Programming languages: Erlang patterns - problem set

Introduction

These problems allow one to gain experience in making use of patterns, local variables, and multi-expression statements in Erlang. These approaches should signficantly simplify the process of writing Erlang functions.

Solution constraints for this exercise

- For all functions, assume that only valid inputs are given (you don't need to do error checking).
- You should not make use of any functions not already presented in this lab.

Getting started

Use the command below to download a file named *erl2.erl* that we will use to hold all of our function definitions.

```
wget http://www.cs.wfu.edu/~turketwh/231/erl2/erl2.erl
```

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

Problems to gain experience with Erlang

Problems to go over as a class

1. Rewrite the *count* function from last week, making use of patterns and without using local variables. It should be recursive and take 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.

```
erl2:count(2,[]).
0
erl2:count(5,[3,2,5,2,2]).
1
erl2:count(2,[3,2,5,2,2]).
3
erl2:count(8,[3,2,5,2,2]).
0
```

2. 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. 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.

```
erl2:productOfPairs([1,2,3,4]).
[2,12] % results from 1*2,3*4
erl2:productOfPairs([5,4,3]).
[20,3] % leave the last, non-paired, item in place
```

3. Here's a solution to last week's largest function, using the requested technique in that assignment.

```
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 1st 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.

```
erl2:largest([5.0]).
5.0
erl2:largest([4.0,5.0]).
5.0
erl2:largest([4.0,6.0,5.0]).
6.0
```

4. Write a function named *delete* using patterns as appropriate and local variables if needed, to support deletion of a value from a set. The set definition says that items in the set may appear in any order in the underlying list, but can only appear once. You should have at least a pattern for empty sets (empty lists) and one for non-empty sets. The function should take two inputs, the first being the item to delete, and the second being a list representing a set.

```
erl2:delete(2,[3,2,4]).
[3,4]
erl2:delete(3,[4,5,6]).
[4,5,6]
erl2:delete(8,[]).
[]
```

Problems to work on alone on in small groups

1. Write a function named *insert* using patterns as appropriate and local variables as appropriate, to support insertion of a value into a 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.

```
erl2:insert(3,[4,5,6]).
[4,5,6,3]  % or [3,4,5,6] <-- 3 just needs to end up in the list
erl2:insert(2,[3,2,4]).
[3,2,4]
erl2:insert(8,[]).
[8]
```

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 coefficients for higher powers over the last non-zero coefficient 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 – see the extra notes on this algorithmic idea on the last page of this document.

```
erl2:polyadd([],[2.0,4.0,6.0]).
[2.0,4.0,6.0]
erl2:polyadd([2.0,4.0,6.0],[]).
[2.0,4.0,6.0]
erl2:polyadd([2.0,4.0,6.0],[2.0,4.0,6.0,8.0]).
[4.0,8.0,12.0,8.0]
```

```
erl2:polydiff([],[2.0,4.0,6.0]).
[-2.0,-4.0,-6.0]
erl2:polydiff([2.0,4.0,6.0],[]).
[2.0,4.0,6.0]
erl2:polydiff([2.0,4.0,6.0],[2.0,4.0,6.0,8.0]).
[0.0,0.0,0.0,-8.0]

erl2:polyeval([2.0],2.0).  % evaluating f(x) = 2 for x = 2
2.0
erl2:polyeval([1.0,2.0],2.0). % evaluating f(x) = 2x+1 for x = 2
5.0
erl2:polyeval([0.0,0.0,3.0],2.0). % evaluating f(x) = 3x^{2} for x = 2
12.0
```

3. 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
- *selectionSort* 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*.

```
erl2:selectionSort([-1.0,-2.0,-4.0]).
[-1.0,-2.0,-4.0]
erl2:selectionSort([]).
[]
erl2:selectionSort([5.0]).
[5.0]
erl2:selectionSort([3.0,5.0]).
[5.0,3.0]
```

Horner's Rule for polynomial evaluation

Let your polynomial, as an example, be $2x^3 + 7x^2 + 5x + 4$

You will receive this as [4,5,7,2] - note the coefficients will come in reverse order (this makes the problem easier to solve actually).

The polynomial written in reverse still means the same: $4 + 5x + 7x^2 + 2x^3$.

From this we can factor out an x and rewrite as:

$4 + x * (5 + 7x + 2x^2)$

Notice in the last line that everything in the parentheses () is itself a polynomial, of degree one less than before

This can be rewritten, factoring out another x from the smaller polynomial

$4 + x * (5 + x * (7 + 2x))$

This can be rewritten, factoring out another x from the smaller polynomial

$4 + x * (5 + x * (7 + x * (2)))$

Now we can't break it down any further.

Thus, it appears that evaluating $4 + 5x + 7x^2 + 2x^3$ at some value x is equivalent to evaluating this formula: $4 + x * (5 + 7x + 2x^2)$, and that formula can be evaluated recursively as everything in the parentheses is itself a polynomial.

In breaking the list/polynomial apart, note that the 4 is the head of the list you are fed – [4,5,7,2] – representing the polynomial, x is an input value, and the smaller polynomial – [5,7,2] – to recursively evaluate is the tail of the list representing the original polynomial.