CSC 231 Spring 2015 **Wake Forest University**
Lab 6: Intermediate Erlang, Part 1
Due Date: Tuesday, 2/24/2015 @ 5pm

Overview of Lab

This lab continues the study of the Erlang programming language. It covers more advanced function definition techniques (patterns and local variables) and introduces higher order functions.

Pattern Matching in Erlang Functions

Here's the function for performing list reverse introduced last week:

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

and a sample of execution:

```
lab5:reverse([1,2,3]);
[3,2,1]
```

Erlang provides an alternative formulation for defining certain types of *if-else* driven functions which tends to be more intuitive than this default formulation. This alternative makes use of *patterns* in the function argument list.

A pattern in Erlang is an expression over a set of variables. This pattern is placed in the argument list when a function is defined. When an actual argument matches the pattern at runtime, the variables listed inside the pattern are assigned the actual parameter values. These variables can then be used directly within the function.

As an example, one commonly used pattern is the expression [X|XS]. This pattern indicates an element X being attached to a list XS. Accordingly this pattern will match any list with one or more elements (anything but the empty list). As an example, the list [1] matches this pattern as [1| [ ]], and the list [1,2,3] matches as [1 | [2,3]]. A programmer can use more than two variables in a pattern.

To define a function using patterns, a function definition is attached to each possible set of arguments for that function. Semicolons are used to separate the multiple patterns allowed for the function and to delineate the different possible code-paths through the function to be executed. The last clause ends with a period. The last clause is often designed to use the pattern Other, representing a variable which will match any input, and is implemented to handle a default case. The general syntax for pattern based function definitions is as follows:

```
<identifier>(<first pattern>) →
        <first expression>;
<identifier>(<second pattern>) →
        <second expression>;
<identifier>(<third pattern>) →<third expression>;
…
<identifier>(<last pattern>) = <last expression>.
```

The identifier (AKA function name) has to be the same on every line, but everything else can change (different patterns, different function code).

With this idea, the list reverse function defined earlier (on page 1) can now be rewritten using patterns:
```
reverse([]) ->
      [];
reverse([X|XS]) ->
      reverse(XS) ++ [X].
```

Many people would argue this easier to read and more intuitive compared to our original implementation:
```
reverse(L) ->
if
      L == [] ->
                  [];
      true ->
                  reverse(tl(L)) ++ [hd(L)]
end.
```

Note that the original if/else, as well as variable assignment (hd(L) to X, tl(L) to XS), are performed implicitly with this pattern approach.

Erlang will parse through a list of patterns for a function, looking for a match and **only executing the first pattern match that is found**. Erlang will report an error at runtime (*no function clause matching…*) if an in-exhaustive set of patterns is used and an actual argument sent to a function is unable to be matched.

Here's another example of pattern matching - a merge function which merges two integer lists in sorted order, given the lists themselves are already in sorted order (a merge like you would use in mergesort):
```
merge(List1,[]) ->
      List1;
merge([],List2) ->
      List2;
merge([X|XS],[Y|YS]) ->
      if (X < Y) ->
            [X | merge(XS,[Y|YS])];
      true ->
            [Y | merge([X|XS],YS)]
      end.
```

The base cases for this function are for handling a merge of a list with data with an empty list. The recursive case determines which head entry is smaller. That smaller entry is attached to the front of the list generated by merging the rest of the list with the small element with the complete other list.

A list of the most commonly used patterns that can be used as function arguments are:
Constants: [ ] (for the empty list), 0 (or any raw numerical value)
List expressions using cons: [X|XS], [X | [Y|YS]]
Tuples as head element: [{A,B} | YS].

The underscore symbol can be used as a wildcard or "don't care" pattern – it matches anything, but since it is not a variable, the value of what was matched against the wildcard cannot be accessed. Here is an example of a list *contains* function that employs pattern matching and the wildcard symbol.
```
contains(_,[]) -> %always false if list empty, regardless of query item
      false;
contains(X,[X|_]) -> %true if query item matches head, regardless of rest of list
      true;
contains(X,[_|YS]) -> %can assume didn't match head, test against rest of list
      contains(X,YS).
```

If you see a *Warning* that a variable is being unused, it is likely that Erlang is suggesting that it could be changed to a wildcard variable.

Not all possible expressions are legal however – these will result in *illegal pattern* errors.

++ operator (list concatenation):

```
llength([]) ->
        0;
llength(XS++[_]) ->     % this is illegal
        1+length(XS).
```

Arithmetic operators:

```
square(0) ->
        0;
square(X+1) ->          % this is illegal
        square(X) + 2*x + 1.
```

Explicit Local Variables Inside of Functions:

The conceptual parts of programming are often easier when one allows the definition of local variables within functions. In particular, local variables are important in improving the readability (and thus write-ability) of software. Local variables may also increase program performance as operations that must be repeated multiple times (when local variables aren't allowed) to get one piece of data can be collapsed into one call.

Local variables can be used inside of a function definition by using the assignment (=) operation. Variables can only be assigned a value once within a function call (but can be re-assigned across calls). To make effective use of variable assignments, we also need to take advantage of the ability to have more than one expression in a function.

A function can be defined as executing more than one expression by separating each expression by a comma. The value of the last expression that is executed is the value returned from the function.

An example of variable assignment and comma separated expressions is shown below in a function for finding x^{20}:

```
power20(X) ->
      Two = X*X, Four = Two*Two, Four*Four*Four*Four*Four.
```

Here's a useful example for variables which performs splitting of a list into two lists – one list contains the elements in the odd positions of the original list, the other contains the elements in the even positions. The returned value is a 2-element tuple, the first element being one of the lists (the odd-based list), the second being the other list (even-based).

```
split([]) ->     %an empty list when split gives 2 empty lists
      {[],[]};
split([X]) ->  %a 1-element list gives the 1-element list, and an empty
      {[X],[]};
split([X|[Y|YS]]) -> %handle multi-element by breaking odd/even elements
      {M,N} = split(YS), {[X|M],[Y|N]}.
```

We have effectively just introduced local variables and blocks of code, so those should facilitate your programming of functions.

A Complex Example:
Given some of the functions already introduced in this lab, we can write a MergeSort sorting function (one of the faster sorts) in Erlang by combining our previously described split and merge functions:

```
% taken from earlier in the lab
split([]) ->
      {[],[]};
split([X]) ->
      {[X],[]};
split([X|[Y|YS]]) ->
      {M,N} = split(YS), {[X|M],[Y|N]}.

% taken from earlier in the lab
merge(List1,[]) ->
      List1;
merge([],List2) ->
      List2;
merge([X|XS],[Y|YS]) ->
      if (X < Y) ->
            [X | merge(XS,[Y|YS])];
      true ->
            [Y | merge([X|XS],YS)]
      end.

% new code
mergeSort([]) ->
      [];
mergeSort([X]) ->
      [X];
mergeSort(Y) ->
      {M,N} = split(Y), A = mergeSort(M), B = mergeSort(N), merge(A,B).
```

Higher Order Functions
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 and to return functions as return values. In this lab, we will focus on the set of functions that take functions as their arguments, which are often called *higher-order functions.* Using functions in this way is one of the most powerful aspects of using functional languages.

Why bring up functions as arguments? Consider that we might be interested in applying a function over an entire list of data. One can imagine (given last week's lab) doing such by using a recursion algorithm to emulate a for-loop process. Higher order functions are very helpful here in abstracting out this process. Consider the *map* function defined below:

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 F is applied to each individual element of the list.

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

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

To pass a function as a parameter, we have to reference it in the following way:

fun moduleName:functionName/arity *(moduleName isn't required if using within the module, but is if using from within the erl interpreter shell)*

Note that *moduleName:functionName/arity* uniquely refers to a function.

An example application of *map* is as follows. First, assume we have written a *square* function in the same module as where we wrote the *map* function (a *lab7Examples* module for example):

Definition:
```
    square(X)  ->
         X * X.
```

Use of map and square together:
```
    lab6Examples:map(fun lab6Examples:square/1,[1,2,3]).
    [1,4,9]
```

We can go one step further and define a *squareList* function using *map*, assuming *map* and *square* are defined as they are above:

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

Use:
```
    lab6Examples:squareList([1,2,3]).
    [1,4,9]
```

<u>Takehome Problems:</u>
A basic set of tests and expected outputs for your functions are available in Sakai. ***The list is not guaranteed to be exhaustive (it doesn't test all possible input cases).***

1. Write a **recursive** function *using patterns and without using local variables* called *productOfPairs* that takes an input list of numbers and outputs a list which contains the product of each two ordered entries in the input list. For example, *productOfPairs*([1,2,3,4]) should return [2,12] (which comes from 1*2,3*4). If there are an odd number of elements, leave the last element in its current place. Make sure you have a pattern which fits all types of lists.

2. Assume we want to represent polynomials with a list representation. Each entry in the list represents the coefficient for a corresponding power of x, with the first entry representing the constant (x^0) and successive entries representing the higher degrees. As an example, x+3 would be represented using [3.0,1.0] and x^3+4x-5 would be represented as [-5.0,4.0,0.0,1.0].

   You can assume that when you receive a polynomial in this encoding, all zero coefficients between x^0 and highest non-zero coefficient degree (x^j) are explicitly represented. All zero coefficients for higher powers over the last non-zero coefficient (x^j) are not represented but can be assumed to be 0. The list of coefficients will always be in "reverse" order of our usual idea of polynomials (i.e. x + 3 is represented as [3.0, 1.0]). ***This is really just saying that you can assume polynomial encodings will be in the same format as those in the examples.***

   Write three functions for polynomials, *using patterns* where appropriate in place of *hd* and *tl* calls and *without using local variables*:
   A **recursive** function to add two polynomials, called *polyadd*, which takes two numerical lists and returns a single numerical list which is the polynomial obtained by adding corresponding coefficient entries in the two input polynomials.

A **recursive** function to subtract two polynomials, called *polydiff*, which takes two numerical lists and returns a single numerical list which is the polynomial computed by subtracting the appropriate coefficient entries in the second list from the corresponding entries in the first list.

A **recursive** function, called *polyeval*, to evaluate the polynomial for a given input value. The first argument to polyeval should be the input polynomial, the second should be a number for which evaluation is requested. Assume an empty input polynomial evaluates to 0 for any input value x. [*Helpful hint:* Horner's Rule for Polynomials]

3. Write three functions, *using patterns as appropriate and local variables as appropriate*, which allow for a basic implementation of sets using lists. The set definition says that items in the set may appear in any order in the underlying list, but can only appear once. Use patterns for the following function definitions – usually you should have a pattern for empty sets (empty lists) and one for non-empty sets.

    member(X,S) returns true if element X is in the set S.
    delete(X,S) removes x from the set S if X is present in the set.
    insert(X,S) adds x to the set S if X is not already present in the set. This should check the underlying list to verify that there isn't already an occurrence of X in the set before adding.

    Recursion may (should?) be useful for some of the set functions above.

4. Here's a solution to last week's *largest* function, using the requested technique:

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

    If there is only one item in the list, this function just returns that one item. Otherwise, it compares the 1st item to the 2nd item in the list. The *largest* function is then called again with the 2nd item dropped out of the list or the front item dropped out of the list, depending on which of the 1st and 2nd was smaller (the smaller is dropped). This technique is quite inefficient, as it calls the same *hd* and *tl* function multiple times. Rewrite this method using *local variables (not patterns)* to reduce the number of times that *hd* and *tl* are called. You can assume that the **initial** call to largest has at least one element in the list.

5. Implement ***descending*** selection sort for integer lists in Erlang, using *patterns* and/or *local variables* as much as possible. The basic selection sort algorithm for sorting a list of numbers from highest to lowest is to first find the largest number and store it at the front of the list. Then repeatedly find the largest number from the leftovers list and put this in the next position. My implementation required three functions: *largest* (written previously) to find the appropriate element to put at the front of the list, *remove* (very much like the set delete operation) to remove an element from the list so the rest of the list can be worked on recursively, and *selectionSort*, the selection sort function which ties everything together and puts the items in the right order in the final list. Assume your input list could be empty and that you may have repeated values in your list (which wasn't true for the *set* functions). Make sure your actual selection sort function is called *selectionSort*.

6. Copy the *map* function as written in the lab and then use it to solve the following problems. For both of these problems, you will need to define a worker function (the function that is mapped across the list) with a name of your choosing as well as a function named as specified in the problem which makes the actual *map* call with the worker function and a list as parameters.

a. Flip the sign of every element in a list of reals – that is, turn all negative numbers into their corresponding positive number, and vice-versa. Name your function *flipSigns* and have it take as a parameter a list of numbers. An appropriate worker function might be named *flipSign* (singular) and have an implementation that flips the sign of any single number.
b. Truncate each string in a list of strings so it is no more than four characters long. That is, delete the fifth and subsequent characters while leaving shorter strings alone. Name this function *truncateWords.* [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].

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 [such as *flipSign* in the *flipSigns* problem]), a brief discussion of your implementation and a justification of correctness.
- In another text/Word/PDF file, for each function include any additional test cases and expected outputs (beyond the basic ones provided) you developed to test your functions with. If you don't add test cases to a function, state that you didn't and why you decided not to.

Due Date: Tuesday, 2/24/2015 @ 5pm

To submit via Sakai:

- **One file called *lab6.erl* which contains the function definitions requested above.**
- **A second file called *lab6Documentation(.txt/.doc/.pdf)* that contains your explanation and justification of your implementations.**
- **A third file called *lab6Testing(.txt/.doc/.pdf)* that contains any additional test cases you developed and your justification for adding/not adding test cases.**