CSC 231 Spring 2015 **Wake Forest University**
Lab 7: Advanced Erlang: Higher Order Functions, Type Definitions
Due Date: Thursday, March 5th at midnight (since we won't have lab next week)

Higher Order Functions

As discussed last week, in most languages the parameters passed to a function are data components – items such as integers, strings, or objects. In Erlang, however, it is also possible to pass functions as arguments into a function. The set of functions that take functions as their arguments are often called *higher-order functions.* In a sense, this allows handling functions in ways in which we would normally handle data. Often times, we are interested in applying a function over an entire list of data, and it is in this case that higher order functions are very helpful.

Last lab you saw the *map* function, which applied a function passed in as a parameter to *map* to each element of a list:

map(F, L): This function takes a function F and a list L of [data1,data2,data3,…,dataN] and returns the list [F(data1), F(data2), F(data3), …, F(dataN)]. The function is applied to each individual element of the list.

This *map* function can be defined in Erlang as follows:

```
map(F,[]) ->
      [];
map(F,[X|XS]) ->
      [F(X)|map(F,XS)].
```

An example application of map is as follows:

```
square(X) ->
      X*X;

squareList(L) ->
      map(fun square/1, L).

squareList([1,2,3]).
[1,4,9]
```

Here are two more useful higher-level functions:

reduce(F, L): This function takes an associative (binary operand) function F and a list L of [data1,data,data3,…,dataN] and returns the results of applying that function in the following manner: F(data1, F(data2, F(data3, … F(dataN)))). As an example, assume that the function F was the plus operation. Then, *reduce* called with the plus operation (addition function) and an integer list as parameters would return the sum of the integers in the list.

The *reduce* function can be defined in Erlang as follows:

```
reduce(_, [A]) ->
                        A;
reduce(F, [A|AS]) ->
                        F(A, reduce(F,AS)).
```

Note that the *reduce* function is undefined for an empty list as there is no data to reduce over.

An example application of *reduce* is as follows:

```
add(X,Y) ->
      X + Y.
sumList(L) ->
      reduce(fun add/2, L).

sumList([1,2,3,4]).
      10
```

Note that the + operator wasn't used as the function parameter to *reduce*. The + operator, and other **infix** operators, need to be mapped to a true function.

Also take note of how *reduce* is defined. The ordering of addition that is done in the above example

```
sumList([1,2,3,4])
```

is

```
1 + (2 + (3 + (4)))
```

since *reduce* is defined as: `F(X, reduce(F,XS))`

filter(F, L): This function takes a predicate function F (which returns true/false) and a list L of [data1,data2,data3,…dataN] and returns the list M which contains all elements of L that satisfy the predicate F.

```
filter(_,[]) ->
      [];
filter(P,[X|XS]) ->
      case P(X) of
            true ->
                  [X|filter(P,XS)];
            _ ->
                  filter(P,XS)
      end.
```

Note that I introduced a different style of if/else testing that we haven't seen before. Remember that the general if/else structure in Erlang will not allow user defined functions to be part of the conditional to prevent side-effects from the conditional. The *case* statement in Erlang does not have these restrictions.

The case statement employs *pattern matching* instead of true Boolean conditional analysis in deciding which case to execute. The expression occurring after the word `case` is evaluated, and its result is then pattern matched against each of the values listed in the body of the case. When one matches, the resulting code is executed. In the code above, the predicate P() is applied to X. If its result (which is either `true` or `false`) pattern-matches the atom `true`, then the item is kept and pre-pended onto the results of filtering with the predicate against the back of the list. If the predicate returns anything else than true, then the item is dropped and the results of filtering are set to be the results of filtering just the tail of the list.

An example of the use of the *filter* function is:

```
greaterThan10(X) ->
      X > 10.

dropSmall(L) ->
      filter(fun greaterThan10/1, L).

dropSmall([1,2,12,10,18,8]).
[12,18]
```

Note how just the values in the list greater than 10 are in the result list returned from the function. As another example, a predicate function can be easily written to determine if an integer is odd. If that predicate is applied over a list with the *filter* function, the returned list will only contain the odd integers.

Defining New Types

Because tuples and lists are capable of holding arbitrary combinations of types internally, including other tuples and lists, it is possible to define higher-level data-types based off of tuples and lists. A problem with these approaches, however, is that we are constrained to interact with tuples using positional access and with lists using head/tail access. It would be useful to allow name-based access (like objects/classes).

To better support the definition of data types, Erlang supports the ability to define a *record*. A record is defined as a combination of other types of data, but with *named* components.

A record's structure is defined with the following syntax:

```
-record(recordName, {atoms, describing, data, fields,…})
```

For example, a record representing a person might be defined as:

```
-record(person, {name, age, phone})
```

One can provide default values for each field by using the = symbol in the record definition:

```
-record(person, {name, age=0, phone=””})
```

In this example, age would default to 0 and phone number would default to the empty string.

To construct an instance of a record, where one assigns actual values to each data component, the following constructor syntax is used:

```
#recordName{dataFieldName=data,dataField2Name=data2,…}
```

Using this to create a person named Joe, with age 21 and phone number 555-0011, we do:

```
#person{name=”Joe”,age=21,phone=”555-0011”}
```

We don't have to set all of the fields – it is OK to use the following if we just want to assign the name Fred to the person (this should be useful when combined with the default values option above).

```
#person{name=”Fred”}
```

To work with an instance of a record, the following syntax is commonly used – first assign the constructor return value to a variable:

```
VariableName = constructor_call
```

As an example,

```
ThePerson = #person{name=”Fred”}
```

Then, to access the fields, use this hash-and-dot based syntax:

```
VariableName#recordName.fieldName
```

For example, retrieving a person's age:

```
Age = ThePerson#person.age
```

To modify a field, use syntax like the constructor syntax (including having a new Variable on the LHS (left-hand-side) of the assignment statement):

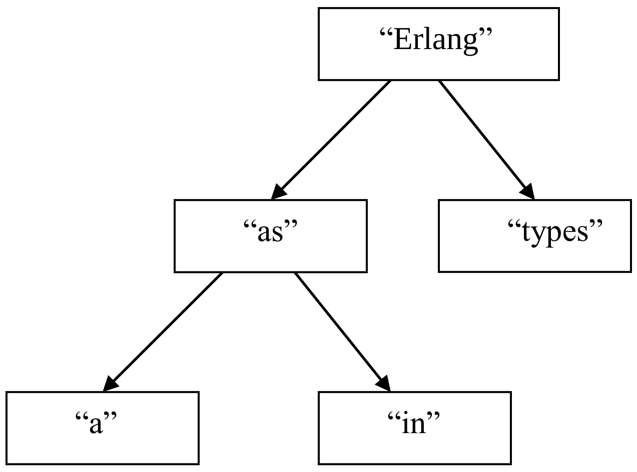
```
ThePersonUpdated = ThePerson#person{age=21}
```

It is possible to include records within records, such as having a *name* record, consisting of a first name and last name, which is part of a *person* record.

The final syntax note concerning records is that to make use of records we now need to both compile and read records from our Erlang modules:

```
c(lab7).          // compile
rr(lab7).         // read records
```

When we allow recursion in these datatype definitions, very powerful data structures can be defined. Take, for instance a binary tree. A binary tree in C++ holds a data value and pointers to the start of two other binary trees [if you haven't heard of binary trees before, think of a linked list node with zero, one, or two outgoing links to other nodes, each of which also has zero to two outgoing links to other nodes].   The following picture illustrates a binary tree with five nodes that holds strings:



A textual description of this tree is as follows: A root node with label "Erlang" exists with a left child node labeled "as" and a right child node labeled "types". The left child node labeled "as" has a left child node labeled "a" and a right child node labeled "in". The nodes labeled "a", "in", and "types" have no children.

Binary trees are defined recursively by the following rules:

Basis: An empty tree is a binary tree.
Inductive Step: Given T1 and T2 are binary trees, then T3 is a binary tree constructed by generating a new node N with data d, left subtree T1, and right subtree T2. The new node N is called the root of T3.

A record definition for a binary tree node and a binary tree in Erlang is as follows (note the default values):

```
-record(binaryTreeNode, {value,left=null,right=null}).
-record(binaryTree, {rootNode=null}).
```

Note what this definition is saying – a *binaryTree* is represented by a *rootNode* data component, initially set to the null atom, and then each *binaryTreeNode* consists of a *value* (no default provided) and a *left* and *right* child, each initially set to the null atom.

Given this definition, different parts of the tree drawn above can be represented as follows:

The tree starting at the node "types":

```
      #binaryTreeNode{value="types"}.
```

The tree starting at the node "as":

```
      #binaryTreeNode{value="as",left=#binaryTreeNode{value="a"},
      right=#binaryTreeNode{value="in"}}.
```

The entire tree:

```
      #binaryTreeNode{value="Erlang",left=#binaryTreeNode{value="as",left=#binaryT
      reeNode{value="a"},right=#binaryTreeNode{value="in"}},right=#binaryTreeNode{
      value="types"}}.
```

To build a variable holding this tree, the following statement could be used:

```
N=#binaryTreeNode{value="Erlang",left=#binaryTreeNode{value="as",left=#binaryTree
Node{value="a"},right=#binaryTreeNode{value="in"}},right=#binaryTreeNode{value="t
ypes"}}.
T=#binaryTree(rootNode=N).
```

Given such a tree structure, it is possible to define functions over the tree recursively (the tree itself is a recursive data-structure, so recursive functions are a nice fit). The base case should be handling the empty tree, while the recursive case should be handling a non-empty tree (which is a node that has data and left and right pointers).

Here's a pair of functions that sum over a binary tree, assuming the tree of interest contains integer data items instead of the strings shown in the previous examples.

One function, *sumTree*, takes a tree as an input and instantly passes off the work to *sumNode* by sending the *rootNode* of the tree to *sumNode*.

sumNode returns 0 if it is passed a null node. Otherwise, it returns the value held in the node it was passed as an argument plus the result of summing the values in the subtree that is its left child plus the result of summing the subtree that is its right child.

```
sumNode(null) -> 0;
sumNode(N) ->
N#binaryTreeNode.value + sumNode(N#binaryTreeNode.left) +
sumNode(N#binaryTreeNode.right).

sumTree(T) -> sumNode(T#binaryTree.rootNode).
```

Takehome Problems:

Test input and output files can be found on the class website as *lab7Tester.erl* and *lab7TestOutput.txt*. **These tests are not guaranteed to be exhaustive.** Build your code starting from the skeleton provided in *lab7.erl*. This contains the reduce and filter functions, the binary tree data type, and several functions that you can use to access pre-built binary trees.

Please see the documentation requirements on the last page of the lab.

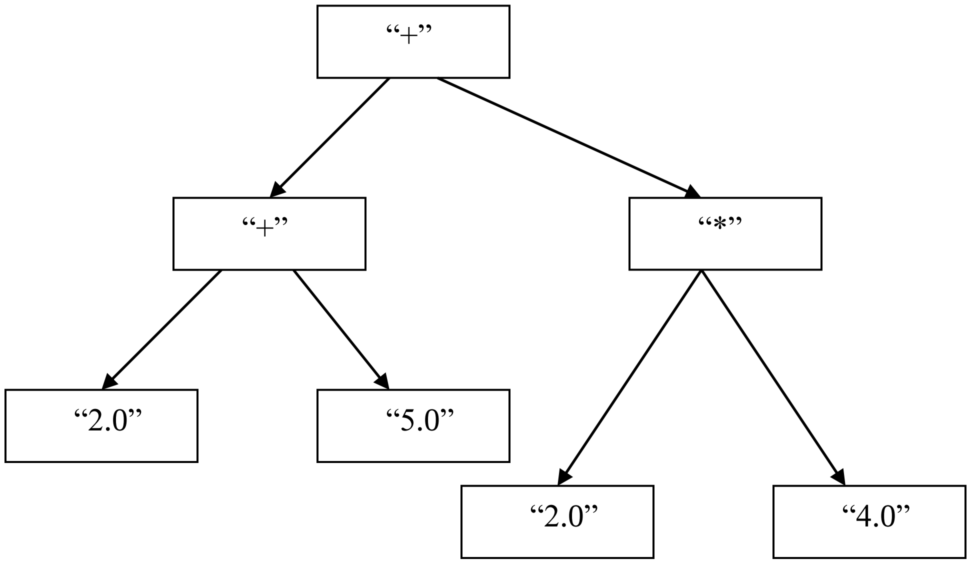
1. Define functions that perform the following operations. Your answers should consist of functions applying a "higher level function" (either *reduce* or *filter*) to an auxiliary function you build yourself which is applied over the elements of the list. So, in these solutions, you will need to build *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 reduce/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 lab to review some of the math operators and how to interact with characters ($charName) and strings (view stringas lists of ASCII numbers).
   a. Return a list containing the elements of a list of integers that are even. Name this function *extractEvens*.
   b. Find the elements of a list of strings that begin with the character *a or A*. Name this function *extractAWords*.
   c. Find the logical OR of a list of Boolean values. 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. Name this function *logicalOR*.
   d. Find the elements of a list of strings that are greater than 3 characters long. Name this function *longStrings*.
   e. Concatenate a list of strings into a single string. Name this function *strcat*.
   f. Find the maximum of a list of numerical values. Name this function *max*.

2. In *lab7Tester.erl,* you will find a binary tree datatype defined as follows:

   ```
   -record(binaryTreeNode, {value,left=null,right=null}).
   -record(binaryTree, {rootNode=null}).
   ```

   Write a function called *findInTree* which takes as its first argument a string label to search for and its second argument the tree to search within. Your function should be able to determine if a particular label is in the tree in time bounded by *n* steps total, where *n* is the number of nodes in the tree (so worst case, it traverses the whole tree, but on average, it does less than that). This is ***not*** a **binary search tree**, so you can't make any assumptions about what's on the left hand side or right hand side of the tree. Your approach should perform *findInTree* correctly while traversing the tree - the "bounded by n steps" requirement says you must compute the answer during traversal, not in a post-processing step.

3. In *lab7Tester.erl*, you will find functions whose names begin with *getExpression* which return binary trees containing operators and numerical values (with both operators and values encoded as strings). The trees have the following general shape:



The result of evaluating this example tree is 15.0 [(2.0+5.0) + (2.0*4.0)].Write a function *evaluate* that takes a binary tree that represents mathematical expressions of this sort and returns the value of the expression. You should implement the following operators (+,-,*,/ [addition, subtraction, multiplication,division]) assuming real numbers. Assume you will only get valid trees as input (where every operator node has two sub-trees which evaluate to a numerical value and number-holding nodes have no children). When you have an operand node, where you want to extract out the numerical value from the string stored in the node, use the function call:

stringToFloat(VariableHoldingString)

Documentation requirements:

- In a separate text/Word/PDF file from your answers, include, for each function you wrote (even if it is a supporting function for solving a bigger problem), a brief discussion of your implementation and a justification of your approach's correctness.
- You do not need to discuss additional test cases (though you might want to come up with them to ensure your code is correct).

Due Date: Thursday, 3/5/2015, 11:59pm (midnight)

To submit via Sakai:

- One file called *lab7.erl* which contains the functions requested above.
- One file, called *lab7Documentation.txt*, which contains the discussion/justification of your choices for implementing the functions

CSC 231 Lab 7 Gradesheet

Name: ________________________________

Task:	Available Points:	Earned Points:
Programming:		
Problem 1a – extractEvens	9 points	
Problem 1b – extractAWords	9 points	
Problem 1c – logicalOR	9 points	
Problem 1d – longStrings	9 points	
Problem 1e – strcat	9 points	
Problem 1f – max	9 points	
Problem 2 – findInTree	11 points	
Problem 3 – evaluate	11 points	
Documentation (solutions):	3 points each task (8 tasks above ➔ 24 points)	

Total: