Laboratory: Erlang (2)

Summary: We continue our exploration of the Erlang programming language, focusing on list comprehensions and some higher-order functions.

Prerequisites: The first Erlang lab. Tate, Section 6.3.

Contents:
- Preparation
- Exercises
  - Exercise 1: Sums and Differences
  - Exercise 2: Iterating List Comprehensions
  - Exercise 3: Finding Points
  - Exercise 4: Inner Product
  - Exercise 5: Modifying Tuples
- For Those with Extra Time
  - Extra 1: Association Lists
  - Extra 2: More Hop-ing

Preparation

a. Create a directory for the lab.

b. Open a browser window on Tate’s examples, in case you want to try any of them.

Exercises

Exercise 1: Sums and Differences

In the section on foldl, Tate defines an expression to sum numbers something like the following:

```erlang
1> Numbers = [1,2,3,4].
[1,2,3,4]
2> lists:foldl(fun(X,Accumulator) -> X + Accumulator end, 0, Numbers).
10
```

a. Turn that code into a function, \texttt{sum(L)} or \texttt{Sum(L)}, that sums the elements of a list of numbers.

b. Consider the following definition of a function, \texttt{difference(L)}, which, given a list, \([N1,N2,N3,...,Nn]\), is supposed to compute \(N1-N2-N3-...-Nn\).
Difference = fun(L) ->
lists:foldl(fun(X,Accumulator) -> X - Accumulator end, 0, L)
end.

c. What do you expect the result of Difference([1,2,3,4]) to be?
d. Check your answer experimentally.
e. It is likely that you’ve found an error in the code. Repair that error.
f. Check your new function on the lists
   - [1,2,3,4]
   - [1,2,3]
   - [1,2]
   - [1]
   - []

And yes, it’s okay if you fail to match the final list.

**Exercise 2: Iterating List Comprehensions**

As you’ve noted, you can have multiple predicates in a list comprehension. One of you asked the very sensible question of which order the predicates are evaluated. Let’s consider that experimentally.

Consider the following function:

5> P = fun(X) -> io:format("~p~n", [X]), X end.

a. What does this function do?
b. What do you expect the result of the following to be?

6> lists:map(P, [1,2,3]).
c. Check your answer experimentally.
d. Consider the following expression.

7> [X || X <- [1,2,3,4,5], P(X) < 5, P(X) > 1].
i. If Erlang runs each predicate on each element of the list, what output would you expect?

ii. If Erlang runs each predicate on the whole list before passing it on to the next predicate, what output would you expect?

iii. Consider whether there are other possible orders of evaluation.
e. Experimentally determine which order of evaluation Erlang uses.

**Exercise 3: Finding Points**

As you know, we often describe areas on the plane in terms of functions of (X,Y) pairs. For example, a circle centered at (A,B) with radius R has the equation $A^2 + B^2 \leq R^2$. How do we identify points in such regions? A brute force approach is to pick a larger region and find out which points are and are not in the region.

a. Write a list comprehension that computes the points with integer coordinates that are within the circle of radius 5 centered at (2,3).

b. Write a list comprehension that computes the points with integer coordinates that are within an ellipse of width 8 and height 4.

c. Write a function that takes five parameters: a binary predicate, a minimum X, a maximum X, a minimum Y, and a maximum Y, and computes all points for which the predicate holds.

d. Rewrite your answers to a and b using the new predicate.

**Exercise 4: Inner Product**

As you may recall, on the first day of class, we noted that the inner product of two vectors, represented as lists A and B, is very easy to write in Scheme.

$$(\text{fold } + (\text{map } \times A B))$$

Write a similar Erlang solution to that problem.

**Exercise 5: Modifying Tuples**

As you may recall, in many languages, we use Tuples (aka Vectors, Arrays) instead of lists for two reasons:

- Tuples provide constant-time access to individual elements.
- Tuples are mutable.

Scala’s `setelement` function sets elements in tuples.

a. What value do you expect for the following expression?

```scheme```
> setelement(1, {1, 2, 3, 4}, "X").
```

b. Check your answer experimentally.

c. What value do you expect for the following set of expressions?
d. Check your answer experimentally.

e. What value do you expect for the following expression (assuming that you’ve already done the code in step c)?

> setelement(2,X,"Y").

f. Check your answer experimentally.

g. After the code in steps c and e have been executed, what value do you expect X to have?

h. Check your answer experimentally.

**For Those with Extra Time**

If you find yourself with extra time, do one or more of the following.

**Extra 1: Association Lists**

As you may remember from your time in Scheme, one common implementation of the dictionary (map, hash, table) ADT is an association list, a list of key/value pairs. For example, we might represent information about this class as follows:

```plaintext
[ {number, "CSC302"} , {name, "Programming Languages"} , {semester, "2011S"} , {prof, "Rebelsky"}, ]
```

a. Write a function, `get(AList, Key)`, that gets the value associated with a key. (It’s fine if your program crashes if the key is not there.)

b. Write a function, `put(AList, Key, Value)`, that associates `Value` with `Key`.

**Extra 2: More Hop-ing**

Implement a library of common higher-order procedures (sectioning, map, folding, etc.).