Laboratory: Erlang (1)

Summary: We begin our exploration of the Erlang programming language.

Prerequisites: Tate, Sections 6.1 and 6.2.

Contents:

- Preparation
- Exercises
  - Exercise 1: Factorial and Fibonacci
  - Exercise 2: Fun with Functions
  - Exercise 3: Pretty Patterns
  - Exercise 4: Some Sums
  - Exercise 5: Recursion and Tail Recursion
  - Exercise 6: Some Numeric List Utilities
  - Exercise 7: Association Lists
  - Exercise 8: Compose
- For Those with Extra Time
  - Extra 1: Big Numbers
  - 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: Factorial and Fibonacci

In Tate’s Yet Again example, he notes that Erlang’s computation is very fast. ("[T]he results are absolutely instantaneous.")

Check his claim.

Exercise 2: Fun with Functions

Here’s a simple “make a pair” function.
pair(X) -> { X, X }.

a. Create a module, load it, and verify that the function works.
b. What do you expect to have happen if you try to define the function in the interpreter?
c. Check you answer experimentally.

Exercise 3: Pretty Patterns

As Tate tells us, Erlang lets us use patterns in our function definitions. You’ve seen examples of patterns in the factorial example.

a. Write a definition of the following silly function:
   - f(3) is 4.
   - f(4) is 3.

b. Verify that f(3) is 4 and that f(4) is 3.
c. What value do you expect for f(2)?
d. Check your answer experimentally.
e. What do you expect to have happen if we add another rule (at the end) that f(3) is 3?
f. Check your answer experimentally.

Exercise 4: Some Sums

a. Define a function sum(N) that inefficietly computes the sum of the numbers from 0 to N using the following strategy.
   - The sum of the numbers from 0 to 0 is 0.
   - The sum of the numbers from 0 to N is N plus the sum of the numbers from 0 to N-1.

b. Verify that your function works correctly on reasonable inputs.
c. It is likely that your definition looked something like the following:

```erlang
sum(0) -> 0;
sum(N) -> N + sum(N-1).
```

What do you expect to have happen if we reverse the lines of the definition?

```erlang
sum(N) -> N + sum(N-1).
sum(0) -> 0;
```
d. Check your answer experimentally.

e. Define a function \( \text{sum}(L, U) \) that inefficiently computes the sum of the numbers from \( L \) to \( U \). (Use direct recursion; You may not use your singleton \( \text{sum} \) procedure.)

**Exercise 5: Recursion and Tail Recursion**

In our explorations of previous languages, we’ve found that normal recursion can create a stack overflow, but for well-designed tail-recursive procedures, there is no stack overflow. Determine what Erlang does for deeply recursive evaluation and deepy tail recursive evaluation.

**Exercise 6: Some Numeric List Utilities**

As you know from your experience in Scheme, it’s always a good idea to write a library of utilities for processing lists, particularly numeric lists. Let’s write some simple list procedures. (Don’t worry about taking advantage of higher-order strategies right now.)

a. Write a function, \( \text{smallest}(L) \), that finds the smallest element in a nonempty list.

b. Write a function, \( \text{lsum}(L) \), that finds the sum of elements in a potentially empty list.

c. What do you expect the result of the following to be, provided you substitute in the name of your module?

\[
\text{> module:smallest("Alpha", "Beta", "Gamma", "Delta", "Aardvark", "Foxtrot").}
\]

(Substitute in the name of your own module.)

d. Check your answer experimentally.

e. What do you expect the result of the following to be, provided you substitute in the name of your module?

\[
\text{* module:sum("A", "B", "C", "D", "E").}
\]

f. Check your answer experimentally.

**Exercise 7: 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:

\[
[ \{\text{number, "CSC302"} \}, \{\text{name, "Programming Languages"} \}, \{\text{semester, "2011S"} \}, \{\text{prof, "Rebelsky"} \}]
\]
a. Write a function, \texttt{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, \texttt{put(AList, Key, Value)}, that associates Value with Key.

**Exercise 8: Compose**

One of the first higher-order functions we typically write is \texttt{compose}, which composes two functions.

a. Write that function in Erlang.

b. Was it easier or harder to write it in Erlang than other languages? Why?

**For Those with Extra Time**

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

**Extra 1: Big Numbers**

In his section on functions in Erlang, Tate writes “I don’t know what the maximum integer size is, but I’m going to go out on a limb and say it’s big enough for me.”

Determine what the maximum integer size is.

**Extra 2: More Hop-ing**

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