Exercise 9.15

You should now exploit the concept continuation in connection with the compilation of expressions to lists of pocket calculator instructions addressed in Exercise 6.8. You should make a *backwards generation* of the instructions using a continuation c that represents the instructions immediately following the code that is currently being generated. Therefore, c has type `Instruction list` and the type of the translation function is:

```
transC: (Fexpr*float) -> Instruction list -> Instruction list
```

A translation of the expression `Const` r using continuation c could, for example, be

$$\texttt{transC}\ (\texttt{Const}\ r, x)\ c = (\texttt{PUSH}\ r) :: c$$

Your solutions to the following questions should not be tail-recursive functions. The task is to exploit the concept continuation in achieving optimized stack-machine instructions.

1. Give an F# implementation of `transC` that implements backwards generation of instructions for the pocket calculator.
2. The continuation should now be used to make certain optimizations at compile time. For example, the following optimized translation

   $$\texttt{transC}\ (\texttt{Const}\ \texttt{0.0}, x)\ (\texttt{ADD} :: c) = c$$

   is correct because adding 'something' to zero is that 'something'. Make a revised version of `transC` that implements this and similar optimizations concerning subtraction, multiplication and division. Hint: use an auxiliary function that addresses the situation where a constant a is added in front of the continuation c.
3. The setting for the pocket calculator is so simple that the optimization can be further refined so that the entire computation is done during the translation. For example

   $$\texttt{transC}\ (\texttt{Mul(Add(Const 3.0, X), Sub(Const 4.0, X))},\ \texttt{2.0})\ \texttt{[]}\ =\ [\texttt{PUSH 10.0}]$$

   Refine your auxiliary function from Question 2 to achieve this.

The ideas for this exercise originate from Chapter 12 of the book [12] by Peter Sestoft. The chapter describes how thinking in terms of continuations helps in making a locally optimizing compiler that translates from “micro-C” to stack-machine code. Considered optimization include optimizing code for expressions, for jumps and for tail calls.