CSC 231 Spring 2015 **Wake Forest University**
Lab 5: Introduction to Erlang – The Basics
Due Date: Tuesday, 2/17/2015 – 5:00pm

Overview of Lab

In today's lab, the basics of the Erlang programming language will be introduced. Erlang falls into a class of programming languages called *functional programming languages*. The core idea behind functional language programming is the development of function definitions which can be evaluated as and within expressions. Higher level programs are built by combining functions according to standard mathematical principles.

There is much less emphasis on control structures like loops in functional languages – loops actually will be replaced by recursive functions. Many other standard language issues, such as memory management, are also all handled by the Erlang system itself. This approach provides for a relatively simple, yet very powerful, programming environment.

You should have an Erlang interpreter already installed on your computer (from lab 1).

Example Erlang Program

Let's examine the traditional "first program", Hello World, in Erlang. This program can be found on the class website as *helloworld.erl.*

```
% Hello World Program in Erlang
-module(helloworld).
-export([main/0]).
main() -> io:format("Hello World\n").
```

The first line of helloworld.erl is an Erlang comment. Comments are started with a *%* and run to the end of the line.

The second line of the program is a declaration of the name of the module (group of functions) being written. It must match the name of the file (minus the .erl) the functions are being saved in.

The third line indicates all of the functions that are to be made accessible from this file to other modules and to the Erlang interpreter. Within the square brackets, each function being exported should be listed, comma separated, as *functionName/arity*, where *arity* indicates the number of parameters to the given function. For this example, *main* takes no parameters so it has an *arity* of 0. Since there are no other functions in the file besides *main*, only *main* is listed as being exported.

Finally, the last line defines the main function. The arrow, ->, indicates that main should be implemented by the code following the arrow, which in this case is to use the *format* function from the *io* module and apply it to the string "Hello World\n". This is much like a `cout << "Hello World\n"` in C++.

All "complete" Erlang statements should end with a period to indicate a terminated statement (complete is a loose term, as you will notice below when function definitions are covered).

So far, this program doesn't look too much different from Hello World programs in other languages. To execute this program, one needs to instantiate the Erlang interpreter and feed it this input file as the program to run. This can be achieved by running the following command at your Unix prompt:

```
erl
```

Once the erl interpreter has started, you can compile (syntax check & load) a code file in the directory you are in by using the command

c(filename). // c indicating the compile function

The Erlang interpreter on the surface is a very simple, line by line interpreter (much like the BASIC interpreter). It is possible to type expressions to be evaluated directly into the interpreter (such as *4+2.*). For us, it will be easier to develop an entire program within an external file via gedit or another editing tool. We can then load that program using the syntax shown above.

After a successful compile, the program can be executed by using the following syntax:

helloworld:main().

This is calling the given function by using the module name (*helloworld*), the function name (*main*) and providing any necessary parameters (none in this case).

The output of the Hello World program is as follows:

```
Hello World
ok
```

The output above indicates that the program first successfully printed the string we requested to be printed with the io.format function and that the function returned without error.

To leave the Erlang interpreter *erl* type the following: q().

Types in Erlang
The built-in types in Erlang: integers, floats, atoms, Booleans, characters, and strings.

Integers are represented as they are in any other language (*one or more digits*). To designate a negative integer, a minus symbol (-) is placed in front of the numbers.

Floating point numbers also have the traditional representation – including an optional negative sign and the ability to use scientific notation. Here are some examples:

-123.0	(equal to -123.0)
1.23E2	(equal to 123.0)
3.0E-3	(equal to 0.003)
3.14E12	(equal to 3.14 x 10^12)

Atoms are constant literals that represent themselves. They are represented by words starting with lowercase letters, optionally surrounded by single quotes. They are ordered lexicographically (dictionary ordering).

The atoms `true` and `false` are used to comprise the Boolean values.

String values are quoted character strings, such as “foo” and “BAR”. These are actually converted into lists of integers (but we’ll come back to that later).

Characters map to the integers, and are represented as *$charOfInterest*.

Operators
Arithmetic operators for Erlang are defined below (from highest precedence to lowest):

Unary positive, negative:+,-
Binary Multiply/Divide:

- * (multiplication)
- / (division of reals)
- div (division of integers)
- rem (remainder for integers).

Binary Add/Subtract: +,-

Parentheses in expressions can override the normal order of operations.
Comparison operators are also available. These operators always return a Boolean value. They also have lower precedence than the arithmetic operators (so you can type 2 < 1 + 3 and it will be evaluated correctly [as *true,* since 2 is less than 4].

Operation	**Syntax**	
`Less than`	`<`	
`Less than/equal`	`=<`	`note the odd syntax here`
`Greater than`	`>`	
`Greater than/equal`	`>=`	
`Equal to`	`==`	
`Not equal to`	`/=`	

Logical operators are as follows: the keyword `not` for *negation*, the keyword `andalso` for *and*, and the keyword `orelse` for *or*. (Technically, you can just use `and` and `or`, but those do not employ shortcut evaluation and thus are inefficient). `xor` returns true if one of its arguments is true and the other is false. Logical operators are lower precedence than comparison or arithmetic.

Shortcut (also called) lazy evaluation exploits only evaluating the smallest amount of the expression possible to determine the value of the expression being computed. If the first condition of the `andalso` statement fails, none of the other conditions are tested. If the first condition of the `orelse` statement holds true, none of the other conditions are tested.

Tuples and Lists

A *tuple* is an arbitrarily sized grouping of data of arbitrary types (think of a C struct or a simple Java object without methods). The syntax for a tuple is as follows:

{data, data, data} // they can be larger or smaller than size 3, as well as empty

Here's an example: `{4, 5.0, true}`

Note that each type being held in this example grouping is different (integer, float, string).

Tuples can also contain sub-tuples. Here are some examples:

{1,{2, 3.0}}	which is of type int * (int * real).
{{1,2}, 3.0)	which is of type (int * int) * real

Three important tuple functions to know are:

Getting the size of the tuple: `tuple_size(TupleGoesHere).`
Obtaining the element at index i (1 <= i <= tupleSize): `element(i, TupleGoesHere).`
Setting the element at index i (1 <= i <= tupleSize): `setelement(i, TupleGoesHere, newValue).`
The `setelement` function returns a new tuple with the i^th entry replaced with newValue.

A separate data-structure that is called a *list* contains multiple items, again of potentially varying type and containing list substructures if desired. The syntax is as follows: [data, data, data, …].

A list of the integers seven through ten would be defined as: `[7,8,9,10].`

Lists are fundamentally different from tuples in how they can be manipulated. While tuples can be thought of much like arrays with indexing, lists should be thought of as recursive data-structures. In this context, a non-empty list should be thought of as a head element, followed by a smaller tail list (which could possibly be empty).

The empty list is designated as a pair of empty brackets `[]`.

The length of a list can be returned using the function `length([list goes here])`.

While there is a indexing function for lists (`lists:nth(index,[list goes here])`), *you will be asked to ignore it for this class*. Instead, remember that the head of a list is defined as the first element, while the tail is defined as the list of all elements in their original order except the first element. (So in reality, lists are kind of like linked lists!). Given this idea, the head element and tail sublist of a list can be retrieved using the functions `hd` and `tl`. Here are some examples (italics are used to separate the Erlang response from the input line).

```
hd([2,3,4]).
2

tl([2,3,4]).
[3,4]

hd([5]).
5

tl([5]).
[]
```

In addition to using `hd` and `tl` to separate a list into components, one can also combine data together into a single list. The cons operator `|` (the bar symbol), when used within square brackets, takes a single element and a list and prepends the element onto the front of the list.

```
[2|[3,4]].
[2,3,4]

[1|[2|[3,4]]].
[1,2,3,4]
```

Note that cons only supports pre-pending – adding a single element just to the front of the list.

The list concatenation operator `++` takes two whole lists and combines them into a new list:

```
[2,3]++[4,5].
[2,3,4,5]
```

Variables

Valid identifiers (variable names) in Erlang are a subset of the character strings, much like most other languages. Identifiers have the following syntax:

1. Start with an uppercase letter (to differentiate from *atoms*)
2. Followed by uppercase or lowercase letters, as well as digits and underscores (_)

Some examples of alphanumeric identifiers are: `Degree`, `Count1`, `NumberOfHamburgers`, `First_name`.

A variable can be assigned a value, but only once. The assignment operator is = as in other languages.

```
<identifier> = <expression>
```

By only being able to be assigned a value once, variables act more like aliases than they do state information.

An example where the radius of a circle, the value PI, and the area of a circle are being defined and set is as follows:

```
Radius = 4.0.
PI = 3.14159.
Area = PI * Radius * Radius.
Area. // prints 3.14159 * 4 * 4 ➔ 50.26544
```

It is possible to forget the values bound to all variables using the function *f()* and to forget the value bound to a given variable with the function *f(variableName)*. These two functions are only available from the Erlang shell and cannot be used within a program.

All variables used within a function are considered to have function-level scope, there are no global variables, and all parameters are passed by value (any expressions being sent to a function as parameters are completely evaluated first).

Variables cannot be declared with a type – any variable can take on any value. Since this is true, compilation cannot find all type errors. Accordingly, many type errors appear at runtime, causing the problematic function to halt. Here is an example of a run-time detected type error in line 26 (trying to multiply an integer by an atom), followed by two uses of multiplication that work correctly:

```
26> 2*true.
** exception error: bad argument in an arithmetic expression
    in operator  */2
       called as 2 * true
27> 2*2.
4
28> 2*2.0.
4.0
```

Functions

Functions make up the core of Erlang programming, so learning how to define functions is the key to learning the Erlang language. Whenever you feel like you should write an iterative (looping) block of code to solve a problem, Erlang will say to write a function (preferably one that is recursive).

The basic way to declare a function has the following syntax:

```
<identifier>(<parameter list>) ->
      <expression>.
```

The identifier representing the function name has to be an atom (start with a lowercase letter). There can be zero or more parameters, separated by commas. Finally, note that in all of today's code a period is only used at the very end of the function definition, even if there is a fairly complicated set of statements that make up the <expression> part of the implementation.

Here's an example of a simple function definition to return the square of a number:

```
square(N) ->
        N * N.
square(3.0).
9.0
```

Any number of parameters to a function are allowed. Here's a function which takes three parameters and returns the maximum value: (it also employs the if-then-else syntax, to be explained later).

```
max3(A,B,C) ->
      if
           A > B -> if
                       A > C -> A;
                       true -> C
                     end;
           true -> if
                       B > C -> B;
                       true -> C
                   end
      end.
```

The if-then-else syntax has the following format:

```
if
      conditional1 -> expression1,expression2,…,expressionN;
      conditional2 -> expression1,expression2,…,expressionN;
      …
      true -> expression1, expression2,…,expressionN
end
```

For today's lab, we will limit conditionals to being Boolean expressions. It is possible to call functions as part of the conditional (length(List1) > length(List2)), but you can NOT call user-defined (your own) functions (the functions in an if-statement test have to be side-effect free, and that is not guaranteed for your functions – it is for built in functions though!).

Each conditional is then associated with a series of one or more comma-separated expressions. The last expression associated with each conditional provides the return value for the case specified by that conditional. A final `true` conditional is required to support catching all cases not explicitly covered by earlier conditionals. Each line except for the `true` conditional requires a semicolon at the end (and the `true` conditional line cannot have one). Finally, the keyword `end` is required to mark the end of the if-statement. In this lab, we will not need to use more than one expression – so you should not make use of the comma separated series of expressions.

Note that by being forced to have a catch-all `true` conditional, we are defining a return value for all possible inputs. One should think of *if statements* not as state changing (adding special code to handle a particular state) but as selection of one out of many values to return (notice again the focus on functions, with every function required to have a return value).

Most of the real work done in Erlang programming is through recursive functions. As with all languages that support recursion, it is necessary to remember the two key ideas behind recursive functions:

1. The solution of large problems is found by combining solutions for smaller problems,
2. One requires the presence of a base case which can't be decomposed further and which has an immediate (non-recursive) solution.

In Erlang, we will use recursion as a replacement for iteration (our usual while and for loops). For a beginning example, let's look at a function which can reverse a list:

Base Case: The list is empty – the reverse of an empty list is the empty list
Recursive Case: The list is non empty – the reverse of a non-empty list is the first element of the list attached to the end of the reverse of the rest of the list.

If our list was [1,2,3], the reversed list is [3,2,1]. This can be achieved through the recursive case above by nothing that [3,2,1] is the concatenation of (the reverse of [2,3]) and (the single element list [1]).

What types of functions do we know of that could be useful in defining this function? The `hd` function will remove the 1 from the list as a single element. The `tl` function will return the [2,3] part of the list as a list. To concatenate two lists, one can use the `++` operator. Right now though, we don't have two lists, we have a list [2,3] and a single element 1. To make 1 into a list, you could append it to the empty list, using the | (cons) operator.

Given these operators, here's a valid reverse definition:

```
reverse(L) ->
      if
            L == [] -> [];
            true -> reverse(tl(L)) ++ [hd(L) | []]
      end.
```

and a sample of execution:

```
reverse([1,2,3]).
[3,2,1]
```

Recursion execution in Erlang is very similar to that of C and Java. Whenever a new recursive call is made, a new set of variables are added to a stack of local variables. When the recursive call returns, these new variables go away and just the returned value from the called function is accessible.

Takehome Problems:

Solution constraints:

- *For all functions, assume that only valid inputs are given (you don't need to do error checking).*
- *Ignore any pang of concern that you have about inefficiency and just learn the basics of the language.*
- *You should not make use of any functions not already presented in this lab.*
- *You should not make use of any comma separated series of expressions - all of the functions that will need to be written today are essentially one expression functions. That is, all the work can be encapsulated in a single statement – either directly in one statement or as components of one or more nested `if` statements (for recursion).*
- *Name your source code file: lab5.erl   Include all functions in this one source code file.*
- *Please see the note on documentation to include on the next page.*

1. Write and test **non-recursive** function definitions in Erlang that perform the following actions:
   a. Return x^4, for any number x, using just multiplication. [Name this function *fourthPower*]
   b. Implement the square function shown in the lab (on page 5), then use this to write another version of the fourthPower function using the square function. [Name this function *fourthPower2*].
   c. Return the median of three components of a tuple of numbers [Name this function *median3*; note: you have 1 input parameter – a tuple of size 3, not 3 separate numerical input parameters]. The max function could be useful for inspiration on how if-else statements are structured.

2. Write and test **recursive** functions that perform the following actions:
   a. Write the factorial function where n is passed to the function as a numerical parameter. You can assume this will only be tested with integers >= 0. Define 0! to be 1 and 1! to be 1. [Name this function *factorial*]
   b. Take the cycleOnce function below and copy it into the file you are building of functions. Write a new function that returns the nth item out of a list and that makes use of cycleOnce. Your function will have two arguments in this order – the input list and the integer index n. Assume the indices start at 1. [Name this function *getNth*]
```
cycleOnce(L) ->
    tl(L) ++ [hd(L) | []].
```
   c. Given two (assumed integer) variables, X and Y, in that order, as parameters, write a function that returns the power of X that equals Y. Assume only actual powers of X >= X^1 will be passed in as the Y parameter (as an example: if 3 was X, valid inputs for Y that you need to handle are 3,9,27,81,…) [Name this function *logarithm*]
   d. Compute the largest element of a list of numbers with one or more elements by using the following algorithm: [Name this function *largest*]
      i. If the list has an empty tail, then the head is the largest element.
      ii. Otherwise, compare the head against the head of the tail (effectively the 1st element vs. the 2nd element). Depending on which is bigger, call the largest function recursively with either the head merged onto the tail of the tail, or call largest on just the tail. For example:
      largest(3,2,5,1) := compare 3 to 2, 3 is larger, call largest on (3,5,1)
      largest(3,4,5,1) := compare 3 to 4, 4 is larger, call largest on (4,5,1)
   e. Write a recursive function that takes two arguments: a data item passed in as the first argument, and a list passed in as the 2nd argument. Return true if the data item is contained in the list, false otherwise. [Name this function *contains*].

Files to help test what you have written can be found on the class website as *tester.erl* and *lab5TestOutput.txt*. They are not guaranteed to be exhaustive in testing, but they should help you find errors!!! To use these, after you build each function above, compile your lab 5 code, then compile the tester code and run the appropriate test. For example, once you have problem 1a completed (a fourthPower function in your lab5.erl file), do the following inside of the Erlang interpreter:

```
c(lab5).            // compile your lab5 code
c(tester).          // compile the tester code
tester:test1a().    // use the tester script to test your solution to problem 1a
// check against the expected output in lab5TestOutput.txt
```

Documentation requirements: In a separate text/Word/PDF file from your functions, include, for each function, a brief (one to two sentences) discussion of why you implemented each function as you did. For example, for the logarithm problem (Recursion, problem (d)), justify that your implemented approach is an appropriate implementation for computing the desired functionality (the logarithm of a number given a particular base) correctly.

Due Date: Tuesday, 2/17/2015, 5pm **To submit via Sakai:**

- **One file called *lab5.erl* which contains the function definitions requested above.**
- **A second file called *lab5Documentation(.txt/.doc/.pdf)* that contains your explanation and justification of your implementations.**