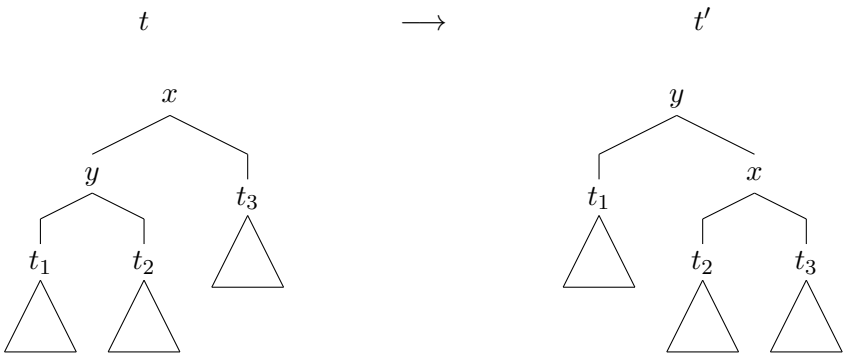
   is

```
Node (3, Node (9, Empty, Empty),
         Node (4, Empty,
                  Node (7, Empty, Empty)))
```

   The function should be such that `lreplace` x t equals t when t does not have a leftmost node.

   Hint: Use `occurs`.

4. Write a parametric function `rotateRight:'a tree -> 'a tree` which takes a non-empty tree t with a non-empty left subtree and rotates t right, that is, produces a tree t' as pictured below, where x and y denote node values and t_1, t_2 and t_3 denote subtrees.



   If the input tree t does not have the shape above, it is returned unchanged.

5. **[Optional, extra credit]** Write a parametric function `flatten:'a tree -> 'a tree` that takes a tree t and returns a tree t' such that

   (a) t' and all of its non-empty right subtrees have an empty left subtree.

(b) an in-order traversal of t' produces the same sequence of values as an in-order traversal of t.

For example,

```
flatten (Node (3, Node (1, Empty, Empty),
                  Node (4, Empty,
                           Node (7, Empty, Empty))))
```

is

```
Node (1, Empty, Node (3, Empty, Node (4, Empty, Node (7, Empty, Empty))))
```