Programming languages: Erlang higher order functions - problem set

Introduction

These problems allow one to gain experience in making use of higher order functions in Erlang.

Solution constraints for this exercise

- For all functions, assume that only valid inputs are given (you don't need to do error checking).
- You should not make use of any functions not already presented in this lab.

Getting started

Use the command below to download a file named *erl3.erl* that we will use to hold all of our function definitions.

```
wget http://www.cs.wfu.edu/~turketwh/231/erl3/erl3.erl
```

Open that file for editing using an editor of your choice.

Problems to gain experience with higher order functions

For each problem below, your answers should consist of functions applying a *higher order function* (either *map*, *reduce*, or *filter*) to an auxiliary function you build yourself which is applied over the elements of the list. So, in these solutions, you will need to build at least two functions (sometimes more) for each solution – one of the two should be the auxiliary function doing the real work (testing a predicate for *filter*, implementing a binary operation for *reduce*) and the second being the function that makes the call to *map*, *reduce*, or *filter*, passing in the auxiliary function as the first parameter and the list of interest as the second parameter.

You may want to go back to the first Erlang notes to review some of the math operators and how to interact with characters ($charName) and strings (view strings as lists of ASCII numbers).

Problems to go over as a class

1. (Example of *map*) Write a function named *flipSigns* that, given a list of double values as input, returns a list containing the elements of the original list, but with the sign flipped for each element – that is, turn all negative numbers into their corresponding positive number, and vice-versa. Remember for each of these, you'll likely need to write a *worker* function that can be passed to *map*. An appropriate worker function might be named *flipSign* (singular) and have an implementation that flips the sign of any single number.

```
erl3:flipSigns([]).
[]
erl3:flipSigns([1]).
[-1]
erl3:flipSigns([2,-3]).
[-2,3]
```

2. (Example of *filter*) Write a function named *extractEvens* that, given a list of integers as input, returns a list containing the elements of the input list that are *even.*

```
erl3:extractEvens([]).
[]
erl3:extractEvens([2]).
[2]
erl3:extractEvens([1,2]).
[2]
erl3:extractEvens([2,1,4]).
[2,4]
```

3. (Example of *reduce*) Write a function named *largest* that, given a list of double values as input, finds the maximum value in the list.

```
erl3:largest([2.0]).
2.0
erl3:largest([2.0,5.0]).
5.0
erl3:largest([5.0,2.0]).
5.0
erl3:largest([2.0,1.0,4.0,3.0]).
4.0
```

Problems to work alone or in small groups

1. (*filter*) Write a function named *extractAWords* that, given a list of strings as input, returns a list composed of the elements of the input list that begin with the character a or A.

The extra *io:format* code below is to support showing strings and characters as strings and characters instead of integers.

```
io:format("[~p]~n",erl3:extractAWords(["and"])).
["and"]
io:write(erl3:extractAWords(["boy"])).
[]
io:format("[~p,~p,~p]~n",erl3:extractAWords(["Acres", "boy", "and", "at"])).
["Acres","and","at"]
```

2. (*reduce*) Write a function named *logicalOr* that, given a list of Boolean values as input, computes the logical OR of the values in the list. Remember that OR, when given two parameters, is true if one or the other or both parameters is true. Accordingly, we will define OR of multiple logical expressions as true if any of the expressions is true.

```
erl3:logicalOR([true,true,true]).
true
erl3:logicalOR([false,false,false]).
false
erl3:logicalOR([true,false]).
true
```

3. (*filter*) Write a function named *longStrings* that, given a list of strings as input, returns a list composed of the elements of the input list that are greater than 3 characters long. You can use the *length* function on lists to see how many elements exist in the list.

```
io:format("[~p,~p]~n",erl3:longStrings(["abc", "abcd", "ab", "a", "abcde"])).
["abcd","abcde"]
io:format("[~p]~n",erl3:longStrings(["abcd"])).
["abcd"]
```

4. (*map*) Write a function named *truncateWords* that, given a list of strings as input, returns a list where each string in the input list is truncated in a manner such that it is no more than four characters long. That is, delete the fifth and subsequent characters while leaving shorter strings alone. [This is more challenging than the previous problem, and may require more than 2 functions; also, strings can be manipulated as if they were lists].

```
io:write(erl3:truncateWords([])).
[]
io:format("[~p,~p,~p]",erl3:truncateWords(["short","longword","longword"])).
["shor","long","long"]
io:format("[~p]",erl3:truncateWords(["t"])).
["t"]
io:format("[~p,~p,~p,~p,~p]",erl3:truncateWords(["mix","of","longish","and","shortish"])).
["mix","of","long","and","shor"]
```