Programming languages: Erlang introduction - problem set

Introduction

These problems allow one to gain experience in writing functions in Erlang. Functions, particularly *recursive* functions, constitute the core of developing programs in Erlang.

Reminder: Compiling and running Erlang programs

Starting the interpreter

Instantiate the Erlang interpreter, by running the following command at your Unix prompt:

```
erl
```

Compiling code

Within the interpreter, compile a code file in the directory you are in by using the command:

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

Executing a function

After a successful compile, a function can be executed by using the following syntax within the interpreter:

```
moduleName:functionName(inputs, to, function).
```

This is calling the given function by using the module name and the function name and providing any necessary input parameters.

Quitting the interpreter

To leave the Erlang system, type the following within the interpreter.

```
q().    (include the period!)
```

Solution constraints for this exercise

- 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.
- 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).

Getting started

Use the command below to download a file named *erl1.erl* that we will use to hold all of our function definitions.

```
wget http://www.cs.wfu.edu/~turketwh/231/erl1/erl1.erl
```

Open that file for editing using an editor of your choice. *sublime2* and *gedit* are two graphical editors that are installed while *nano* and *vi* are two text-based editors that are installed.

Problems to gain experience with writing Erlang functions

Problems to go over as a class

1. Write and test a *non-recursive* function definition, named *fourthPower* that returns x^4, for any number x, using just multiplication in the definition.

```
erl1:fourthPower(2).
16
erl1:fourthPower(3).
81
```

2. Write a *recursive* function, named *factorial*, where n is passed to the function as a numerical parameter and the value for $n!$ is returned. Remember that $n!$, for $n \geq 2$ is the product of all values between n and 1, and 0! and 1! are both 1. You can assume this will only be tested with integers >= 0.

```
erl1:factorial(0).
1
erl1:factorial(1).
1
erl1:factorial(3).
6
```

3. Write a new *recursive* function, named *getNth*, that returns the n^{th} item out of a list, making use of the *cycleOnce* function provided to you in extracting the n^{th} item. Your function should have two arguments in this order – the integer index n and the input list and . Assume the indices of the list start at 1.

```
erl1:getNth(1,[2,4,10]).
2
erl1:getNth(2,[2,4,10]).
4
erl1:getNth(3,[2,4,10]).
10
```

Problems to work on alone on in small groups

1. Use the *square* function provided to you as the basis for implementing another *non-recursive* function, named *fourthPower2*, to compute n^4.

```
erl1:fourthPower2(2).
16
erl1:fourthPower2(3).
81
```

2. Given two (assumed integer) variables, X and Y, in that order, as parameters, write a *recursive* function, named *logarithm*, that returns the power of X that equals Y. Assume only actual powers of $X \geq X^1$ 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,...).

```
erl1:logarithm(2,32).
5
erl1:logarithm(3,9).
2
erl1:logarithm(4,4).
1
```

3. Write a *recursive* function, named *largest*, that compute the largest element of a list of numbers with one or more elements by using the following algorithm:

- If the list has an empty tail, then the head is the largest element.
- 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)
```

```
erl1:largest([2]).
2
erl1:largest([6,9,12,3]).
12
```

4. Write a *recursive* function, named *contains*, 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.

```
erl1:contains(2,[]).
false
erl1:contains(2,[3,2,5]).
true
erl1:contains(4,[3,2,5]).
false
```

5. Write a *recursive* function, named *count*, that takes two arguments: a data item passed in as the first argument, and a list passed in as the 2nd argument. Return how often the data item appears in the list.

```
erl1:count(2,[]).
0
erl1:count(5,[3,2,5,2,2]).
1
erl1:count(2,[3,2,5,2,2]).
3
erl1:count(8,[3,2,5,2,2]).
0
```