Introduction to programming in Erlang, Part 1: The basics

Martin Brown May 10, 2011

Erlang is a multi-purpose programming language used primarily for developing concurrent and distributed systems. It began as a proprietary programming language used by Ericsson for telephony and communications applications. Released as open source in 1998, Erlang has become more popular in recent years thanks to its use in high profile projects, such as the Facebook chat system, and in innovative open source projects, such as the CouchDB document-oriented database management system. In this article, you will learn about Erlang, and how its functional programming style compares with other programming paradigms such as imperative, procedural and object-oriented programming. You will learn how to create your first program, a Fibonacci recursive function. Next, you will go through the basics of the Erlang language, which can be difficult at first for those used to C, C++, Java™, and Python.

What is Erlang?

Erlang was developed by Ericsson to aid in the development of software for managing a number of different telecom projects, with the first version being released in 1986, and the first open source release of the language in 1998. You can see this in the extended Erlang release information where the Open Telecom Platform (OTP), the application development platform for Erlang, exists as the primary method of delivering the Erlang development environment.

Erlang provides a number of standard features not found in or difficult to manage in other languages. Much of this functionality exists in Erlang because of it's telecom roots.

For example, Erlang includes a very simple concurrency model, allowing individual blocks of code to be executed multiple times on the same host with relative ease. In addition to this concurrency Erlang uses an error model that allows failures within these processes to be identified and handled, even by a new process, which makes building highly fault tolerant applications very easy. Finally, Erlang includes built-in distributed processing, allowing components to be run on one machine while being requested from another.

Put together, Erlang provides a great environment to build the type of distributed, scalable, and high-performance discrete applications that we often use to support modern network and web-based applications.

Functional programming versus other paradigms

A main difference between Erlang and more popular languages is that Erlang is primarily a functional programming language. This has nothing to do with whether it supports functions, but is related to how the operation of programs and components works.

With functional programming, the functions and operations of the language are designed in a similar way to mathematical calculations, in that the language operates on functions taking input and generating a result. The functional programming paradigm means that the individual blocks of code can produce consistent output values for the same input values. This makes predicting the output of the function or program much easier and, therefore easier to debug and analyze.

The contrasting programming paradigm is the imperative programming language, such as Perl, or Java, which rely on changing the state of the application during execution. The change of state in imperative programming languages means that the individual components of a program can produce different results with the same input values, based on the state of the program at the time.

The functional programming approach is easy to understand, but can be difficult to apply if you are used to the more procedural and state-focused imperative languages.

Get Erlang

You can get Erlang directly from the Erlang website (see Related topics). Many Linux distributions also include it within their repositories. For example, to install on Gentoo, you can use: `$ emerge dev-lang/erlang`. Or you can install Erlang on Ubuntu or Debian distributions using: `$ apt-get install erlang`.

For other UNIX® and Linux platforms you can download the source code and build it by hand. You will need a C compiler and the make tool to build from source (see Related topics). The basic steps follow:

1. Unpack the source: `$ tar zxf otp_src_R14B01.tar.gz`
2. Change into the directory: `$ cd otp_src_R14B`
3. Run the configure script: `$ ./configure`
4. Finally run make to build: `$ make`

A Windows® installer is available from the Erlang website too (see Related topics).

Your first Erlang program, a recursive Fibonacci function

A perfect way to see the benefits of the functional programming style, and how this works within Erlang, is to look at the Fibonacci function. The Fibonacci numbers are an integer sequence where for a given number, the Fibonacci value can be calculated using: `F(n) = F(n-1) + F(n-2)`.

The result of the first value, `F(0)`, is `0`, and the result of `F(1)` is `1`. After this, you can determine `F(n)` by calculating the previous two values and adding them together. For example, the calculation for `F(2)` is shown in Listing 1.

Listing 1. Calculation for `F(2)`

```
F(2) = F(2-1) + F(2-2)
F(2) = F(1) + F(0)
F(2) = 1 + 0
F(2) = 1
```

The Fibonacci series is an important calculation and shortcut for a number of systems, including the analysis of financial data, and as a basic for the arrangement of leaves on a stem or branch of a tree. If you play video games with 3D trees, the arrangement of the branches and leaves were probably calculated using the Fibonacci series to determine the location of the branches and leaves.

When programming the calculation in a programming language, the calculation can be achieved by using recursion, where the function calls itself in order to calculate the numbers from the roots, `F(0)` and `F(1)`.

In Erlang you can create a function with variables and with fixed values. This simplifies the situation in the Fibonacci series where `F(0)` and `F(1)` return explicit values, rather than a calculated one.

Thus, the basic function has three situations, the one when supplied `0`, another when `1`, and another when any higher value. In Erlang, you separate these statements using a semicolon, so the basic Fibonacci function can be defined as shown in Listing 2.

Listing 2. The basic Fibonacci function

```
fibo(0) -> 0 ;
fibo(1) -> 1 ;
fibo(N) when N > 0 -> fibo(N-1) + fibo(N-2) .
```

The first line defines the result of calling fibo(0) (the `->` separates the definition and the body of the function), the second line defines the result when calling `fibo(1)`, and the third line defines the calculation to execute when we have been supplied a positive value of `N`. This works because of a system in Erlang called pattern matching, which we will look at in more detail later. Note how the final statement (and all statements in Erlang) is terminated by a period. The actual calculation is straightforward.

Now, let's take a closer look at the structure of the Erlang language.

The basics

If you are used to languages like Perl, Python, or PHP then the structure and layout of Erlang will seem slightly odd, but there are some aspects that make the entire process of writing any application a lot simpler, with you having to worry less about many aspects of the code. In particular, Erlang can look very sparse compared to other languages, and certain operations, expressions, and constructs are frequently shown on only one line.

The easiest way to get to know Erlang is by using the Erlang shell, which you can execute by running `erl` at the command line after Erland has been installed. You can see this in Listing 3.

Listing 3. Using the Erlang shell

```
$ erl
Erlang R13B04 (erts-5.7.5) [source] [rq:1] [async-threads:0]

Eshell V5.7.5  (abort with ^G)
1>
```

You can use the prompt to enter statements, which you should terminate with a period. The shell evaluates the statement when completed. Thus, entering a simple sum returns the result as shown in Listing 4.

Listing 4. Entering a simple sum

```
1> 3+4.
7
```

We'll use the shell to look at some of the different types and constructs.

Number types

Erlang supports basic data types such as integers, floats and more complex structures, like tuples and lists.

Integers, and most integer operations, are the same as in other languages. You can add two numbers together, as shown in Listing 5.

Listing 5. Adding two numbers together

```
1> 3+4.
7
```

And you can use parentheses to group calculations together, as shown in Listing 6.

Listing 6. Using parentheses to group calculations

```
2> (4+5)*9
2> .
81
```

Note that in Listing 6, the terminating statement period is on a separate line and evaluates the preceding calculation.

Floats in Erlang are used to represent real numbers, and can be expressed naturally, as shown in Listing 7.

Listing 7. Floats expressed naturally

```
3> 4.5 + 6.2 .
10.7
```

Floats can also be expressed using exponents, as shown Listing 8.

Listing 8. Floats expressed using exponents

```
4> 10.9E-2 +4.5
4> .
4.609
```

The standard mathematical operators, +, -, /, * are supported on both integer and floating point values, and you can mix and match floating-point and integers in calculations. However, an error will be raised when using an equivalent of the modulus and remainder operators on floating values, as these support only integer values.

Atoms

Atoms are static (or constant) literals. Listing 9 shows an example.

Listing 9. Atoms

```
8> abc.
abc
9> 'Quoted literal'.
'Quoted literal'
```

Atoms should be used in the same way as you would use a `#define` value in C, that is, as a clearer method of specifying or identifying a value. As such, the only valid operation on an atom is a comparison. Using atoms in this way also extends to boolean logic, with the availability of true and false atoms to identify the boolean result of a statement. Atoms must start with a lower case character, or you can delimit with single quotes.

For example, you can compare integers and get a boolean atom as the result, as shown in Listing 10.

Listing 10. Comparing integers to get a boolean atom

```
10> 1 == 1.
true
```

Or you can compare atoms, as shown in Listing 11.

Listing 11. Comparing atoms

```
11> abc == def.
false
```

Atoms are themselves ordered lexically (that is, `z` has a greater value than `a`), for example (see Listing 12).

Listing 12. Atoms ordered lexically

```
13> a < z.
true
```

Standard boolean operators are available, such as `and`, `or`, `xor` and `not`. You can also use the `is_boolean()` function to check whether the supplied value is true or false.

Tuples

Tuples are a composite data type and are used to store collections of items. Tuples are delimited by curly brackets (see Listing 13).

Listing 13. Tuples

```
14> {abc, def, {0, 1}, ghi}.
{abc,def,{0,1},ghi}
```

The contents of a tuple do not all have to be the same type, but one special construct is of a tuple where the first value is an atom. In this case the first atom is called a tag and this can be used to identify or classify the contents (see Listing 14).

Listing 14. Tuple where the first value is an atom

```
16> { email, 'example@example.org'}.
{email,'example@example.org'}
```

Here the tag is email, and the tag can be used to help identify the remainder of the content in the tuple.

Tuples are very useful for containing defined elements and describing different complex data structures, and Erlang allows you to `set` and `get` values in a tuple explicitly (see Listing 15).

Listing 15. Setting and getting values in a tuple explicitly

```
17> element(3,{abc,def,ghi,jkl}).
ghi
18> setelement(3,{abc,def,ghi,jkl},mno).
{abc,def,mno,jkl}
```

Note that the elements of the tuple are indexed with `1` as the first value, instead of 0, which is common in most other languages. You can also compare tuples in their entirety (see Listing 16).

Listing 16. Comparing tuples in their entirety

```
19> {abc,def} == {abc,def}.
true
20> {abc,def} == {abc,mno}.
false
```

Lists

The last data type is the list, which is denoted by square brackets. Lists and tuples are similar, but whereas a tuple can only be used in a comparison, lists allow a wider variety of manipulation operations to be performed.

Basic lists look like Listing 17.

Listing 17. Basic list

```
22> [1,2,3,abc,def,ghi,[4,56,789]].
[1,2,3,abc,def,ghi,[4,56,789]]
```

Strings are actually a special type of list. Erlang does not support the idea of a string directly, although you can use a double quoted value to create a string value (see Listing 18).

Listing 18. Using a double quoted value to create a string value

```
23> "Hello".
"Hello"
```

However, a string is actually just a list of ASCII integer values. Thus the above string is stored as a list of the ASCII character values (see Listing 19).

Listing 19. String stored as a list of the ASCII character values

```
24> [72,101,108,108,111].
"Hello"
```

As a shortcut, you can also specify characters using the `$Character` notation (see Listing 20).

Listing 20. Specifying characters using the `$Character` notation

```
25> [$H,$e,$l,$l,$o].
"Hello"
```

Lists, including strings (lists of characters), support a number of different methods for manipulation. This highlights the main difference between a string and an atom. Atoms are static identifiers, but you can manipulate a string by examining the component parts (each character). You cannot, for example, identify the individual words in the atom (for example, `'Quick brown fox'`) because the atom is a single entity. But a string could be split into different words: `["Quick","brown","fox"]`.

A number of functions for manipulating lists are provided in the lists module For example, you can sort the items in a list using the sort function. As these are note built-in functions, you have to specify the module and function name (see Listing 21).

Listing 21. Specifying the module and function name

```
34> lists:sort([4,5,3,2,6,1]).
[1,2,3,4,5,6]
```

List manipulation

You can construct lists with multiple elements using the constructor, which builds a list from an element and another list. In other languages, this kind of construction is handled by functions or

operators for `push()`. In Erlang, the `|` (pipe) operator is used to differentiate between the head (start of the list) and the tail, in the notation `[Head|Tail]`. Here the head is a single element, and the tail is the rest of the list.

Listing 22 shows how to add a new value to the beginning of the list.

Listing 22. Adding a new value to beginning of the list

```
29> [1|[2,3]].
[1,2,3]
```

You can repeat this for the entire list as shown in Listing 23.

Listing 23. Repeating this for the entire list

```
31> [1|[2|[3|[]]]].
[1,2,3]
```

In this example, the empty list at the end means that you have constructed a well-formed (proper) list. Note that the first item must an element, not a list fragment. If you merge the other way, you construct a nested list (see Listing 24).

Listing 24. Constructing a nested list

```
30> [[1,2]|[2,3]].
[[1,2],2,3]
```

Finally, you can merge lists using the `++` operator, as shown in Listing 25.

Listing 25. Using the ++ operator to merge lists

```
35> [1,2] ++ [3,4].
[1,2,3,4]
```

And, you can subtract each element from the list on the right of the operator from the list on the left (see Listing 26).

Listing 26. Subtracting each element from the list

```
36> [1,2,3,4] -- [2,4].
[1,3]
```

Because strings are lists, the same is true for them too (see Listing 27).

Listing 27. The same is true for strings

```
37> "hello" ++ "world".
"helloworld"
40> ("hello" -- "ello") ++ "i".
"hi"
```

Although this has been a top level flight over the data types, hopefully it will give you enough to understand the basic types and operations.

Expressions and pattern matching

We've already seen a number of different expressions and constructs while looking at the different data types. One important element of expressions is the variable. Variables in Erlang must start with a capital letter and be followed by any combination of upper and lowercase letters and underscores (see Listing 28).

Listing 28. Variables in Erlang

```
41> Value = 99.
99
```

Assignment of a variable within Erlang is a one-time binding of the value to the variable. Once you have bound the variable once, you cannot change it's value (see Listing 29).

Listing 29. Binding a value to a variable

```
42> Value = 100.
** exception error: no match of right hand side value 100
```

This is unlike most languages — the very definition of variable tends to imply that the value is variable. Single assignment in this means that if you want to calculate the result of a value you must assign it to a new variable (see Listing 30).

Listing 30. Limitations of variables in Erlang

```
43> Sum = Value + 100
199
```

The benefit of this single assignment is that it makes it more difficult to introduce or accidentally change the value of a variable during a calculation, which makes identifying a value and debugging during programming much easier. It also makes the code clearer, and sometimes much shorter, as you can simplify the structure.

Note that this operation means that the value is calculated before being bound to the variable. In other languages it's possible to set the value based on a reference to a function or operation, which means the value can be different depending on the value of the reference at the time it is accessed. In Erlang, the value of the variable is always known at the time it is created.

You can explicitly forget the bounding of a variable using `f(Varname)`, or all variables using `f()`.

Assigning values to variables is actually a special type of pattern matching. Pattern matching in Erlang also handles the execution flow of individual statements, and extracts the individual vales from compound data types (tuples, arrays). The basics of pattern matching is therefore: Pattern = Expression.

The expression is composed of data structures, bound variables (that is, those with a value), mathematical operations and function calls. The two sides of the operation must match (that is, if you have 2 element tuple as the pattern, the expression must resolve to a 2 element tuple). When the expression is executed, the expression is evaluated, and the results assigned to the pattern.

For example, we can assign values to two variables simultaneously with one pattern matching (see Listing 31).

Listing 31. Assigning values to two variables simultaneously

```
48> {A,B} = {(9+45),abc}.
{54,abc}
```

Note in the evaluation, if the pattern is a bound variable, or the element of the pattern is a bound variable, then the result of the pattern match becomes a comparison. This allows for more powerful selective assignment. For example, to get the name and email from a tuple, we can use pattern matching: `{contact, Name, Email} = {contact, "Me", "me@example.com"}`.

Finally, we can use pattern matching to pull out the elements from a list or tuple using the construct notation mentioned earlier. For example, Listing 32 shows how to get the first two elements from a list while retaining the remainder.

Listing 32. Getting the first two elements from a list while retaining the remainder

```
53> [A,B|C] = [1,2,3,4,5,6].
[1,2,3,4,5,6]
```

`A` has been assigned the value `1`, `B` the value `2`, and `C` the remainder of the list.

Functions

Functions in Erlang are the basic building blocks of any program, just as with any other language. A function is composed of the function name (defined by an atom), and zero or more arguments to the function in parentheses: `sum(N,M) -> N+M`.

Functions must be defined in within modules in files (you cannot define them from within the shell). The arguments can contain more complex types. For example, tags in tuples can be used to select different operations (see Listing 33).

Listing 33. Tags in tuples can be used to select different operations

```
mathexp({sum, N,M}) -> N+M ;
mathexp({sub, N,M}) -> N-M ;
mathexp({square, N}) -> N*N.
```

The semicolon is an or operator between each function definition. Pattern matching is used to evaluate the arguments to the function, so if you supply a tuple with three elements to the `mathexp()` function, the pattern matching will fail.

Functions in Erlang can also accept a different number of arguments, with the right definition of the function being selected by evaluating the pattern match until a valid definition of the function has been located. The number of arguments to a function is called it's arity, and is used to help identify the functions.

Looking back at our Fibonacci example, you can now see that when `fibo(0)` was called, this patten matched the first definition of the function, returning the value, and `fibo(1)` matched the second definition, while any other value matched the last definition. It also demonstrates how the recursion of the function execution works. When calling `fibo(9)`, for example, the `fibo(N)` function is called with the corresponding value until the `fibo(0)` and `fibo(1)` functions are reached, and when the fixed values are returned.

The return value of any function is the result of the last expression in that clause (in our examples there is only one line). Note that variables are assigned only when a match is found, and when the variables are local to the function.

Modules

Modules, just like modules in other languages, are used to collate similar functions together.

You specify the module name in the file (and the file name must match), and then specify the functions within the module that you want to export to other programs that load the module. For example, Listing 34 shows the file fib.erl, which contains the definition of our fib module.

Listing 34. the fib.erl file

```
-module(fib).
-export([fibo/1, printfibo/1]).

%% print fibo arg. and result, with function as parameter

printfibo(N) ->
   Res = fib:fibo(N),
   io:fwrite("~w ~w~n", [N, Res]).

fibo(0) -> 0 ;
fibo(1) -> 1 ;
fibo(N) when N > 0 -> fibo(N-1) + fibo(N-2) .
```

The module specification is in the `-module()` line. The `-export()` contains the list of functions to be exported. The definition for each function shows the function name with the arity of the function. This enables you to export very specific definitions of the function.

To use the module it needs to be compiled and loaded. You can do this within the shell using the `c()` statement, as shown in Listing 35.

Listing 35. Using the `c()` statement to compile and load a module

```
1> c(fib).
{ok,fib}
2> fib:printfibo(9).
9 34
ok
```

Note that the function call includes the module name to ensure that we are calling the `printfibo()` function within the fib module.

Conclusion

The structure and format of Erlang programs is very different to most other languages. Although many of the data types and fundamental expressions are the same, their use and application is slightly different. Single use variables, and the significance of the evaluation of different expressions as demonstrated through the pattern matching system provide some powerful extensions to the typical language environment. For example, the ability to define multiple approaches to the same functions and employ pattern matching for recursive calls simplifies some functions.

Next time we will look at the processes, message parsing, and network facilities of Erlang, and take a look at the MochiWeb HTTP server as a way of understanding the power and flexibility of the language.

Related topics

- Erlang website: Find more information about this programming language.
- Wikipedia entry for Erland: Learn more about Erland's history.
- Erlang and OTP in Action, by Martin Logan, Eric Merritt and Richard Carlsson from Manning Publications, is a book for developers interested in creating practical applications in Erlang using the OTP framework.
- Erlang Programming: A Concurrent Approach to Software Development, by Francesco Cesarini, Simon Thompson from O'Reilly Media, offers you an in-depth explanation of Erlang, a programming language ideal for any situation where concurrency, fault-tolerance, and fast response is essential.
- Programming Erlang: Software for a Concurrent World, by Joe Armstrong, from Pragmatic Programming, shows you how to write high reliability applications, even in the face of network and hardware failure, using the Erlang programming language.
- Erlang: Download Erlang programming language.
- The Apache CouchDB Project: CouchDB, which is written in Erlang and makes use of the MochiWeb HTTP server can be obtained from Apache.
- developerWorks on Twitter: Follow us for the latest news.
- developerWorks Open source zone: Find extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.