Homework 2

Due Monday, January 27, at 11:30pm

Turn in your homework via the course web page as an updated version of the `hw2.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 using datatypes as well as higher-order functions in ML. In a few places below, you are required to use one of the higher-order functions from the List module that we saw in class, so *pay attention to those directives or you will get no credit for the problem*. In addition you should obey our usual style rules:

- *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 wherever it is natural to do so.
- Use local variables to avoid recomputing an expression that is needed multiple times in a function.
- Similarly, avoid code duplication. If the same code is needed in multiple places, possibly with slight variations, make a helper function so that the code only has to be written once.

A few other tips:

- Create any number of helper functions as needed. It may be advantageous to make these functions local to the main function being defined, so they can refer to names bound in the enclosing function. Try to find opportunities to make use of this feature.
- Write comments where useful to tell the reader what's going on. Comments in OCaml are enclosed in `(*` and `*)`. The grader should be able to easily understand what your code is doing. One useful comment is to provide the type of any helper function that you define.
- Test your functions on several inputs, including corner cases -- we will be doing the same.

Now on to the assignment! I've provided a file `hw2.ml` that has a comment giving the name and type of each function that you must implement. **Make sure each of your functions has exactly the expected name and type; otherwise you will get no credit for it.** I've also declared some datatypes that are useful in certain problems.

Problem #1: Warm-Up

a. In class we saw the `map` function. A useful variant of `map` is the `map2` function, of type `('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`, which is like `map` but works on two lists instead of one. For example, `map2 (fun x y -> x*y) [1;2;3] [4;5;6]` is equal to `[1*4;2*5;3*6]`, which is `[4;10;18]`. Define the `map2` function using explicit recursion. You may assume that the two argument lists have the same length. *Do not use any functions from the `List` module or other modules.*

b. On the last homework you implemented the `rev` function of type `'a list -> 'a list` to reverse a list. Implement it again, but now where the entire implementation is a single call to `List.fold_right`.

c. Implement another version of list reversal, which we'll call `rev2`, but now where the entire implementation is a single call to `List.fold_left`.

d. We've seen two ways to define a two-argument function in OCaml: the arguments can either be supplied as a tuple, or they can be supplied separately through currying. For example, a function having two integer inputs that returns an integer could be written to have either the type `int * int -> int` or the type `int -> int -> int`. In different circumstances, one or the other form of function may be more convenient. It turns out that a function defined in either form can be converted to the other.

   Define a function `curry` of type `('a * 'b -> 'c) -> ('a -> 'b -> 'c)` and a function `uncurry` of type `('a -> 'b -> 'c) -> ('a * 'b -> 'c)` that perform the conversions. Note: The `->` operator is right-associative, so the types of `curry` and `uncurry` can be equivalently written as `('a * 'b -> 'c) -> 'a -> 'b -> 'c` and `('a -> 'b -> 'c) -> 'a * 'b -> 'c`, respectively.

   For example, suppose `f` has type `int * int -> int`. Then `(curry f) e1 e2` should yield the same answer as `f(e1,e2)`, for all arguments `e1` and `e2`. The `uncurry` function performs the reverse transformation.

e. Implement another variant of `map` called `mapAllPairs`, of type `('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`. It has the same type as `map2` above but applies the given function to all pairs in the *cross product* of the two lists. For example, `mapAllPairs (fun x y -> x*y) [1;2;3] [4;5]` is equal to `[1*4;1*5;2*4;2*5;3*4;3*5]`, which is `[4;5;8;10;12;15]`. Note that unlike with `map2`, the two argument lists can have different lengths. *Your implementation should not use recursion at all; instead do all the iteration over lists using higher-order functions from the `List` module.*

Problem #2: Dictionaries

A *dictionary* (sometimes also called a *map*) is a mapping from keys to values, supporting three main operations: `empty`, which returns an empty dictionary; `put`, which adds a new key-value pair to a given dictionary; and `get`, which looks up the value associated with a given key in a given dictionary. If the given key is already mapped to some value in the dictionary, then `put` should (conceptually) replace the old key-value pair with the new one. If the given key is not mapped to some value in the dictionary, then `get` should raise OCaml's pre-defined `Not_found` exception.

In this problem we'll explore three different implementations of a dictionary data structure.

a. Our first implementation of a dictionary is as an *association list*, i.e. a list of pairs. We actually saw this briefly in class and we implemented the `get` function (but called it `lookup`). Implement `empty1`, `put1`, and `get1` for association lists (we use the suffix `1` to distinguish from other implementations below). To get the effect of replacing old entries for a key, `put1` should simply add new entries to the front of the list, and accordingly `get1` should return the leftmost value whose associated key matches the given key.

b. A different way to implement a dictionary is by declaring a new datatype:

```
type ('a,'b) dict2 = Empty | Entry of 'a * 'b * ('a,'b) dict2
```

Here `dict2` is *polymorphic* over the key and value types, which respectively are represented by the *type variables* `'a` and `'b`. For example, the dictionary that maps `"hello"` to `5` and has no other entries would be represented as the value `Entry("hello", 5, Empty)` and would have type `(string,int) dict2`.

Implement `empty2`, `put2`, and `get2` for `dict2`. As above, new entries should be added to the front of the dictionary, and `get2` should return the leftmost match.

c. Conceptually a dictionary is just a function from keys to values. Since OCaml has first-class functions, we can choose to represent dictionaries as actual functions. We define the following type:

```
type ('a,'b) dict3 = ('a -> 'b)
```

We haven't seen the above syntax before (note that the right-hand side just says `('a -> 'b)` rather than something like `Foo of ('a -> 'b)`). Here `dict3` is a *type synonym*: it is just a shorthand for the given function type rather than a new type. As an example of how this representation of dictionaries works, the following "dictionary" maps `"hello"` to `5` and has no other entries:

```
(function s ->
  match s with
    "hello" -> 5
  | _ -> raise Not_found)
```

One advantage of this representation over the two dictionary implementations above is that we can represent infinite-size dictionaries. For instance, the following dictionary maps all strings to their length (using the `String.length` function):

```
(function s -> String.length s)
```

Implement `empty3`, `put3`, and `get3` for `dict3`. *It's fine if the types that OCaml infers for these functions use `('a -> 'b)` in place of `('a,'b) dict3`, since they are synonyms for one another.*

Problem #3: Calculators

a. Consider a simple calculator that accepts arithmetic expressions and computes their values. An

implementation of the calculator might *parse* the user input into a nice tree structure like the following:

```
type op = Plus | Minus | Times | Divide
type aexp = Num of float | BinOp of aexp * op * aexp
```

For example, input to the calculator like `(1.0 + 2.0) * 3.0` would be parsed into the value `BinOp(BinOp(Num 1.0, Plus, Num 2.0), Times, Num 3.0)`.

Write a function `evalAExp` of type `aexp -> float` that evaluates a given arithmetic expression. For example, the result of evaluating our expression above should be `9.0` (which OCaml prints as just `9.` without the trailing zero). You do not need to handle division-by-zero errors.

b. Some HP calculators (as well as the `dc` calculator at the terminal in Linux and Mac OS X) and some programming languages (e.g., the language of the Java virtual machine) evaluate expressions using a stack. An arithmetic computation is expressed as a sequence (i.e., a list) of stack operations, each of which is represented as a value of the following datatype:

```
type sopn = Push of float | Swap | Calculate of op
```

The operations are defined to manipulate a stack of floating point numbers (which we'll represent as a `float list`). The operation `Push n` pushes the number `n` onto the stack (thereby increasing the stack size by 1). The operation `Swap` pops the top two numbers off the stack and pushes them back on the stack in reverse order (thereby keeping the stack the same size). The operation `Calculate op` pops the top two numbers `n1` and `n2` off the stack and pushes the result of evaluating `(n2 op n1)` onto the stack (thereby decreasing the stack size by 1). *Note that the first operand in the computation is the second value popped off the stack, and the second operand is the first value popped off the stack. This makes a difference for non-commutative operations like subtraction and division. This behavior makes sense since it corresponds with the order in which the operands were originally computed (and pushed onto the stack).*

For example, the arithmetic expression `(1.0 + 2.0) * 3.0` can be represented by the sequence of stack operations `[Push 1.0; Push 2.0; Calculate Plus; Push 3.0; Calculate Times]`. This style of inputting the two operands before the operator is known as Reverse Polish notation (RPN).

Write a function `evalRPN` of type `sopn list -> float` that evaluates a sequence of stack operations. For example, `evalRPN([Push 1.0; Push 2.0; Calculate Plus; Push 3.0; Calculate Times])` should evaluate to `9.0`. You will find it useful to do most of the work in a helper function that takes an extra argument of type `float list`, which is used as a stack. Once all of the stack operations have been processed, the number at the top of the stack will be the final result value to return. You don't need to handle *stack underflow*, which occurs when the stack has too few elements to perform the next operation. (But note how the ML typechecker properly warns about this possibility!)

c. Write a function `toRPN` of type `aexp -> sopn list` that converts an arithmetic expression into a sequence of stack operations which represents the same computation. There should be a `Calculate`

`Plus` stack operation for every `Plus` in the input, and so on; evaluating the input expression to a number `n` and then returning `[Push n]` will get you no credit. *Hint: This function corresponds exactly to a postorder traversal of the input expression when viewed as a tree. Your function should not use `Swap` operations at all, but see the next problem*.

d. The same expression can be computed several ways in the stack machine, because for each subexpression, you can choose to evaluate the left side first or the right side first. (Because subtraction and division are not commutative, evaluating the right side first and then the left will require a `Swap` to fix things up.)

   For example, 1.0 - (2.0 + 3.0) can be represented either by

   ```
   [Push 1.0; Push 2.0; Push 3.0; Calculate Plus; Calculate Minus]
   ```

   or by

   ```
   [Push 2.0; Push 3.0; Calculate Plus; Push 1.0; Swap; Calculate Minus]
   ```

   The first list of commands requires a stack that can hold at least three numbers at once, whereas the second list never requires more than two numbers on the stack at any one time.

   Write a function `toRPNopt` of type `aexp -> (sopn list * int)`. The function returns a pair containing (1) an optimal sequence of commands to evaluate the given arithmetic expression; and (2) the minimum size of the stack necessary for evaluating this sequence of commands. Optimal here means requiring the smallest stack size for its evaluation.

   `toRPNopt` can be computed recursively, using the optimal command sequences for the two operands and the stack sizes they each require. You can decide which side (left or right) to evaluate first just by looking at the stack sizes each requires, without looking at their particular command sequences. By default you should evaluate the left operand first, as with `toRPN`. However, you should instead evaluate the right operand first if it will reduce the maximum stack size required. Also, in that case don't forget to insert a `Swap` command if the operation being computed is non-commutative; don't add a `Swap` command for commutative operations.