Homework 2

Due Tuesday, April 25, 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 with higher-order functions and datatypes in ML. In a few places below, you are required to implement your functions in a particular way, 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:

- Feel free to use functions from the `List` module that you find helpful.
- Create any number of helper functions as needed.
- Write comments where useful to tell the reader what's going on. Comments in OCaml are enclosed in `(*` and `*)`.
- 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.

Now on to the assignment! I've provided a file `hw2.ml` that has the name and type of each function that you must implement. **Make sure each of your functions has exactly the expected name and type (or an equivalent type); otherwise you will get no credit for it.** I've also declared some datatypes that are needed in Problem #2.

Problem #1: Vectors and Matrices

The `hw2.ml` file defines a `vector` as a *type alias* for a list of floating-point numbers. Rather than being a new type, `vector` is just a shorthand for the type `float list` -- they can be used interchangeably. Similarly, a `matrix` is a type alias for a list of vectors, with each vector representing one row in the matrix. You may assume that all rows of a matrix will have the same length.

In this problem you will implement several functions for vectors and matrices. **None of your functions, including helper functions, can be recursive. Instead, you will need to make good use of higher-order functions from the List module.** Functions are allowed to call functions that have been defined above them in the file, as the solution to prior questions.

a. Implement `vplus`, which performs vector addition. For example, `vplus [1.1;2.2;3.3] [4.4;5.5;6.6]` returns `[5.5;7.7;9.9]` (or something very close to that, due to errors in floating-point arithmetic). You may assume that the two argument vectors have the same length. *Hint: Check out `List.map2`, which will be useful here and throughout this section.*

b. Implement `mplus`, which performs matrix addition. You may assume that the two argument matrices have the same dimensions.

c. Implement `dotprod`, which computes the dot product of two vectors. You may assume that the two argument vectors have the same length.

d. Implement `transpose`, which computes the transpose of a given matrix.

e. Implement `mmult`, which performs matrix multiplication. For any call `(mmult m1 m2)` you may assume that the number of columns in `m1` is equal to the number of rows in `m2`.

Problem #2: 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 = Add | Sub | Mult | Div
type exp = Num of float | BinOp of exp * op * exp
```

For example, the expression `(1.0 + 2.0) * 3.0` would be parsed into the value `BinOp(BinOp(Num 1.0, Add, Num 2.0), Mult, Num 3.0)`.

Write a function `evalExp` of type `exp -> 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). Subexpressions within an expression should be evaluated from left to right. You do not need to handle division-by-zero errors.

b. A stack machine is a computer that executes instructions using only a stack (instead of a set of registers) for its memory. Stack machines can be implemented in a simple way and have compact instructions, so they are often the representation used for virtual machines. For example, Java programs compile to a relatively simple bytecode language that is executed on a (virtual) stack machine called the Java virtual machine.

A stack machine for a calculator might support the following set of instructions:

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

The instructions are defined to manipulate a stack of floating point numbers (which we'll represent as a `float list`). The instruction `Push n` pushes the number `n` onto the stack (thereby increasing the stack size by 1). The instruction `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 instruction `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 instructions `[Push 1.0; Push 2.0; Calculate Add; Push 3.0; Calculate Mult]`. This style of providing the two operands before the operator is known as Reverse Polish notation (RPN).

Write a function `execute` of type `instr list -> float` that implements a stack machine. For example, `execute([Push 1.0; Push 2.0; Calculate Add; Push 3.0; Calculate Mult])` 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. Starting from an empty stack `[]`, executing the above instructions one by one will cause the stack to consecutively look like `[1.0]`, `[2.0; 1.0]`, `[3.0]`, `[3.0; 3.0]`, and `[9.0]`. Once all of the stack instructions 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 the possibility of stack underflow, which occurs when the stack has too few elements to perform the next instruction. (But note how the OCaml typechecker properly warns about this possibility!)*

c. Write a function `compile` of type `exp -> instr list` that *compiles* (i.e., translates) an arithmetic expression into a sequence of stack instructions that represents the same computation. There should be a `Calculate Add` stack instruction for every `Add` in the input, and so on; evaluating the input expression to a number `n` and then returning `[Push n]` will get you no credit. **You should not use the `Swap` instruction at all (but see the last problem below).** *Hint: This function corresponds exactly to a postorder traversal of the input expression when viewed as a tree.*

d. Write a function `decompile` of type `instr list -> exp` that decompiles (i.e., translates) a list of stack instructions into an arithmetic expression that represents the same computation. Again, simply executing the stack instructions will get you no credit. You may assume that the given sequence of stack instructions does in fact correspond to a single arithmetic expression. *Hint: This function is not too different from the `execute` function above. Like that one, here you will also want to define a helper function that uses a stack, but instead of the stack holding numbers, now it will hold values of type `exp`.*

e. **EXTRA CREDIT:** Your `compile` function above uses a particular compilation strategy, but in fact there are multiple ways to compile a single arithmetic expression. In particular, for each subexpression, you can choose to compile 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` immediately afterward to preserve the behavior of the original expression.)

For example, 1.0 - (2.0 + 3.0) can be compiled to

```
[Push 1.0; Push 2.0; Push 3.0; Calculate Add; Calculate Sub]
```

but also to

```
[Push 2.0; Push 3.0; Calculate Add; Push 1.0; Swap; Calculate Sub]
```

Real compilers spend a lot of effort to *optimize* the code that they generate, typically for both time and space. In this problem, we will consider space usage, which is an especially important consideration for small embedded devices (see the Internet of Things).

Which of the above two compiled code sequences is more optimal in terms of space usage? Well, if you look carefully, the first list of instructions requires a stack that can hold at least three numbers at once, whereas the second list never requires more than two numbers to be on the stack at any one time. Therefore, the second code sequence is more optimal.

Write a function `compileOpt` of type `exp -> (instr list * int)`. The function returns a pair containing (1) an optimal sequence of instructions for the given arithmetic expression; and (2) the minimum size of the stack necessary for evaluating this sequence of instructions. "Optimal" here means requiring the smallest stack size for its evaluation.

`compileOpt` can be computed recursively, using the optimal instruction sequences for the two operands and the stack sizes they each require. You can decide which side (left or right) to compile first just by looking at the stack sizes each requires, without looking at their particular instruction sequences. By

default you should compile the left operand first, as with `compile`. However, you should instead compile 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, since it's unnecessary there.