Assignment 4

Exercise 1

Create a library called MultiSet to represent ordered multisets in F#. Remember that a library contains both a signature file and an implementation file. A multiset is a set that can contain duplicate elements, so multisets such as $\{x, y, x\}$ are perfectly valid. An ordered multiset also allows you to retrieve the elements in order of some given comparison relation. You may use whatever data type you like to store your multiset, but by far the easiest (and it will give you all ordering properties completely for free) is `Map<'a, uint32> when 'a : comparison`, where the key of the map represents elements in the multiset and the values of the map represents the number of occurrences of their corresponding keys.
So the example multiset before could be implemented using the map $[x \mapsto \mathbf{2u}; y \mapsto \mathbf{1u}]$. Other things you may consider including is the size of the multiset so that size lookups can be done in constant rather than linear time, but that is entirely optional.

A short recapitulation on unsigned integers is in order. A 32-bit unsigned integer has the type `uint32` and there literals have a `u` postfix (e.g. `42u`). Moreover F# does not directly allow operators of different types to, for instance, be added so the expression `5 + 8u` will not type check, but casting the expression to `5 + (int 8u)` or to `(uint32 5) + 8u` is perfectly valid.

For the description below, we will use the notation `{(a1, x1), ..., (an, xn)}` for a multiset with elements $a_1 \ \cdots \ a_n$ and x_i, where $x_i > 0$, represents how many occurrences of a_i there are are in the multiset. Moreover, the multiset is ordered so we know that $a_i < a_j$ for all i and j where $i < j$.

The library should contain the following functions:

`empty : MultiSet<'a>` that creates an empty multiset.

`isEmpty : MultiSet<'a> -> bool` that given a multiset `s`, returns `true` if a `s` is empty, and `false` otherwise.

`size : MultiSet<'a> -> uint32` that given a multiset `s`, returns the size of `s` (the sum of the number of occurrences of all elements in `s`).

`contains : 'a -> MultiSet<'a> -> bool` that given an element `a` and a multiset `s`, returns `true` if `a` is an element of `s` and `false` otherwise.

`numItems : 'a -> MultiSet<'a> -> uint32` that given an element `a` and a multiset `s`, returns how many occurrences of `a` there are in `s`.

`add : 'a -> uint32 -> MultiSet<'a> -> MultiSet<'a>` that given an element `a`, a number `n`, and a multiset `s`, returns the multiset that has `a` added to `s` `n` times.

`addSingle : 'a -> MultiSet<'a> -> MultiSet<'a>` that given an element `a` and a multiset `s`, returns `s` with a single instance of `a` added to it.

`remove : 'a -> uint32 -> MultiSet<'a> -> MultiSet<'a>` that given an element `a`, a number `n`, and a multiset `s`, returns the multiset with `n` occurrences of `a` from `s` removed (remember that a multiset cannot have fewer than 0 entries of an element). If `s` contains `n` or fewer occurrences of `a` then the result should contain no occurrences of `a`.

`removeSingle : 'a -> MultiSet<'a> -> MultiSet<'a>` that given an element `a` and a multiset `s` returns the multiset where a single occurrence of `a` has been removed from `s`. If `s` does not contain `a` then return `s`.

`fold : ('a -> 'b -> uint32 -> 'a) -> 'a -> MultiSet<'b> -> 'a` that given a folding function `f`, an accumulator `acc` and a multiset `{(a1,x1), ..., (an,xn)}`, returns `f (... (f acc a1 x1) ...) an xn)`. Note that the folder function `f` takes the number of occurrences of each element as an argument rather than calling `f` that number of times. It is up to the user of the `fold` method to determine how to handle multiple occurrences of the same element (this is better practice, and it makes your life easier).

`foldBack : ('a -> uint32 -> 'b -> 'b) -> MultiSet<'a> -> 'b -> 'b` that given a folding function `f`, an accumulator a multiset `{(a1,x1), ..., (an,xn)}`, and an accumulator `acc`, returns `f a1 x1 (... (f an xn acc) ...)`. The same reasoning applies to `foldBack` as `fold`.

`map : ('a -> 'b) -> MultiSet<'a> -> MultiSet<'b>` that given a mapping function `f` and a multiset `{(a1, x1), ..., (an, xn)}` returns `{(f a1, x1), ..., (f an, xn)}`. Note that the mapping function is only applied on the elements themselves, and not the number of times they occur. This is in line with how maps generally work - they change the payload, but not the size of a collection. Also note that this exercise is actually a bit trickier than usual as you cannot use the regular Map.map to solve this, as that map leaves the keys unchanged, and you are most likely using your keys to represent the elements of your multiset.

`ofList : 'a list -> MultiSet<'a>` that given a list `lst` returns a multiset containing exactly the elements of `lst`

`toList : MultiSet<'a> -> 'a list` that given a multiset `{(a1, x1), ..., (an, xn)}` returns `[a1; ..(x1 times).. a1; ...; an; ..(xn times)..; an]`

`union : MultiSet<'a> -> MultiSet<'a> -> MultiSet<'a>` that given multisets `s1` and `s2` returns the union of `s1` and `s2`. For any element `a` ocurring in `s1` or `s2`, the resulting set will contain the *maximum* number of ocurrences that `a` occurs in either `s1` or `s2`.

`sum : MultiSet<'a> -> MultiSet<'a> -> MultiSet<'a>` that given multisets `s1` and `s2` returns the sum of `s1` and `s2`. For any element `a` ocurring in `s1` or `s2`, the resulting set will contain the *sum* of the ocurrences of `a` in either `s1` or `s2`.

`subtract : MultiSet<'a> -> MultiSet<'a> -> MultiSet<'a>` that given multisets `s1` and `s2` subtracts `s2` from `s1`

`intersection : MultiSet<'a> -> MultiSet<'a> -> MultiSet<'a>` that given multisets `s1` and `s2` returns the intersection of `s1` and `s2`

Finally, override the `ToString()` method of `MultiSet` such that a multiset `{(a1, x1), ..., (an, xn)}` is converted to the string string `{(a1, #x1), ..., (an, #xn)}` where the `ToString()` method for `'a` is used to convert `a1...an` to strings.

Exercise 2

Create a library called `Dict` to represent a dictionary of, for example, the English language in F#. Remember that a library contains both a signature file and an implementation file.

For this exercise you have three options of how you want to implement your dictionary, all of which give full credit for this assignment but only two are viable for the final Scrabble project so we highly recommend one of these. Your three options are to represent dictionaries using:

- Collection of strings (lists, or sets for instance)
    - **Pros**
      Easy to implement and to do lookups (all required functions already exist in the standard library)
    - **Cons**
      Linear lookup times with respect to the number of words in the language, which is not close to efficient enough for the project.
- Trie
    - **Pros**
      Linear lookup time with respect to the size of the word being found (not the number of words in the language)
    - **Cons**
      Implementing Scrabble search strategies is a bit complicated as you need to keep track of where to start writing words, which can be far away from where anything is currenly placed.
- Gaddag - A Trie where you can look up words in a dictionary by starting in the middle of a word and work yourself to the front and then to the end of the word (described in detail below).
    - **Pros**
      Linear lookup time with respect to the size of the word being found.
      Scrabble search strategies can start on or directly adjacent to already placed words (not several squares away).
    - **Cons**
      The size of the data structure is significantly larger than that of a regular Trie.

If you are short on time, then go for the collection variant and come back to the Trie later. If you want to go for the Gaddag, start with the regular Trie as that work can be modified to a Gaddag later. You are, however, highly encouraged to start with the Trie or the Gaddag rather than a collection as it will give you a head start on the project and allow you to get early feedback. The Gaddag makes searching for words in the project significantly simpler, but is not a requirement. Jesper's own Scrabble engine Oxyphenbutazone has, for instance, not been using a Gaddag until now.

Do **one** of the exercises 2a-2c below.

Exercise 2a - Collection of strings

This exercise will give you full points for the assignment, but cannot be used for the Scrabble project

Create a library (`.fs` and `.fsi` file) called `Dict` that implements a dictionary using a collection of strings (a list, or a set for instance) and which contains the following functions:

`empty : unit -> Dict` that returns an empty dictionary.

`insert : string -> Dict -> Dict` that given a string `s` and a dictionary `dict` inserts `s` in `dict`.

`lookup : string -> Dict -> bool` that give a string `s` and a dictionary `dict` returns `true` if `s` is in `dict` and `false` otherwise.

Exercise 2b - Trie

Create a library (`.fs` and `.fsi` file) called `Dict` that implements a dictionary using a Trie as described in the lecture. Every level of your trie should contain a lookup table from characters in your word to the next level in the trie. You are free to implement your Trie using non-functional datastructurse if you wish but the lookup time for a charecter at every level of the trie must be at most $O(log\ n)$ - we suggest either `Map` or `System.Collections.Generic.Dictionary`.

Your library must contain the following four functions:

`empty : unit -> Dict` that returns an empty dictionary.

`insert : string -> Dict -> Dict` that given a string `s` and a dictionary `dict` inserts `s` in `dict`.

`lookup : string -> Dict -> bool` that give a string `s` and a dictionary `dict` returns `true` if `s` is in `dict` and `false` otherwise.

Finally, we create a function that is extremely useful for your Scrabble project as it allows you to look up words incrementally which helps when taking letters already placed on the board into consideration.

`step : char -> Dict -> (bool * Dict) option` that given a character `c` and a dictionary `dict` returns:

- `Some (b, dict')` if there is a path from `dict` to `dict'` along an edge `c` and where `b` is
  - `true` if following `c` completes a word
  - `false` otherwise
- `None` otherwise

The intuition behind this function is that it allows you to go down the trie one step at a time and check the validity of the path and if you have found a complete word or not.

Exercise 2c - Gaddag

A Gaddag is a special version of a Trie that allows users to search for word membership by starting from any position inside the word. This is acomplished by storing a word `n` times, where `n` is the length of the word, by starting from every character in the word and:

- read backwards to the beginning
- read a special character `#` that is not a character in your alphabet.
- read from where you started to the end of the word

For instance, the word `HELLO` would be stored five times in your Gaddag as follows

```
1: H#ELLO
2: EH#LLO
3: LEH#LO
4: LLEH#O
5: OLLEH#
```

A quick motivation for why this is useful can be demonstrated by the following example (this is only for motivational purposes, the actual search strategies will be coded during the project):

Assume you have the word `HELL` placed on the board and you have the letter `O` on hand. You can determine if you can use your `O` by starting anywhere within the word `HELL` on the board. For the sake of this example let's say we start at the first occurrence of `L` in `HELL`, which we denote `HE(L)L`

1. Read `L` from the dictionary (hit in row `3` and `4` above so we have a path), move left to `H(E)LL`.
2. Read `E` from the dictionary (only row `3` matches now), move left to `(H)ELL`
3. Read `H` from the dictionary, move left to `()HELL`
4. Since we are outside the word, look for `#` in the dictionary, which does exist on our path on row `3`, move to the right of where we started to `HEL(L)`.
5. Read `L` from the dictionary, move right to `HELL()`
6. Take the letter `O` from your hand, try a lookup in the dictionary and see that have a match.

Going back to the list above, you can see that we took the path `L->E->H->#->L->O` which exactly matches row `3`.

To implement the Gaddag you first have to think about how to store your special character `#`. By far the simplest is to use the Trie from Exercise 2b but instead of characters store an algebraic datatype that you create yourself with two members - either a unary contructor that takes a character as an argument, or a constructor that takes no arguments (similar to `None` or `[]`) to represent `#`.

Your library must contain the following four functions:

`empty : unit -> Dict` that returns an empty dictionary.

`insert : string -> Dict -> Dict` that given a string `s` and a dictionary `dict` inserts `s` in `dict` in all ways as described above (the word `HELLO` will be inserted five times).

`step : char -> Dict -> (bool * Dict) option` that given a character `c` and a dictionary `dict` returns:

- `Some (b, dict')` if there is a path from `dict` to `dict'` along an edge `c` and where `b` is
  - `true` if following `c` completes a word
  - `false` otherwise
- `None` otherwise

`reverse : Dict -> (bool * Dict) option` that given a dictionary `dict` returns:

- `Some (b, dict')` if there is a path from `dict` to `dict'` along an edge `#` (your special character) and where `b` is
  - `true` if following `#` completes a word
  - `false` otherwise
- `None` otherwise

You may wonder why there is no `lookup` function here, and the reason is that it is not really needed for the project (and easy enough to make if you want it) as your rarely look up words from start to end. You may, however, use the following function to test that a word has been successfully added to your dictionary.

The function `testGaddag: string -> Dict -> bool` takes a string `str` and a dictionary `dict` and returns `true` if all variants of `str` with the special character `#` inserted as described above are in `dict` and `false` otherwise. The empty string `""` is not considered to be part of any dictionary.

Note: This function uses `Dict.step` and `Dict.reverse` that you have already created.

```
let testGaddag str gdag =
    let rec lookup =
        function
        | [] -> fun _ -> failwith "This can never happen"
        | [x] -> Dict.step x
        | x :: xs ->
            Dict.step x >>
            Option.map (snd >> lookup xs) >>
            Option.flatten

    let rec lookups acc back =
        function
        | [] -> fun _ -> Some false
        | [x] ->
            lookup (x :: back) >>
            Option.map (snd >> Dict.reverse) >> Option.flatten >>
            Option.map (fun (b, _) -> acc && b)
        | x :: xs ->
```

```
            lookup (x :: back) >>
            Option.map (snd >> Dict.reverse) >> Option.flatten >>
            Option.map (snd >> lookup xs) >> Option.flatten >>
            Option.map
                (fun (b, _) -> lookups (acc && b) (x :: back) xs gdag) >>
            Option.flatten

    gdag |>
    lookups true [] [for c in str -> c] |>
    (=) (Some true)
```