Erlang Project

Chris McClure

29 April 2019

1 Introduction

1.1 History

Erlang was first created in 1986 and had potential for prototyping telephone exchanges in 1988, but it wasn't quite fast enough. After several years of refining the language, Erlang started compiling their language to C for a mixture of performance and disk space. Created by Joe Armstrong, Robert Virding, and Mike Williams; It was originally a proprietary software for the Swedish telecommunications network owned by *Ericsson*. Erlang became open-source in 1998 and is still updated and maintained to this day. Although not a very popular language, Erlang is being used in large companies; namely, *WhatsApp*. *WhatsApp* uses Erlang because of it's impressive performance when it comes to concurrency, which we'll cover later.

1.2 Overall Characteristics

Erlang was first created using Prolog and much of the syntax is still very similar. Erlang is compiled into byte code called *BEAM*. The language is useful in a couple of different aspects. Erlang is a dynamically typed language simply because they believe that any failure in a component shouldn't affect everything else. This is an important feature because they knows errors are bound to happen, but won't stop a system from running because of them. Erlang supports hot swapping code, meaning that new code can be written and upgraded on a machine without ever taking the machine offline. This is particularly useful if bug fixes need to be implemented and the machine can't afford to go down. Erlang is also very robust, if any node fails, it can automatically fail over to another. Being a functional programming language with immutable data, many operations require a couple more steps to perform, as opposed to their imperative counterparts.

2 Building and Execution

2.1 Acquisition

Windows and Linux downloads for Erlang can be found at `https://www.erlang.org/downloads` and installed by following the installation instructions after download. Erlang is easily installed on Mac with the assistance of Homebrew. If Homebrew has never been installed on your machine, youâĂŹll need to install it from `https://www.brew.sh` before proceeding. Once Homebrew is installed, simply type **brew install erlang** in the terminal. Windows users may need to add Erlang to the PATH.

2.2 Syntax

Before we begin writing some code in Erlang, we should verify that the Erlang VM was installed correctly. Open up Terminal or Command Prompt, and type **erl**.

```
erl
Erlang/OTP 21 [erts-10.3.4] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threa
ds:1] [hipe] [dtrace]

Eshell V10.3.4  (abort with ^G)
[1>
```

2.2.1 Expressions

Erlang has support for all of the arithmetic operations that you'd expect. One thing to note is that '/' does floating-point division where as 'div' uses integer division. Expressions can include commas and semicolons, which are similar to **andalso** and **orelse**, but they must always end with a period, otherwise they won't be evaluated.

```
[1> 5 + 2.
7
[2> 5 - 2.
3
[3> 5 * 2.
10
[4> 5 / 2.
2.5
[5> 5 div 2.
2
```

2.2.2 Simple Variables

Variables in Erlang are bound by the '=' symbol. They are immutable and must begin with an uppercase letter. If an attempt to is made to use a variable that hasn't been bound to a value, an error will be returned. The reason why they can't begin with a lowercase is because of atoms. Atoms are essentially what you see is what you get, if you create an atom *alpaca* and pass it around, you'll get *alpaca* back (assuming you didn't encounter in an error in the function you passed it to).

```
1> A = 4.
4
2> A.
4
3> alpaca = 10.
** exception error: no match of right hand side value 10
4> alpaca.
alpaca
5> B.
* 1: variable 'B' is unbound
6>
```

There is a key thing to understand about the '=' symbol in Erlang. Because data is immutable, '=' returns the value associated with the variable only if it's true. Therefore, if we decided to declare a variable **B = 4.** and evaluate it with our previously bound variable **A**, by stating **A = B.**, we'd get 4 back. Otherwise an error would be thrown stating that there was no match of the right hand side value.

2.2.3 Creating Files

To make an Erlang file, create an empty file named *tut.erl*. At the beginning of the file, type **-module(tut)**. This creates a reference to the file that the Erlang VM can see. Any functions that you'd like to placed in the module should lie below it.

```
-module(tut).
```

2.3 Compilation

Before we get into functions, let's make sure that the file actually compiles. Open the Erlang VM in the command line. Type **c(tut).**.

```
1> c(tut).
{ok,tut}
2>
```

2.3.1 Functions

Before using a function, it needs to be exported **-export([funcName/numArgs, ...]).**. The export command creates a list of functions that should be seen by the outside world. If the function doesn't need to be called directly, it shouldn't be exported.

```
-module(tut).
-export([add2/2]).

% simple function that returns the sum of two arguments
add2(X, Y) ->
    X + Y.
```

Declaring functions is simple, they follow the syntax **Name(Args)→Body.**, where the *Name* is an atom and *Body* can be one or multiple expressions that are separated by commas.

```
1> c(tut).
{ok,tut}
2> tut:add2(-5 , 2).
-3
```

2.3.2 Flow of Control

Loops are a little more difficult if you're migrating from an imperative language, but not trivial to understand. Loops are done through recursive calls until the base case is satisfied. In the factorial example, if the value passed is 0, it matches the first pattern and returns 1. Otherwise it recursively calls itself with N-1 as the parameter until N is 0, and then returns the factorial. In the case of the argument being 3, the recursive call would look something like this.

```
%% returns the factorial of a passed number
factorial(0) ->
    io:format("~nFactorial:~n"),
    1;
factorial(N) ->
    if N < 0 -> 0;
       true -> N*factorial(N-1)
    end.
```

$$3 * factorial(3-1)$$
$$3 * 2 * factorial(2-1)$$
$$3 * 2 * 1 * factorial(1-1)$$
$$3 * 2 * 1 * 1 = 6$$

```
-export([add2/2, old/1]).

old(N) ->
    if N =< 40 -> "not old yet...";
       true -> "officially old"
    end.
```

```
[2> tut:old(40).
"not old yet..."
[3> tut:old(41).
"officially old"
[4>
```

If statements are a little different than most programming languages. The fundamental idea is the same (evaluate an expression if it's true, otherwise do something else). However, the syntax is different. As stated earlier, Erlang requires everything to return something. So we need a catch all statement (equivalent to *else*) that returns something in case of the *if* statement failing. This comes to us as the **true**→ branch.

2.3.3 Guards

Guards are another flow control operation that act similar to if statements. In the provided example it's obvious that the expression will evaluate to true as long as N is greater than or equal to 40. If passed "_" (which represents anything else we don't particularly care for), then we evaluate that expression.

```
[2> tut:still_old(40).
"you're still old..."
[3> tut:still_old(39).
"you're not old yet."
[4>
```

```
% simple guard statement
still_old(N) when N >= 40 -> "you're still old...";
still_old(_) -> "you're not old yet.".
```

2.4 Lists

Lists in Erlang are pretty easy to understand. Lists aren't limited to just one type, they can be mixed around such as **[bob, 1, 3, [hello, "no", -7.3]]**.. The list is separated into two sections, the head and the tail. Retrieving these values are as simple as $[Head|Tail] = SomeList$. .

```
head([H|_]) -> H.
second([_,X|_]) -> X.
tail([_|T]) -> T.
```

```
2> functions:head([bob, 1, 3,[hello, "no", -7.3]]).
bob
3> functions:second([bob, 1, 3,[hello, "no", -7.3]])
1
4> functions:tail([bob, 1, 3,[hello, "no", -7.3]]).
[1,3,[hello,"no",-7.3]]
```

The head will now store the first value in the list, whereas the tail will store all of the rest. Iterating through a list means using recursive calls on the head and tail of the list until each element of the tail has been removed and all that remains is the empty list.

3 Special Features

3.1 Pattern Matching

Pattern matching is a useful aspect of flow control that let's the compiler choose the correct pattern of the function that is being called. Rather than entering a function and checking to see if an argument is equal to some value and then doing something, we can skip the comparison step all together. Should the argument be equal to a function pattern, the evaluation would take place immediately. It should be noted that the the last pattern ends with a period, whereas the previous patterns end in a semicolon. This is because the semicolon loosely represents *or else do this*.

```
pmatch(4) -> "FOURRR!!!";
pmatch(_) -> "no four...".
```

```
[8> functions:pmatch(dog).
"no four..."
[9> functions:pmatch(4).
"FOURRR!!!"
```

3.2 Concurrency

Concurrency is one of Erlang's most important features. It helps keep processes that are running reliable. Because processes don't share memory with each other, there isn't a chance that they can overwrite each others memory. However, because the operating system is busy all the time and can't be trusted to keep a reliable performance, processes are created and

ran by the Erlang VM. Depending on the number of cores in your machine, you could have have several processes or just a few running at the same time. Each core gets a scheduler that allots a process a certain amount of time, as well as load-balancing processes in case one scheduler has several more than another.

To create a function process, the Erlang function **spawn/1** needs to be used. Spawn takes a function and returns a process ID (PID). The PID is an arbitrary number that represents a process that could have or still exists. This value is the address to communicate with the process. The PIDs are represented a little more clearly in the timing function below. There's a list of 10 PIDs, each corresponding to a value that is displayed. Because processes are being evaluated at the same time, the corresponding values aren't done in an incremental order. Whatever process is ready to go goes.

```
9> Square4 = fun() -> 4 * 4 end.
#Fun<erl_eval.20.128620087>
10> spawn(Square4).
<0.94.0>
```

```
1> A = fun(X) -> timer:sleep(10), io:format("~p~n", [X])end.
#Fun<erl_eval.6.128620087>
2> [spawn(fun() -> A(X) end) || X <- lists:seq(1,10)].
[<0.81.0>,<0.82.0>,<0.83.0>,<0.84.0>,<0.85.0>,<0.86.0>,
 <0.87.0>,<0.88.0>,<0.89.0>,<0.90.0>]
1
2
3
4
5
7
6
8
10
9
```

3.3 List Comprehensions

A simple example of a list comprehension displaying a sequence of numbers can see in the figure above. Erlang's use of comprehensions is an easy and practical way to perform operations over a list. The syntax is very similar to set notation.

$$[N \| N \leftarrow [elem1, elem2, elem3, ...], CONDITION].$$

4 Conclusion

Erlang is great for it's robustness and use of concurrency. Given that it can automatically fail over to a different node in a distributed system when an error occurs; it's appealing to companies that they don't have to take their system down for bug fixes. Erlang was created to interact with machines in real-time, making it a reliable choice when monitoring data is a primary concern for a company.

Although there are many benefits to Erlang, there are a few drawbacks. For example, Erlang doesn't handle computation-heavy operations on large lists quickly. Secondly, it isn't

feasible to develop websites or applications that don't provide real-time communications. Erlang was created to efficiently pass along messages via processes to multiple servers at a time; attempting to bottleneck the speed of those processes would result in an inefficient system.

Erlang was designed to provide quick, real-time communications, and they've accomplished that. Although it's not a well known or mainstream language today, I believe that with it's continued support, it has potential to become one. The language is easy to understand, has several benefits, and is already helping pave the way for quick and reliable communications.