Programming languages: Erlang introduction

Introduction

In these notes, the basics of the Erlang programming language will be introduced. Erlang falls into a class of programming languages called *functional programming languages.* The core idea behind functional language programming is the development of function definitions which can be evaluated both as and within expressions. Higher level programs are built by combining functions according to standard mathematical principles.

There is much less emphasis on control structures like loops in functional languages – in fat, loops actually will be replaced by recursive functions. Many other standard language issues, such as memory management, are also all handled by the language system itself. This approach provides for a relatively simple, yet very powerful, programming environment.

"Hello World" in Erlang

Let's examine the traditional "first program" for many languages, "Hello World", in Erlang.

```
% Hello World Program in Erlang
-module(helloworld).
-export([main/0]).
main() -> io:format("Hello World\n").
```

The first line of the program is an Erlang comment. Comments are started with a % and run to the end of the line. The second line of the program is a declaration of the name of the module (a group of functions) being written. It must match the name of the file (minus the .erl) the functions are being saved in, so this module would need to be saved in the file *helloworld.erl*.

The last line defines the main function. The arrow, $\rightarrow$, indicates that main should be implemented by the code following the arrow, which in this case is to use the format function from Erlang's built0in io module and apply it to the string "Hello World". This is much like a *cout* statement in C++.

All "complete" Erlang statements should end with a period to indicate a terminated statement (complete is a loose term, as you will notice below when function definitions are covered).

Finally, back-stepping a bit in the code, the third line indicates a list of the functions that are to be made accessible from this file to other modules and to the Erlang interpreter. Within the square brackets, each function being exported should be listed, comma separated, as *functionName/arity*, where *arity* indicates the number of parameters to the given function. For this example, the *main* function takes no parameters so it has an arity of 0. Since there are no other functions in the file besides *main*, only *main* is listed as being exported.

Compiling and running Erlang programs

To execute this program, one needs to instantiate the Erlang interpreter and feed it this input file as the program to run. This can be achieved by running the following command at your Unix prompt:

```
erl
```

Once the *erl* interpreter has started, you can compile (syntax check & load) a code file in the directory you are in by using the command

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

The Erlang interpreter on the surface is a very simple, line by line interpreter. It is possible to type expressions to be evaluated directly into the interpreter (such as 4+2.). For us, it will be easier to develop an entire program within an external file via gedit, sublime, or another editing tool. We can then load that program using the syntax shown above.

After a successful compile, the program can be executed by using the following syntax within the interpreter.

```
helloworld:main().
```

This is calling the given function by using the module name (helloworld), the function name (main) and providing any necessary parameters (none in this case).

The output of the Hello World program is as follows:

```
Hello World
ok
```

The output above indicates that the program first successfully printed the string we requested to be printed with the io.format function and that the function returned without error.

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

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

Types in Erlang

The built-in types in Erlang are : integers, floats, characters, strings, atoms, and Booleans.

Integers and float (floating point numbers) look and act as other languages.

Characters are represented as *$charOfInterest* and can be compared (as they ultimately are treated like integers). *String* values are quoted ordered sets of characterss, such as “foo” and “BAR”. These are actually converted into lists of integers (but we’ll come back to that later).

true and *false* are used to comprise the Boolean values. These are *atoms*. Notice atoms look like words, but are not strings. They are represented by words starting with lowercase letters.

Operators

The operators for Erlang are defined in three sections below. The order of operations across the three section is arithmetic > comparison > logical. Parentheses can change the order of operation.

Arithmetic operators (ordered highest precedence to lowest)

Description	Symbols
Unary positive, negative	+,-
Binary multiply/divide	∗ (multiplication), / (real division), div (integer division), rem (modulus)
Binary add/subtract	+,-

Comparison operators

Description	Symbols
Less than	<
Less than or equal	=<
Greater than	>
Greater than or equal	>=
Equal to	==
Not equal to	/=

Logical operators

Description	Symbols
And	andalso
Or	orelse
Not	not
Exclusive or	xor

Lists

A list contains multiple items, similar to an array, but the items can be of varying types and a list can contain list substructures if desired. The syntax is as follows:

```
[data, data, data, ... ]
```

A list of the integers seven through ten would be defined as

```
[7,8,9,10].
```

Lists should be thought of as recursive data-structures. In this context, a non-empty list should be thought of as a head element, followed by a smaller tail list (which could possibly be empty).

The empty list is designated as a pair of empty brackets:

```
[]
```

The length of a list can be returned using the function *length([list goes here])*.

```
length([7,8,9,10]). // evaluates to 4
length([7,8,[9,10]]). // evaluates to 3 (the three items are 7, 8, and the list [9,10])
```

While there is a indexing function for lists (lists:nth(index,[list goes here])), you will be asked to ignore it for this class. Instead, remember that the head of a list is defined as the first element, while the tail is defined as the list of all elements in their original order except the first element. (So in reality, lists are kind of like linked lists!). These can be retrieved using the functions *hd* and *tl* respectively.

```
hd([2,3,4]). // evaluates to 2
tl([2,3,4]). // evaluates to [3,4]
hd([5]). // evaluates to 5
tl([5]). // evaluates to []
```

One combine data together into a single list. The cons operator | (the bar symbol), when used within square brackets, takes a single element and a list and prepends the element onto the front of the list. It only works for pre-pending!

```
[2|[3,4]]. // evaluates to [2,3,4]
[1|[2|[3,4]]]. // evaluates to [1,2,3,4]
```

The list concatenation operator *++* takes two whole lists and combines them into a new list:

```
[2,3]++[4,5]. // evaluates to [2,3,4,5]
```

If-else

The *if-then-else* syntax has the following basic format.

```
if
    conditional1 -> expressionA;
    conditional2 -> expressionB;
    ...
    true -> expressionZ
end
```

This format can be extended, but this is what we'll start with first. We will also limit conditionals to being Boolean expressions. While it is possible to call functions as part of the conditional, you can NOT call your own functions. The reason is that the functions in an if-statement test have to be side-effect free, and that is not guaranteed for your functions – it is for built in functions though!. A final *true* conditional is required to support catching all cases not explicitly covered by earlier conditionals. Each line except for the *true* conditional requires a semicolon at the end (and the *true* conditional line cannot have one). Finally, the keyword *end* is required to mark the end of the if-statement.

Variables

Variables are represented by words starting with an uppercase letter. The only context we'll see them in today is as parameters to functions.

Defining functions

Learning how to define functions is the key to learning the Erlang language. Whenever you feel like you should write an iterative (looping) block of code to solve a problem, Erlang will say to write a function (preferably one that is recursive).

The basic way to declare a function has the following syntax:

```
<identifier>(<parameter list>) ->
<expression>.
```

The identifier representing the function name has to start with a lowercase letter. There can be zero or more parameters, separated by commas. Finally, note that in all of today's code a period is only used at the very end of the function definition, even if there is a fairly complicated set of statements that make up the part of the implementation.

Here's an example of a simple function definition to return the square of a number:

```
square(N) ->
  N * N.
square(3.0). // evaluates to 9.0
```

Any number of parameters to a function are allowed. Below is a function which takes three parameters and returns the maximum value.

```
max3(A,B,C) ->
     if A > B -> if A > C -> A;
                   true -> C
                end;
       true -> if B > C -> B;
                 true -> C
               end
     end.
```

Most of the real work done in Erlang programming is through recursive functions. We will use recursion as a replacement for while and for loops. As with all languages that support recursion, it is necessary to remember the two key ideas behind recursive functions:

1. The solution of large problems is found by combining solutions for smaller problems,
2. One requires the presence of a base case which can't be decomposed further and which has an immediate (non-recursive) solution.

Let's look at a function which can reverse a list:

- Base Case: The list is empty – the reverse of an empty list is the empty list
- Recursive Case: The list is non empty – the reverse of a non-empty list is the first element of the list attached to the end of the reverse of the rest of the list.

If our list was [1,2,3], the reversed list is [3,2,1]. This can be achieved through the recursive case above by nothing that [3,2,1] is the concatenation of (the reverse of [2,3]) and (the single element list [1]).

What types of functions do we know of that could be useful in defining this function?

- The *hd* function will remove the 1 from the list as a single element.
- The *tl* function will return the [2,3] part of the list as a list.
- To concatenate two lists, one can use the ++ operator.

Given what's above, we have a list [2,3] and a single element 1. To make 1 into a list, you could append it to the empty list, using the | (cons) operator within square brackets.

Given these operators, here's a valid *reverse* definition:

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

and a sample of execution:

```
reverse([1,2,3]). // evaluates to [3,2,1]
```