TCSS 380, Winter 2020, Lab 6

The objective of this lab assignment is to give you more practice with Erlang recursive types, list library, first-class functions, and concurrency.

You must work with a partner during the lab. Find a partner of your own choosing – it could be a person you worked with before. If you cannot find a partner, ask the lab instructor for help. Before the end of the lab submit all the parts you complete during the lab session – one submission per partnership. Remember to list both names at the top of each file on which you work together. At the very least half of the lab must be completed during the lab session to receive full credit. If you do not complete some part within the two-hour session, finish at home on your own or with your lab partner by the designated due date. If you complete this part of the lab on your own, list your name only. If you complete this part with your lab partner, list both names and submit only once per partnership. All together, you need to upload:

- lab06.erl
- bst.erl
- cooperate.erl
- screenshot of running functions in the shell (each function call should appear in this screenshot)

In the text editor of your choosing create a file **lab06.erl** and write function definitions described below – make sure you use the exact same names as indicated in the description. Test all functions in the shell.

1. function *removeNonPerfects* that takes a list of numbers and returns the copy of this list with only perfect squares remaining; a perfect square is an integer that can be expressed as the product of two equal integers, e.g. 9 can be expressed as the product of 3 x 3; so if this function is called with [1,2,3,5,6,7,8,9,25,36], the resulting list should contain [1,4,9,25,36]
   To determine whether a number is a perfect square use the following helper function:

```
isPerfectSquare(X) ->
    if
         X < 0 -> false;
         true ->  Sqrt = math:sqrt(X),
                  (Sqrt - trunc(Sqrt)) == 0
    end.
```

   When using this function in *removeNonPerfects*, use the case expression – it is not mandatory to do so but it will be easier to write the matching statement using the case expression. You are NOT allowed to use built-in functions or list comprehension
2. function *flippedDiff* that takes one argument; if the value of this argument is letter *a*, then the function returns a function that takes two values and subtracts the second argument from the first argument (as in first – second); if the value of the *flippedDiff* argument is letter *b*, it returns a functions that takes two values and subtracts the first argument from the second one (as in second – first)
3. function *mapFilter* that takes two functions (the first will be used in a map, the second one in a filter) and a list, and returns a list calculated by first applying a map to the list, then applying a filter to the list returned from the map; use map and filter defined in the list library; for example if the first function multiplies each element by 3 and the second one checks for an even number, then the list [3,6] should result in [9,18] after sending it to the map, and in [18] after sending it to the filter
   - when calling the function, use the following two functions for arguments: an anonymous function that multiplies by 3 for the first argument and an anonymous function that checks if the number is even for the second argument (show this call in your screenshot)
   - add function `triple(N) -> 3 * N` to your **lab06.erl** and then call function *mapFilter* again, this time passing *triple* as the first argument (show this call in your screenshot; if you don't remember how to pass an existing function, take a look at the map example discussed during the lectures)
4. function *foldResult* that takes a function (that will be used in a fold), an accumulator, and a list; the

function returns a tuple consisting of three elements: (a) the result of folding the list from the left, (b) the result of folding the list from the right, and (c) a boolean value indicating if both results are the same or not; for example, if this function is called with a function that calculates the sum, 0, and [1,2,3,4,5], it should return { 15, 15, true}; if it is called with a function that concatenates strings, "", and ["abc", "de", "f"], it should return {"abcdef", "fdeabc", false}. Show both of these calls in your screenshots as well (hint: use ++ for string concatenation function).

Download the file **bst.erl** discussed during the lectures and add the following functions. Test them in the shell:

5. function *isFull* that takes a tree and returns true, if the tree is a full binary tree (every node but the leaves has exactly two children), false otherwise; in the shell, construct a sample tree before testing this function
6. function *fullCount* that takes a tree and returns a tuple consisting of a boolean value that indicates if the tree is full (you can call the function you defined above to determine this), and if the tree:
    - is not full, then the second element of the tuple is an atom *undefined*
    - is full, then the second element of the tuple is the number of nodes in the tree; the number of nodes in the tree is to be calculated using an inner function

In the text editor of your choosing create a file **cooperate.erl** – in this file you will encode producer-consumer model.

7. Start by writing a parent function *go()* that spawns two processes: a producer process and a consumer process; *cooperate:go()* will be called from the shell to run the program
8. Create a function *producer(Pid, N)* that takes consumer's pid and an integer N as arguments. The function is to repeat N times. While the function is running, it generates random numbers and prints them along with its process id to the shell (as in *process <0.55.0> producing 5, process <0.55.0> producing 1*, …):, and then sends these numbers as messages to the consumer process using consumer's pid. (Note: the producer function is set up to send messages but NOT to receive). When you **spawn** this function in *go(),* decide on the value of N – you will want to pick some small N for testing purposes (e.g. 7)
9. Create a function *consumer()* that consumes the numbers sent by the producer via the receive statement, and prints them with along with its process id to the shell (as in *process <0.88.0> consuming 5, process <0.88.0> consuming 1*, …). (Note: the consumer function is set up to receive messages but NOT to send)
    - After verifying everything works, extend the *consumer()* function by putting the process to sleep for 10 units and only then beginning to receive**,** `timer:sleep(10)`**.**
    - Finally, end the consumer process when no new message arrives for about 40 units of time (hint: use **after** primitive to do so)

--- CONGRATS – YOU ARE DONE – UNTIL NEXT LAB THAT IS ☺ ---