CPS506 Lab 2

Elixir: Lists and Tail Recursion

Preamble

Powerful built-in functions for list operations are a hallmark of functional programming languages. In this lab you'll explore the Elixir interactive shell and write several Elixir functions that operate on lists. Every problem in this lab can be very easily solved with explicit looping or branching. However - we're trying to experience functional programming through its best practices rather than simply writing imperative-style code using a functional language. This means getting very good at working with lists and writing recursive functions!

Getting Started

Create an Elixir source file called **`ElixirLab.ex`**. In this file you will define a module named **`ElixirLab`**. All functions written for this lab will be placed in this single module. Make sure each of your functions is named precisely according to the specifications below. Alongside this file, you may (should) create another Elixir script file (*.exs). In this script you can write a series of test calls to your functions to make sure they work correctly.

Part 1: (Relatively) Simple List Operations

Write an Elixir function in your ElixirLab module to solve each of the problems below. Each function will accept a list as input and return either a Boolean value or another list, depending on the question. For this part of the lab, you may ***not*** use branching or repetition of any kind. No recursion, no if/else, no indexing. Head and tail will serve you well here.

i) **`firstTwo` –** Returns True if the first two elements are the same, False otherwise.

ii) **`evenSize` –** Returns True if the list has an even number of elements, False otherwise.

iii) **`frontBack` –** Removes the first element of the input list and appends it to the back. Return the new list.

iv) **`nextNineNine` –** Insert the integer 99 in the second position of the list.

v) **`isCoord` –** Return True if the input list could represent a coordinate, False otherwise. A coordinate has two elements, both are numbers.

vi) **`helloIfSo` –** If the input list contains the string "Hello", remove it and place it at the end of the list. If it does not contain "Hello", add it to the end of the list.

Part 2: Tail Recursion

For the problems below, you will implement tail recursive functions. You must implement your own tail recursive functions. No built-in shortcuts, no indexing, no iteration. Your functions must call themselves in some recursive fashion. Unless otherwise specified, input lists may contain any combination of types, and be of arbitrary length. Assume nothing!

vii) **`sumEven`** **–** Return the sum of all even *integers* in an input list.

viii) **`sumNum`** **–** Returns the sum of all *numeric* values in the list.

ix) **`tailFib`** **–** accepts an integer argument, **`n`**, and returns the nth Fibonacci number. Assume the first two Fibonacci numbers are 1 and 1. That is, `tailFib(1)` == 1, `tailFib(2)` == 1. There is no `tailFib(0)`. If the function is called with argument less than or equal to 0, return the atom **`:error`**. Your tail-recursive implementation must avoid the branching recursive call. ***No O(2^n)!***

x) **`reduce`** **–** Consider MyEnum.reduce from this week's lecture (you can find it in the slides). Your ElixirLab.reduce will build on this. You will add an implementation that handles the optional 3rd argument to initialize acc. For example – the following two examples, exactly as written, should work correctly:

```
ElixirLab.reduce([1, 2, 3], fn(x, acc) -> x+acc end)
ElixirLab.reduce([1, 2, 3], 10, fn(x, acc) -> x+acc end)
```

You may use helper functions if you wish, so long as the user can invoke your function as seen above.

Submission

Labs are to be completed and submitted *individually*. Submit your **ElixirLab.ex** source file containing the ElixirLab module and all completed functions on D2L, under the submission for Lab 2: