Homework 1

Due Thursday, April 13, at 11:30pm

Turn in your homework via the course web page as an updated version of the `hw1.ml` text file that I have provided.

Make sure the file can be successfully loaded into the OCaml interpreter via the `#use` directive; if not you get an automatic 0 for the homework!

Recall the CS131 Academic Honesty Policy! You must list whom you discussed the assignment with at the top of your assignment, and also what other resources you used.

For this assignment, you will get practice implementing a number of relatively small functions in OCaml. As important as getting the right behavior is using the style that is natural for ML and its unique language features:

- *Never* use imperative features like assignment and loops. If you're using a construct not discussed in class or in the book, you're probably doing something bad!
- Use pattern matching instead of conditionals when it is natural to do so.
- Use local variables to avoid recomputing an expression that is needed multiple times in a function.

Except where explicitly allowed, **you may not define any helper functions**. Recursion is powerful, so make good use of it! Remember the key to solving problems with recursion: Assume the recursive call does the right thing, and then figure out how to turn that result into the overall answer you desire.

You may use any functions and operators available at the top level in OCaml (e.g., list operators like `::` and `@`), but **you may not use any OCaml library functions, i.e. functions that are accessed through a module (e.g., `List.length`) unless it is explicitly allowed in the problem description**.

Test your functions on several inputs, including corner cases -- we will be doing the same. One useful approach is *test-driven development*: write a bunch of test cases first, before you start coding, to act as a "spec" for what you need to do. You can use OCaml's `assert` operation to make a simple test suite in a file, e.g.

```
let _ = assert((add 1 []) = [1])
let _ = assert((add 1 [1]) = [1])
```

You can also share test cases with one another (e.g., on Piazza).

Now on to the assignment! I've provided a file `hw1.ml` that declares the name and type of each function that you must implement. **Make sure each of your functions retains exactly the expected name and type; otherwise you will get no credit for it.** Currently instead of defining a function, for each of the problems we simply throw an exception called `ImplementMe` which is defined at the top of the file. (We will talk about exceptions later in the course.) You should remove the `raise ImplementMe` code and replace it with a function definition.

Problem #1: Implementing sets using lists

In this problem you will use lists to implement sets. That is, we will treat a list containing no duplicate elements as a set. The order of each element in the set is irrelevant (except in the `fastUnion` problem in part d -- see below).

Though you may not define helper functions except where explicitly allowed, functions are allowed to call functions that have been defined above them in the file, as the solution to prior questions. This should come in handy.

a. Implement a function `member x s` that checks whether the item `x` is a member of the set `s`.

b. Implement a function `add x s` that returns a set that represents `s` ∪ `{x}`. Of course, the returned set should not contain duplicates.

c. Implement a function `union s1 s2` that returns a set that represents `s1` ∪ `s2`. Of course, the returned set should not contain duplicates.

d. The `union` function above takes time O(mn), where m is the size of `s1` and n is the size of `s2`. We can reduce this to O(m+n) if we assume that each input set has its elements in sorted order, from least to greatest. Under this assumption, implement a function `fastUnion s1 s2` that returns a set that represents `s1` ∪ `s2`. The returned set should also be in sorted order, from least to greatest. Of course, the returned set should not contain duplicates.

e. Now we're back to ordinary sets with an arbitrary order of elements (here and for the rest of the problems in this section). Implement a function `intersection s1 s2` that returns a set that represents `s1` ∩ `s2`. Of course, the returned set should not contain duplicates. **Do not implement this function recursively; instead the body of your function should consist of a single call to either `List.map` or `List.filter`.**

f. Implement a function `setify l` that takes a list `l` which may contain duplicates and removes all duplicates in order to produce a set of all the elements in `l`.

g. Implement a function `powerset s` which returns a set that represents the power set of `s`. That is, the returned set should contain all $2^{|s|}$ subsets of `s`. Of course, the returned set should not contain duplicates. *For this problem, you are allowed to define any helper functions that you need. You are also allowed, and encouraged, to use `List.map` and/or `List.filter` as you see fit in your solution.*

Problem #2: Fun with Functions

a. In class we saw the `filter` function, which retains only the elements of a list that satisfy a given predicate. Implement a function `partition`, which is similar except that it returns a pair of lists, where the first list contains the elements that satisfy the predicate and the second list contains the elements that falsify the predicate. The elements should retain the same relative ordering that they had in the original list.

   For example, `partition (function x -> x > 3) [1;5;4;3;2;6]` returns `([5;4;6], [1;3;2])`.

b. I know that by now you are missing good old while loops from the other languages that you know. In this problem you will implement a kind of while loop using recursion. Intuitively, `while p f x` iteratively invokes the function `f` (which represents the loop body) to create new values, starting from `x`,while the result of invoking `p` (which represents the loop guard), is true. For example,
   ```
   while (function x -> x < 10) (function x -> x * 2) 1
   ```
   returns `16`.

   More precisely, `while p f x` should return the unique value `v` such that there exists some integer `n >= 0` and values `v_0`,...,`v_n` where:

   - `x` is equal to `v_0`
   - `v` is equal to `v_n`
   - `p v` is equal to `false`
   - for each `i` between `0` and `n-1`, we have that `p v_i` equals `true`

- for each `i` between `0` and `n-1`, we have that `f v_i` equals `v_(i+1)`.

c. In class we saw the `twice` function, whereby `twice f x` computes `f(f x)`. In this problem you will implement a generalization of this function that can "raise a function `f` to a given power `n`." The behavior should be as follows:
   - `pow 0 f` returns a function `g` such that `(g v)` is equivalent to `v` for any argument value `v`.
   - `pow 1 f` returns a function `g` such that `(g v)` is equivalent to `(f v)` for any argument value `v`.
   - `pow 2 f` returns a function `g` such that `(g v)` is equivalent to `f (f v)` for any argument value `v`.
   - `pow 3 f` returns a function `g` such that `(g v)` is equivalent to `f (f (f v))` for any argument value `v`.
   - Etc.

   You may assume that the given number is nonnegative.