```
// ----------------------------------------------------------------
//        Homework 3
//        Adam Levy, Alejandro Palacios, & Alicia Rodriguez
//        November 2, 2016
// ----------------------------------------------------------------


// 1. Given vectors u = (u1, u2,..., un) and v = (v1, v2,..., vn), the inner
// product of u and v is defined to be u1*v1 + u2*v2 + ... + un*vn. Write a
// curried F# function inner that takes two vectors represented as int lists
// and returns their inner product:

(*
   > inner [1;2;3] [4;5;6];;
   val it : int = 32
*)
// (Assume that the two lists have the same length.)

// Solution

(*
   If the first list is empty, 0 is returned. The recursive call multiplies the
   heads of both lists together and then calls the function on the tails of the lists.
*)
// Without pattern matching
let rec inner xs ys = if xs = [] || ys = [] then 0
                      else List.head xs * List.head ys + inner (List.tail xs) (List.tail ys)

// With pattern matching
let rec inner2 u v =
  match (u, v) with
  | ([], []) -> failwith "empty lists have no product"
  | ([u], [v]) -> u * v
  | (u::us, v::vs) -> (u * v) + inner2 us vs
```

This version is better, except that the base case should be
|([], []) -> 0

```
// 2. Given an m-by-n matrix A and an n-by-p matrix B, the product of A and B
// is an m-by-p matrix whose entry in position (i,j) is the inner product of
// row i of A with column j of B. For example,
(*
                  / 0 1 \
   / 1 2 3 \  *  | 3 2 |  =  /  9 11 \
   \ 4 5 6 /     \ 1 2 /      \ 21 26 /
*)
// Write an uncurried F# function to do matrix multiplication:
(*
   > multiply ([[1;2;3];[4;5;6]], [[0;1];[3;2];[1;2]]);;
   val it : int list list = [[9; 11]; [21; 26]]
*)
// Assume that the dimensions of the matrices are appropriate.
// Hint: Use transpose (from Homework 2), inner, and List.map.

// Solution

let rec transpose = function
| [] -> failwith "cannot transpose a 0-by-n matrix"
| []::xs -> []
| xs -> List.map List.head xs :: transpose (List.map List.tail xs)

(*
   The base case returns the empty list when multiplying by an empty matrix.

   The recursive case will first transpose the second list so that it has the same
   number of rows as the first matrix.

   It then takes the first row of the first list and does the inner operation on
   the rows of zs (columns of ys) which will. This will calculate the first row
   of the resulting matrix.

   It is then consed onto the recursive call of the second function which is done
   on the next row of the first matrix.
*)
let rec multiply = function
| ([], ys) -> []
| (x::xs, ys) -> let zs = transpose ys
                 List.map (inner x) zs :: multiply (xs,ys)
```

This works, but it is inefficient to compute (transpose ys) in every recursive call.

```
// 3. Two powerful List functions provided by F# are List.fold and
// List.foldBack. These are similar to List.reduce and List.reduceBack, but
// more general. Both take a binary function f, an initial value i, and a
// list [x1;x2;x3;...;xn]. Then List.fold returns
(*
   (f ... (f (f (f i x1) x2) x3) ... xn)
*)
// while List.foldBack returns
(*
   (f x1 (f x2 (f x3 ... (f xn i) ... )))
*)
// In spite of this complicated behavior, they can be implemented very simply:
(*
   > let rec fold f a = function
       | []    -> a
       | x::xs -> fold f (f a x) xs;;

   val fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

   > let rec foldBack f xs a =
         match xs with
         | []    -> a
         | y::ys -> f y (foldBack f ys a);;

   val foldBack : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
*)
// (Note that they don't take their arguments in the same order.)
// Each of these functions can be used to implement flatten, which "flattens"
// a list of lists:
(*
   let flatten1 xs = List.fold (@) [] xs

   let flatten2 xs = List.foldBack (@) xs []
*)
// For example,
(*
   > flatten1 [[1;2];[];[3];[4;5;6]];;
   val it : int list = [1; 2; 3; 4; 5; 6]
*)
// Compare the efficiency of flatten1 xs and flatten2 xs, both in terms of
// asymptotic time complexity and experimentally. To make the analysis
// simpler, assume that xs is a list of the form [[1];[2];[3];...;[n]].
```

```
// Solution

//Test inputs
  let listOfLists xs = List.map (fun n -> n :: [] ) xs;;
  let ys = listOfLists [1..50000];;

//Given functions
  let flatten1 xs = List.fold (@) [] xs;;
  let flatten2 xs = List.foldBack (@) xs [];;

//Analysis of flatten 1:

    //Asymptotic time complexity analysis:
      (*
        let rec fold f a = function
          | []    -> a
          | x::xs -> fold f (f a x) xs;;


        To begin, the length of a will be N and the length of the list passed in (we will call it ys) is of
length M

        1.) Amount of recursive invocations.

            The amount of recursive invocations will be M because each call will be done on the tail
of ys
            (O(M) time complexity).

        2.) Cost of each invocation directly

            The cost of each invocation directly will be O(N) because the append of A with the head
of ys will be
            done before the function is called again.

        O(M) invocations * O(N) cost = O(M * N) total time complexity.
        This is not good because although the length of M is reduced every time, the length of N
increases.
        The experimental analysis shows just how slow this operation can get.

                                                The size of the input is n, so your formulas can use only n.
                                                The answer here should be O(n^2).
      *)

    //Experimental analysis:
```

```
(*
  flatten1 (listOfLists[1..50000]);;
  Real: 00:00:14.862, CPU: 00:00:14.859, GC gen0: 3499, gen1: 3499, gen2: 0
*)
//Analysis of flatten 2:

  //Asymptotic time complexity analysis:
  (*
      let rec foldBack f xs a =
      match xs with
      | []    -> a
      | y::ys -> f y (foldBack f ys a);;


    1.) Amount of recursive invocations.

        The amount of recursive invocations is M where M is the length of ys
        (O(M) time complexity).

    2.) Cost of each invocation directly

        The cost of each invocation directly is O(M) because the head of the list ys
        is appended to the result of the recursive call.
        Even though it is always the head of the list which is just 1 element,
        that is still some length m - n which is generalized into O(M).

    O(M) invocations * O(M) calls = O(M^2) time complexity
    This is far better than flatten 1 because the parameters passed into append are not
    getting bigger after each call.

  *)

  //Experimental analysis:
  (*
    flatten2 (listOfLists[1..50000]);;
    Real: 00:00:00.007, CPU: 00:00:00.015, GC gen0: 1, gen1: 1, gen2: 1
  *)
```

Does not make sense.

No, flatten2 takes O(n) time.

```
// 4. An interesting higher-order function is twice, which can be defined by
(*
   > let twice f = (fun x -> f (f x));;

   val twice : ('a -> 'a) -> 'a -> 'a
*)
// or, using F#'s function composition operator <<, by
(*
   > let twice f = f << f;;

   val twice : ('a -> 'a) -> ('a -> 'a)
*)
// If we also define
(*
   > let successor n = n+1;;

   val successor : int -> int
*)
// then we can evaluate expressions like
(*
   > (twice (twice (twice (twice successor)))) 0;;
   val it : int = 16
*)
// It is pretty easy to see that with k occurrences of twice, these
// expressions will return 2k.
// Remarkably, F# also allows us to evaluate expressions like
(*
   twice twice twice twice successor 0
*)
// in which the function applications are associated to the left, by F#'s
// default parsing conventions. (Notice that this means that twice gets
// applied to itself!) Can you figure out a formula that gives the value of
(*
   twice twice twice ... twice successor 0
*)
// when there are k occurrences of twice? (I suggest that you approach this
// problem experimentally.)

// Solution

   (*
      To begin, the function was runned when k = 1,2,3,4
      this gave the values 2,4,16,65536 respectively.
```

I found that aside from when k = 1, the resulting numbers could be square rooted to give integers.

I created a temporary pattern that would get me closer to the answer:
(2^1), (2^2), (4^2), (256^2)

To further expound upon this temporary pattern, I found the roots of the cases where the would result in integers:
(2^1), (2^2), ((2 ^ 2) ^ 2), ((((2^2)^2)^2) ^ 2)

It became clear that the variable amount of times you must raise 2 to the power of 2 is k - 1 times. The final power of 2 stays the same.

The pattern should resolve to (2 ^ (2^k-1)) ^ 2.

This resolved the initial issue of k = 1 because it would result in sqrt(2) ^ 2 which would simply be 2.

*)

No, you did not get the right pattern. When k=4, the result is actually

2^2^2^2

(which associates to the right).

```
// 5. Recall our discussion of infinite streams in F#, with definition
(*
   type 'a stream = Cons of 'a * (unit -> 'a stream)
*)
// Show how to define map f s on streams; this should give the stream formed
// by applying function f to each element of stream s.

// Solution

 type 'a stream = Cons of 'a * (unit -> 'a stream);;

 //Used as test
 let rec upfrom n = Cons(n, fun () -> upfrom(n+1));;

 (*
    This function takes as input some function f and a type of cons.
    It will give a type cons where the type n is applied to the function.
    The function type in cons will return a function which will apply some
    function f to the result of the function passed in from the type cons.

 *)
 let rec mapstream f (Cons(n, s)) = Cons(f n, fun () -> mapstream f (s()) );;
```

// 6. Interpreter 0 In this problem, we begin our exploration of the use of F#
// for language-oriented programming. You will write an F# program to
// evaluate arithmetic expressions written in the language given by the
// following context-free grammar:

```
(*
   E -> n | -E | E + E | E - E | E * E | E / E | (E)
*)
```

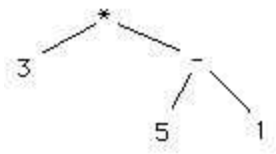
// In the above, n is an integer literal, -E is the negation of E, the next
// four terms are the sum, difference, product, and quotient of expressions,
// and (E) is used to control the order of evaluation of expressions, as in
// the expression 3*(5-1).
// Rather than working directly with the concrete syntax above, we will
// imagine that we have a parser that parses input into an abstract syntax
// tree, as is standard in real compilers. Hence your interpreter will take
// an input of the following discriminated union type:

```
(*
   type Exp =
     Num of int
   | Neg of Exp
   | Sum of Exp * Exp
   | Diff of Exp * Exp
   | Prod of Exp * Exp
   | Quot of Exp * Exp
*)
```

// Note how this definition mirrors the grammar given above. For instance,
// the constructor Num makes an integer into an Exp, and the constructor Sum
// makes a pair of Exp's into an Exp representing their sum. Interpreting
// abstract syntax trees is much easier than trying to interpret concrete
// syntax directly. Note that there is no need for a constructor
// corresponding to parentheses, as the example given above would simply be
// represented by

```
(*
   Prod(Num 3, Diff(Num 5, Num 1))
*)
```

// which represents the parse tree which looks like



// Your job is to write an F# function evaluate that takes an abstract syntax

```
// tree and returns the result of evaluating it. Most of the time, evaluating
// a tree will produce an integer, but we must address the possibility of
// dividing by zero. This could be handled by raising an exception, but
// instead we choose to make use of the built-in F# type
(*
   type 'a option = None | Some of 'a
*)
// Thus evaluate will have type Exp -> int option, allowing it to return Some
// m in the case of a successful evaluation, and Nonein the case of an
// evaluation that fails due to dividing by zero. For example,
(*
   > evaluate (Prod(Num 3, Diff(Num 5, Num 1)));;
   val it : int option = Some 12
   > evaluate (Diff(Num 3, Quot(Num 5, Prod(Num 7, Num 0))));;
   val it : int option = None
*)
// Naturally, evaluate e should use recursion to evaluate each of e's
// sub-expressions; it should also use match to distinguish between the cases
// of successful or failed sub-evaluations. To get you started, here is the
// beginning of the definition of evaluate:

// Solution

    type Exp =
     Num of int
   | Neg of Exp
   | Sum of Exp * Exp
   | Diff of Exp * Exp
   | Prod of Exp * Exp
   | Quot of Exp * Exp



  (*
   This function recursively evaluates some sequence of calls of type Exp
   and returns a type int option.

   If the Exp is Neg, the negation must be returned.

   The other types (except for Quot) can all be done in a similar fashion, so the pattern
   matched for Sum will be used.

   Consider the following value Sum(e,s) where e and s are type Exp
```

```
  First, e is resolved locally to an integer value by pattern matching.

  Then, pattern matching is used on s. If the type is None, None is returned.
  Otherwise an int option will be returned with the appropriate operation done.

  For Quot, it is somewhat similar except you cannot divide by 0.
  if the int option contains 0 or is None, None is returned. Otherwise the appropriate
  operation is done.

  Note that in the local declaration, None is matched with 0. This is to prevent
  an error from an erroneous input such as:
  evaluate (Sum(Quot(Num 2,Num 0),Num 2));;
  It can be modified to result in an error.

*)
let rec evaluate = function
| Num n -> Some n
| Neg e -> match evaluate e with
          | None -> None
          | Some n -> Some (n * -1)
| Sum (e,s) ->  let e = match evaluate e with
                        | Some e -> e
                        | None -> 0
                match evaluate s with
                | None -> None
                | Some n -> Some (e + n)
| Diff (e,s) -> let e = match evaluate e with
                        | Some e -> e
                        | None -> 0
                match evaluate s with
                | None -> None
                | Some n -> Some (e - n)
| Prod (e,s) -> let e = match evaluate e with
                        | Some e -> e
                        | None -> 0
                match evaluate s with
                | None -> None
                | Some n -> Some (e * n)
| Quot (e,s) -> let e = match evaluate e with
                        | Some e -> e
                        | None -> 0
                match evaluate s with
```

Your syntax could be greatly improved:

```
| Sum(e,s) -> match (evaluate e, evaluate s) with
                        | (Some n1, Some n2) -> Some (n1+n2)
                        | _ -> None
```

```
| None   -> None
| Some 0 -> None
| Some n -> Some (e / n)
```