Exercise 2: Recursion, Tail Recurison, and Pattern Matching

This exercise is an opportunity to practice writing functions on these using structural pattern-matching, tail recursion, and pattern matching.

For this and future exercises, remember that:

- You may not import any libraries or modules unless explicitly told to do so.
- You may not use the function "eval" for any exercise or assignment unless explicitly told otherwise.
- You may not use any iterative or mutating functionality unless explicitly allowed. Remember that a big goal of this course is to learn about different models and styles of programming!
- You may write helper functions freely; in fact, you are encouraged to do so to keep your code easy to understand.
- We generally provide a set of tests for each exercise or assignment, but these tests are meant to help with understanding expected function behaviour. Please be warned that they are not nearly comprehensive enough to give confidence about correctness.

Breaking any of these above rules can result in a grade of 0.

- Code that cannot be imported (e.g., due to a syntax error, compilation error, or runtime error during import) will receive a grade of zero! Please make sure to run all of your code before your final submission, and test it on the Teaching Lab environment (which is the environment we use for testing).
- The `(provide ...)` code in your exercise file is very important. Please follow the instructions, and don't modify those lines of code! If you do so, your code will not be able to run and you will receive a grade of zero.

Starter code

Download these files from Quercus files.

- `Exercises/ex2/ex2.rkt`

Submission instructions

Submit the file `ex2.rkt` to MarkUs: https://mcsmark.utm.utoronto.ca/csc324f20/

But before you submit, check that:

- Your code does not have any syntax error. If you only completed part of the exercise, replace any possibly syntactically incorrect code with the starter code.
- The `(provide ...)` code in your Racket file is left intact. Otherwise we won't be able to run our tests!

Task 1: Practicing with higher-order list functions [0 pt]

Review the code you wrote for `num-evens` and `num-many-evens` on Exercise 1. The structure should look basically identical, except one part: the predicate (boolean function) you are using to check the list elements. This sort of repeated functionality begs to be abstracted, and (so far) we only have one tool for code abstraction: writing a function, with parameters for the parts of functionality that change. In this case, the part we want to change is the predicate—this means we'll need to pass these predicates directly to our generalized function. In other words, we'll need to write a function that *takes another function as an argument.* This is a powerful idea. In functional programming, functions are just another type of value, and so if other values like numbers and strings can be passed as arguments, then so can functions.

1. To check that you understand this, write a function `num-pred`, which takes a (unary) boolean function and a list, and returns the number of items in the list that make the function return `#t`.

   We call `num-pred` a *generalization* of `num-evens` and `num-many-evens` because we should be able to implement both of the latter two functions using `num-pred`, just by passing in the right predicate as an argument. For example:

```
(test-equal? "num-pred/num-evens"
  (num-evens (list 1 2 3 4 5))
  (num-pred even? (list 1 2 3 4 5)))

(test-equal? "num-pred/num-many-evens"
  (num-many-evens (list (list 1 2 3 4 5) (list 2 4 6 10))
  (num-pred ...    (list (list 1 2 3 4 5) (list 2 4 6 10))))
```

2. In the second test above, replace the `...` with either a lambda expression or a helper function so that the test passes. That is, we want you to understand how to *really* generalize `num-many-evens` to `num-pred`.

Task 2: More Higher-Order Function [3 pt]

Complete the functions `make-counter`, `apply-fn`, `sum-map-both` and `sum-map-both-helper` according to their documentation. For these functions, you may change the function header to use `define/match` instead of `define` if you wish to use pattern-matching.

The function `make-counter` is an alternative approach to abstracting `num-pred`. Suppose we have `num-pred`, and use it to count the number of even elements in several different lists:

```
(num-pred even? (list 1 2 3 4 5))
(num-pred even? (list))
(num-pred even? (list 100 300))
(num-pred even? (list 1 2 3 4 5 6 7 8 9 10))
(num-pred even? (list -1 -2 -3 -4 -5 -6 -7))
```

Instead of repeating the `num-pred even?` part over and over, we can define the following function:

```
(define (num-evens lst)
  (num-pred even? lst))
```

Of course, we could do the same for `num-many-evens`:

```
(define (num-many-evens lst)
  (num-pred ... lst))
```

This is actually a fairly standard idea in functional programming: if you have a multi-parameter function in which one (or more) parameters are frequently passed on the same value (e.g., passing `even?` for `pred`), we can define a new function that *fixes* some of the arguments to the old function, and takes in only the remaining ones. (This is very much related to the idea of *currying*, which we discuss in lecture.)

But if you look at these two definitions for `num-evens` and `num-many-evens`, and consider taking the same approach for defining, say, a `num-odds` or `num-greater-than-3`, there is still some repetition: `lst`. Even though `num-pred` has allowed us to abstract away the `(if ... (+ 1 ...) ...)` part, it hasn't abstracted the process of function creation itself.

In other languages, we might be forced to give up here; but because Racket functions are first-class values, there is nothing stopping us from defining a function that *returns another function.*

The ability of a function to return other functions is one of the hallmarks of functional programming; it opens up new opportunities for abstraction, but raises some interesting questions when it comes to *how* these things are actually implemented.

Task 3: Calculator [7 pts]

Over the course of the next few weeks, we will build a calculator that will evaluate **binary arithmetic expressions**. The features supported by our language will grow over the next exercises. For now, our binary arithmetic expressions will have the following grammar:

```
Binary Arithmetic Expression Grammar

<expr> = NUM | (<op> <expr> <expr>)
```

```
<op>   = + | - | * | /
```

where `NUM` represents any integer literal in base 10, and the arithmetic operations `+`, `-`, `*`, `/` have the standard mathematical meanings.

Arithmetic expressions will be represented as a list of elements in Racket, and operators will be represented using symbols. For example, `(list '+ 2 3)` and `(list '/ 8 2)` are examples of valid expressions.

Note that with the way quoting distributes in Racket, `'(+ 2 3)` represents the same data structure as `(list '+ 2 3)`. Likewise, `'(+ 2 (+ 4 5))` represents `(list '+ 2 (list '+ 4 5))`. In other words, you should *not* nest the quote operator `'`.

Your tasks is to write a function `calculate` that returns the value of an expression.

In addition for correctness, we will be checking for the implementation:

- You *must* use pattern matching for this function, so that you have some practice using pattern matching in Racket.
- You must correctly use structural recursion.
- 1 point (out of 7) will be allocated to your programming style (e.g. are you using more cases than you need to? Is your function as simple as it can be?)

Hint: You might find the Racket built-in functions `number?` and `list?` helpful.