Erlang/OTP

VOLUME I

A Concurrent World

MANUEL ÁNGEL RUBIO JIMÉNEZ

Erlang/OTP

Volume I: A Concurrent World

Manuel Angel Rubio Jiménez

Translated by

Ana María Rubio Jiménez

Reviewed by

Ayanda Dube

Erlang/OTP
Volume I: A Concurrent World

Manuel Angel Rubio Jiménez

Translated by
Ana María Rubio Jiménez

Reviewed by
Ayanda Dube

Abstract

The Erlang programming language was born around the year 1986 in Ericsson laboratories by the hand of Joe Armstrong. It is a functional language based on Prolog, fault-tolerant, and oriented to real-time work and concurrency, which provides certain advantages in terms of algorithm declaration.

Like most functional languages, Erlang requires an analysis of the problem and a way to design the solution differently than it would be done in an imperative programming language. It suggests a better and more efficient way to carry it out. It is based on a syntax that is more mathematical than programmatic, so it tends more to solve problems than to order and execute orders.

All this makes Erlang a very appropriate language for the programming of critical mission elements, both at the server level and at the desktop level, and even for the development of embedded systems.

This book contains a compendium of information about what language is, how it covers the needs for which it was created, how to get the most out of its way of performing tasks and its orientation to the audience. It is a review from the beginning about how to program in a functional and concurrent way in a distributed and fault tolerant environment.

ISBN 978-84-945523-7-3

9 788494 552373

[1] http://erlang-otp.es/
[2] http://creativecommons.org/licenses/by-nc-sa/3.0/

Chapter2.The language

There are only two kinds of languages: the ones people complain about and the ones nobody uses.
—Bjarne Stroustrup

Erlang has a very particular syntax. There are people who end up liking it and other people who consider it uncomfortable. It must be understood that it is a language based on Prolog and with a touch of Lisp, so it resembles more the functional languages than the imperative ones.

Most people start programming in languages such as Basic, Modula-2 or Pascal, which have a very similar syntax between them. The same goes for the branch of C/C++, Java, Perl or PHP, which have a syntax that is also similar as well as the use of conditional, iterative blocks and function and class declaration.

In imperative languages the syntax is based on the achievement of commands that the programmer sends through the code to the machine. In Erlang and other functional languages, the syntax is designed as if it were the definition of a mathematical function or a logical proposition. Each element within the function has a purpose: to obtain a value; the set of all these values, with or without processing, makes up the result. A basic example:

```
area(Base, Height) -> Base * Height.
```

In this example you can see the definition of the function ***area***. The parameters required to obtain their result are ***Base*** and ***Height***. Parameter declaration is followed by the achievement symbol (->), as if it were a logical proposition. Finally, there is the internal operation that returns the result you want to obtain.

When dealing with mathematical functions or logical propositions, there is no correlation between imperative and functional. For a common imperative code like the following:

```
for i <- 1 to 10 do
    if nailing(i) = 'yes' then
        hammering_nail(i)
    endif
endfor
```

There is no equivalent in Erlang that can transcribe an imperative action as we know it. To develop in Erlang, you have to think about ***what*** you want to do rather than ***how***. If in a functional language you want to nail into a wall, you can select the function of ***hammering_nail***, it could be done through an understanding list:

```
[ hammering_nail(X) || X <- Nails, hammer(i) =:= 'yes' ].
```

It must be understood that to solve problems in a functional way many times the imperative mentality is an obstacle. We have to think about the data we have and what data we want to obtain as a result. It is what will lead us to the solution.

Erlang is a free-form language. You can insert as many spaces and line breaks between symbols as you want. This ***area*** function is completely equivalent to the previous one at the execution level:

```
area(
  Base,
  Height
) ->
  Base * Height
.
```

Throughout this chapter, we will review the basis of Erlang's language. We will see what is necessary to write general purpose basic programs and understand this brief introduction in a more detailed and clear way.

1.Data Types

In Erlang, several data types are handled. Making a quick distinction we can say that they are distinguished between: simple and complex; other organizations could lead us to think of data as: scalars and sets or atoms and compounds. However, the way to organize them is not relevant in order to know them, identify them and use them correctly. We will use the denomination simple and complex (or compound), being able to refer to any of the other forms of categorization if the explanation is clearer.

As simple data we will see in this section **atoms** and **numbers**. As complex data we will see the **lists** and **tuples**. We will also see the **binary lists**, a quite powerful data type from Erlang and the **records**, a data type derived from the tuples. To finish we will review a data type that was introduced in version 17 of Erlang, the **maps**.

1.1.Atoms

Atoms are character identifiers that are used as keywords and help to semantize the code.

An atom is a word that begins with a lowercase letter and is followed by uppercase or lowercase letters, numbers and / or underscores. You can also use uppercase letters at the beginning, spaces and whatever you want, as long as you enclose the expression in single quotes. Some examples:

```
> is_atom(square).
true
> is_atom(a4).
true
> is_atom(signup_client).
true
> is_atom(removeClient).
true
> is_atom(alert_112).
true
> is_atom(false).
true
> is_atom('HELLO').
true
> is_atom('     eh??? ').
true
```

Atoms have the sole purpose of helping the programmer identify specific structures, algorithms and code.

There are atoms that are used very frequently such as: true, false and undefined.

Important

Each time an atom is used, the representation is added to an internal table. This table has a finite but configurable size. By default, the maximum value of different atoms that can be used in an Erlang virtual machine is 1,048,576. This value can be extended with the ***+t*** parameter.

One of the ways to ensure not to exceed the limit of available atoms is to use them explicitly in the code. In this way you have a control of how many atoms are being used and at the moment of loading the code we can be notified when the number has been exceeded.

If we need to convert other data types to atoms and make sure we do not exceed the limit, we can use the set of functions `*_to_existing_atom/1-2`.

Atoms along with integer and real numbers as well as text strings make up what is known in other languages as ***literals***. They are data that have a meaning in themselves, and can be assigned directly to a variable.

Note

As literals we can specify numbers, but also values of representations of the character table. As in other languages, Erlang allows to give the value of a specific character through the use of the syntax: $A, $1, $!. This will return the numerical value to the symbol indicated after the dollar symbol in the character table.

1.2.Integer and Real Numbers

In Erlang, there are two types of numbers, as shown in this code example in the console:

```
> is_float(5).
false
> is_float(5.0).
true
> is_integer(5.0).
false
> is_integer(5).
true
```

Another thing that surprises about Erlang is its numerical accuracy. If we multiply very high numbers we will see how the result is still shown in integer notation, without using the scientific notation that other languages show when an operation exceeds the limit of calculation of the integers (or erroneous values because of the ***overflow effect***):

```
> 102410241024 * 102410241024 * 1234567890.
12947972063153419287126752624640
```

This feature makes Erlang a very accurate and suitable platform for bank interest calculations, telephone pricing, stock indexes, statistical values, position of three-dimensional points, etc.

Note

The numbers can also be indicated by prefixing the base in which we want to express them and using the hash key (#) as a separator. For example, if we want to express the numbers on octal basis, we will do it by putting the base before the number we want to represent 8#124. Analogously 2#1011 represents a binary number and 16#f42a represents a hexadecimal number.

1.3.Variables

Variables, as in mathematics, are symbols to which one (and only one) value is linked throughout the execution of the specific algorithm. This means that each variable can only contain one value during its lifetime.

The format of the variables starts with a capital letter, followed by as many letters, numbers and underlines as needed or desired. A variable can have this form:

```
> Pi = 3.1415.
3.1415
> Telephone = "666555444".
"666555444"
> Debug = true.
true
```

Arithmetic expressions can be performed on the variables, if they contain numbers, list operations or they can be used as a parameter in function calls. An example of variables containing numbers:

```
> Base = 2.
2
> Height = 5.2.
5.2
> Base * Height.
10.4
```

If at any given moment, we want ***Base*** to have the value 3 instead of the value 2 initially assigned, we would see the following:

```
> Base = 2.
2
> Base = 3.
** exception error: no match of right hand side value 3
```

What is happening is that ***Base*** is already binded to the value 2 and that the match with the value 2 is correct, whereas if we try to fit it with the value 3, it results in an exception. An error because Base was binded to value 2 and it's not possible to match with 3.

Note

For our tests, at the console level and to avoid having to leave and enter each time we want Erlang ***to forget*** the value with which a variable was linked, we can use:

```
> f(Base).
ok
> Base = 3.
3
```

To eliminate all the variables stored in the console, you can use: `f().`

The advantage of the unique assignment is the ease of analyzing code even though it is often not considered that way. If a variable during the execution of a function can only contain a certain value, the behavior of that function is very easily verifiable.

1.4.Lists

The lists in Erlang are vectors of heterogeneous information, that is, they can contain information of different types, whether numbers, atoms, tuples or other lists.

The lists are one of the powers of Erlang and other functional languages. As in Lisp, Erlang manages the lists as a high-level language, in declarative mode, allowing things such as list comprehensions or the aggregation and elimination of specific elements as if they were sets.

1.4.1.What can we do with a list?

A list of elements can be defined directly as presented below:

```
> [ 1, 2, 3, 4, 5 ].
[1,2,3,4,5]
> [ 1, "Hello", 5.0, hello ].
[1,"Hello",5.0,hello]
```

To these lists you can add or subtract elements with the special operators ***++*** and --. As it is given in the following examples:

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

Another common use of the lists is the way in which you can take items from the list header leaving the rest in another sublist. This is done with this simple syntax:

```
> [H|T] = [1,2,3,4].
[1,2,3,4]
> H.
1
> T.
[2,3,4]
> [H1,H2|T2] = [1,2,3,4].
[1,2,3,4]
> H1.
1
> H2.
2
> T2.
[3,4]
> [1, 2, 3, 4] = [1|[2|[3|[4|[]]]]].
[1,2,3,4]
```

In this simple way, the implementation of the well-known ***push*** and ***pop*** algorithms for insertion and extraction in stacks are as trivial as:

```
> List = [].
[]
> List2 = [1|List].
[1]
> List3 = [2|List2].
[2,1]
> [PopElement|List2] = List3.
[2,1]
> PopElement.
2
> List2.
[1]
```

However, not being able to maintain a single variable for the stack makes it difficult to use. We will analyze this matter later with the treatment of processes and functions.

1.4.2.Strings

Strings are a specific type of list. It is an homogeneous list of elements that can be represented as characters. Erlang detects that if a list in its entirety meets this premise, it is a string of characters.

Therefore, the representation of the word ***Hello*** in the form of a list can be done as a list of integers representing each of the letters or as the text enclosed in double quotes ("). A demonstration:

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

As you can see, the assignment does not give any error since both values, left and right, are the same for Erlang.

Important

This way of dealing with strings is very similar to that used in C language, where the ***char*** data type is an 8-bit data in which a value from 0 to 255 can be stored and the printing functions will take as representations of the table of characters in use by the system. In Erlang, the only difference is that each data is not 8 bits but is an integer which leads to greater memory consumption but better support of new tables such as UTF-16 or extensions of UTF-8.

As with the rest of the lists, the strings of characters also support the aggregation of elements, so that the concatenation could be done in the following way:

```
> "Hello, " ++ "world!".
"Hello, world!"
```

One of the advantages of Erlang's own allocation is that if it finds a variable that has not been binded to any value, it automatically takes the necessary value for the ***equation*** to be true. Erlang always tries to make the elements on both sides of the assignment sign equal. An example:

```
> "Hello, " ++ A = "Hello, world!".
"Hello, world!"
> A.
"world!"
```

This notation has its limitations, specifically the unassigned variable must be at the end of the expression, since otherwise the code to perform the match would be much more complex.

1.4.3.Binary lists

Character strings are formed by sets of integers, that is, twice as much memory is consumed for a string of characters stored in a list in Erlang than in any other language. Binary lists allow you to store character strings with byte size and to perform specific jobs with byte sequences or even at bit level.

The syntax of this type of lists is as follows:

```
> <<"Hello">>.
<<"Hello">>
> <<72,101,$l,$l,$o>>.
<<"Hello">>
```

The binary list does not have the same functionality as the previously viewed lists. You cannot add elements or use the annexing and deleting elements syntax as it had been seen before. But it can be done in a more powerful way.

For example, the way in which we took the head of the list in one variable and left the rest in another variable, can be simulated in the following way:

```
> <<H:1/binary,T/binary>> = <<"Hello">>.
<<"Hello">>
> H.
<<"H">>
> T.
<<"ello">>
```

The concatenation in the case of binary lists is not done as in normal lists using the ++ operator. In this case it must be done in the following way:

```
> A = <<"Hello ">>.
<<"Hello ">>
> B = <<"world!">>.
<<"world!">>
> C = <<A/binary, B/binary>>.
<<"Hello world!">>
```

To obtain the size of the binary list we use the function `byte_size/1`. In the previous case for each of the variables used:

```
> byte_size(A).
6
> byte_size(B).
6
> byte_size(C).
12
```

This syntax is a little more elaborate than that of the lists, but it is because we go into the true power that binary lists have: the handling of bits.

1.4.4.Working with Bits

In the previous section we saw the basic syntax to simulate the behavior of the chain when taking the head of a stack. This syntax is based on the following format: ***Var:Size/Type***; being optional Size and Type, although not for all the cases. We will see this further.

The size is linked to the type, since a unit of measurement is nothing without its quantizer. In this case, the quantizer (or type) that we have chosen is ***binary***. This type indicates that the variable will be binary list type, so the size will be referring to how many elements of the list will contain the variable.

In case the size is not indicated, it is assumed that it is as much as the support type and/or to fit the value to which it must be matched (if possible) so in the example of the previous sections to get the head and tail values, the variable T remains with the rest from the binary list.

The types also have a complex way of forming themselves, since several elements can be indicated to complete the definition of the same. These elements are, in order of specification: ***Endian-Sign-Type-Unit***, we will see the possible values for each of them:

Endian

Is the way in which the bits are read in the machine, whether it is in Intel or Motorola format, that is, ***little*** or ***big*** respectively. In addition to these two, it is possible to choose ***native***, which will use the native format of the machine on which the code is running. The default value is set to ***big***.

```
> <<1215261793:32/big>>.
<<"Hola">>
> <<1215261793:32/little>>.
<<"aloH">>
> <<1215261793:32/native>>.
<<"aloH">>
```

In this example you can see that the machine where I'm running these examples is of the little type or Intel sorting.

Sign

It is indicated if the number will be stored in signed or unsigned format. The sign usually affects only to integer numbers for which the first bit could be used as the sign taking the value of 0 for positive numbers and 1 for negative numbers:

```
> <<S1:16/signed>> = <<32767:16>>, S1.
32767
> <<S2:16/signed>> = <<32768:16>>, S2.
-32768
> <<U1:16/unsigned>> = <<32768:16>>, U1.
32768
```

This example shows us how a 16-bit integer is interpreted in two different ways depending if we use signed or unsigned parameters.

Type

Is the type with which the data is stored in memory. Depending on the type, the size is relevant to indicate precision or number of bits, for example. The available types are: ***integer***, ***float*** and ***binary***.

By default type is integer as you can see in previous examples. If we are using integer type we can omit it to save space.

Unit

This is the value of the unit, by which it will multiply the size. In case of integers and floating point the default value is 1, and in case of binary it is 8. Therefore: ***Size x Unit = Number of bits***; for example, if the unit is 8 and the size is 2, the bits occupied by the element are 16 bits. It's useful if you want to retrieve elements from a binary using other bit sizes.

The syntax is as follows:

```
> <<A:1/big-signed-integer-unit:16>> = <<65535:16>>, A.
-1
```

This simplifies the way to indicate sizes in some specific uses like when we want to indicate nibbles (4 bits), bytes (8 bits), words (16, 32 or 64 bits) instead of bits.

If we wanted to store three data of red, green and blue color in 16 bits, taking for each of them 5, 5 and 6 bits respectively, we would have that the partition of the bits could be done in a somewhat difficult way. With this handling of bits, to compose the chain of 16 bits (2 bytes) corresponding, for example, to the values 20, 0 and 6, would be like this:

```
> <<20:5, 0:5, 60:6>>.
<<" <">>
```

Note

To obtain the size of the binary list in bits we can use the function `bit_size/1` that will return the size of the binary list:

```
> bit_size(<<"Hello world!">>).
96
```

1.5.Tuples

Tuples are organizational data types in Erlang. Tuple lists can be created to form homogeneous datasets of heterogeneous individual elements.

Tuples, unlike lists, cannot increase or decrease their size except for the complete redefinition of their structure. They are used to group data with a specific purpose. For example, imagine that we have a directory with a few files. We want to store this information in order to treat it and we know it will be: route, name, size and date of creation.

This information could be stored in the form of a tuple in the following way:

```
{ "/home/me", "text.txt", 120, {{2011, 11, 20}, {0, 0, 0}} }.
```

The keys indicate the start and end of the definition of the tuple, and the elements separated by commas make up its content.

Note

In the example you can see that both date and time have been entered in a somewhat peculiar way. In Erlang, the functions of the modules of its standard library work with this format, and if it is used, it is easier to process and work with dates. For example, if we were to execute:

```
> {date(), time()}.
{{2011,12,6},{22,5,17}}
```

This type of data is also used to emulate the ***associative arrays*** (or hash). These arrays store information so that it is possible to rescue it by means of the text or specific identifier that was used to store it. It is used in those cases in which it is easier to access the element by a known identifier than by an index that could be unknown.

1.5.1.Dynamic modification of tuples

There are times when we need to add a value to a tuple, remove a value or take the value that is in a position without having to make a concordance. For this we can choose to use this set of functions that I list below:

erlang:setelement/3

Change the element of a tuple without modifying the rest of the elements:

```
> erlang:setelement(2, {a, b, c}, 'B').
{a,'B',c}
```

erlang:append_element/2

Add an element to the end of the tuple:

```
> erlang:append_element({a, b, c}, d).
{a,b,c,d}
```

erlang:element/2

Gets an element of the tuple given its index:

```
> element(1,{a,b,c}).
a
```

erlang:delete_element/2

Remove an element from a tuple:

```
> erlang:delete_element(2,{a,b,c}).
{a,c}
```

1.5.2.Property list

A property list is a list of key-value pair tuples. It is managed through the ***proplists*** module. Property lists are widely used to store configurations or in general any variable information that is required to be stored.

Let's suppose we have the following data sample:

```
> A = [{path, "/"}, {debug, true}, {days, 7}].
```

Now suppose that from this list, which has been loaded from any file or by any other method, we want to check whether or not to debug the system, that is, show log messages if the ***debug*** property equals ***true***:

```
> proplists:get_bool(debug, A).
true
```

As it is very possible that the keys that exist at a certain moment in the list are not known, the functions `is_defined/2`, or `get_keys/1` help us to check if a key exists in the list or obtain a list of keys from the list respectively.

An example of possible use as a hash table would be:

```
> Months = [
    {january, 31}, {february, 28}, {march, 31},
    {april, 30}, {may, 31}, {june, 30},
    {july, 31}, {august, 31}, {september, 30},
    {october, 31}, {november, 30}, {december, 31}
].
> proplists:get_value(january, Months).
31
> proplists:get_value(june, Months).
30
```

The use of property lists in this way gives us access to the data that we know exist within a collection (or list) and allows us to extract only those we want to obtain.

Note

The ***proplists*** module contains many more useful functions to handle this type of data collection in an easy way. It is not a bad idea to go over it and see all the possible options that we can get out of this module in our programs.

1.6.Records

Records are a specific type of tuple that facilitates the access to the individual data within the same one by means of a name and a syntax of access much more comfortable for the programmer. Internally for Erlang, records do not really exist. At the preprocessor level they are exchanged for tuples. This means that the records themselves are a simplification at the level of use of the tuples.

Since records are used at the preprocessor level, in the console we can only define records using a specific console command. In addition, we can load the existing records into a file and use them from the console itself to define data or to use the own data management commands with records.

The definition of records from the console is done in the following way:

```
> rd(agenda, {name, surname, phone}).
```

To declare a record from a file, the format is as follows:

```
-record(agenda, {name, surname, phone}).
```

The declaration can be complicated a bit more if we add default values in the previous definition:

```
-record(agenda, {
    name,
    surname = "",
    phone
}).
```

Or from the console in this way:

```
> rd(agenda, {name, surname = "", phone}).
```

The default values of each of the elements of the record is always ***undefined***. In the previous case, we can check it in the following way:

```
> #agenda{}.
#agenda{name = undefined,surname = [],phone = undefined}
```

At the time of declaring or using a record we may want to change most fields for a specific value different from ***undefined***:

```
> #agenda{_=null}.
#agenda{name = null,surname = null,phone = null}
> #agenda{name = "Manuel", _ = no}.
#agenda{name = "Manuel",surname = no,phone = no}
```

Note

Erlang code files usually have the extension ***erl***, however, when it comes to codes of ***header*** type, these files maintain an extension halfway between the C header (which have the extension .h) and the code typical of Erlang. Its extension is: ***hrl***. Normally definitions and records will be introduced in these files.

Let's see with a little test that if we create a tuple A, Erlang recognizes it as a tuple of four elements. If we later load the file `agenda.hrl` whose content is the definition of the agenda record, the treatment of the tuple is automatically modified and we can now use the notation for records of the following examples:

```
> A = {agenda, "Manuel", "Rubio", 666666666}.
{agenda,"Manuel","Rubio",666666666}
> rr("agenda.hrl").
[agenda]
> A.
#agenda{name = "Manuel",surname = "Rubio",
        phone = 666666666}
```

Erlang recognizes the name of the record as the first data of the tuple and since it has the same number of elements, if we do not consider the identifier, it automatically considers it as a record. You can also continue using the functions and typical elements of the tuple since for all intents and purposes it is still so.

Note

To obtain the position within the tuple of a field, simply write it in the following way:

```
#agenda.nombre
```

This will return to us the relative position defined as a name in relation to the tuple that contains the agenda type record.

To process the data of a record, we can perform any of the following actions:

```
> A#agenda.name.
"Manuel"
> A#agenda.phone.
666666666
> A#agenda{phone = 911232323}.
#agenda{name = "Manuel",surname = "Rubio",
        phone = 911232323}
> #agenda{name = "Juan Antonio",surname = "Rubio"}.
#agenda{name = "Juan Antonio",surname = "Rubio",
        phone = undefined}
```

Always remember that the assignment remains unique.

To access to the content of a data in a field of the record, we will access indicating that it is a record (***data#record*** (***A#agenda*** in the example) and then add a point and the name of the field we want to access to.

To modify the data of an existing record instead of the point we will use the keys. Within the keys we will establish as many ***key=value*** equalities as we need (separated by commas), as seen in the previous example.

To obtain information about the records at a given moment, we can use the function `record_info/2`. This function has two parameters, the first is an atom that can contain **fields**, if we want to return a list of atoms with the name of each field; or **size**, to return the number of fields that the tuple has where the record is stored (including the identifier, in our examples ***agenda***).

Important

As mentioned above, records are entities that work at the language level but Erlang does not consider them at runtime. This means that the preprocessor works to convert each instruction concerning records to be relative to tuples and therefore the function `record_info/2` cannot be used with variables. Something like the following:

```
> A = agenda, record_info(fields, A).
```

It will return us ***illegal record info***.

Since the records are internally tuples, each field can contain any other data type, not only atoms, strings or numbers, but also other records, tuples or lists. Therefore, this structure proposes an interesting organizational system to be able to directly access the data that we need at a given moment facilitating the work of the programmer enormously.

1.7.Maps

Maps are data structures. Each item is stored under an index or key and contains a value. The index can be of any type as well as its content. We can define a map in the following way:

```
> M = #{ name => "Manuel" }.
#{name => "Manuel"}
```

We can add and change map data as we would do with a record, in the following way:

```
> M2 = M#{ surname => "Rubio" }.
#{surname => "Rubio",name => "Manuel"}
```

We can also indicate a change on an existing index. The indicator change and its function are useful for modifying data initially defined in the map. If the data does not exist, the system will fail:

```
> M3 = M2#{ name := "Juan" }.
#{surname => "Rubio",name => "Juan"}
> M4 = M3#{ phone := 666555444 }.
** exception error: bad argument
```

To extract a value, we must perform values matching. In the previous example, to extract the name:

```
> #{ name := Name } = M3.
#{surname => "Rubio",name => "Juan"}
> Name.
"Juan"
```

In this case, the symbol ***:=*** is used to indicate the required existence of the key to obtain its value.

Note

EEP-0043[1] indicates the possibility of extracting a simple value using another syntax easier and without requiring the use of a variable for concordance. In version 22.0 it has not yet been implemented although it appears in the standard.

The ***maps*** module contains the functions that allow you to delete a key, search using a function instead of concordance, detect if it is a map or not, obtain the number of keys and some other options.

[1] http://www.erlang.org/eeps/eep-0043.html

Some examples:

```
> maps:remove(name, M3).
#{surname => "Rubio"}
> maps:get(name, M3).
"Juan"
> is_map(M3).
true
> map_size(M3).
2
> maps:keys(M3)
[surname,name]
```

1.8. Data conversion

It is important to know how to convert data types. Mainly to be able to change a character string data type to a binary string, or towards an atom or even like number.

Many times, it is necessary to convert to another format a data entry collected from a file, a connection from/to another machine or from a standard input.

The most useful conversions are those we can make between string and numeric type formats. For example, if we have a list of characters containing a decimal number and we would like to convert it to a number, we would have to execute the following:

```
> list_to_integer(101).
101
```

We can also indicate that the text represents a number in another different base such as 2, 8, 16 or 36:

```
> list_to_integer("101", 2).
5
> list_to_integer("101", 8).
65
> list_to_integer("101", 16).
257
> list_to_integer("101", 36).
1297
> list_to_integer("101", 64).
** exception error: bad argument
     in function  list_to_integer/2
        called as list_to_integer("101",64)
```

The maximum base number for the representation of a number is 36. If we try to indicate a higher number, as we can see above, an error will occur. This is due to the range of symbols chosen when specifying the base: 10 numeric digits and 26 letters.

Note

From the R16 version of Erlang, functions were added to convert from binary to integer and vice versa. Previously you had to convert first to list to be able to convert after or to whole or to binary.

The conversion functions are the following:

atom_to_list/1, atom_to_binary/2

These functions are responsible for converting an atom data to a character or binary list. The second parameter in the binary conversion indicates the set of characters to be used. The most used are ***utf8*** and ***latin1***, but there are many more.

binary_to_atom/2, binary_to_float/1, binary_to_integer/1-2, binary_to_list/1-3

From binary to atom, floating point number, integer and character list. For the atom, the character set must be indicated. For an integer we can optionally indicate the base and for list of characters we can specify the start and end (starting from 1) to take only the elements of the binary to be included in the list of characters.

```
> binary_to_atom(<<"1234">>, utf8).
'1234'
> binary_to_float(<<"1234">>).
** exception error: bad argument
     in function  binary_to_float/1
        called as binary_to_float(<<"1234">>)
> binary_to_float(<<"1234.0">>).
1234.0
> binary_to_integer(<<"1234">>).
1234
> binary_to_integer(<<"1234">>, 16).
4660
> binary_to_integer(<<"ff">>, 16).
255
```

list_to_atom/1, list_to_binary/1, list_to_float/1, list_to_integer/1-2, list_to_pid/1, list_to_ref/1

Convert the content of the list to the data type specified in the function.

integer_to_binary/1-2, integer_to_list/1-2, float_to_binary/1-2, float_to_list/1-2

We saw examples of these functions above. The number stored in the variable is represented and stored in a binary variable or character list. The second parameter allows you to specify the base in which the number will be converted.

list_to_tuple/1, tuple_to_list/1

Convert a list into a tuple and a tuple in a list respectively:

```
> erlang:list_to_tuple([1,2,3,4]).
{1,2,3,4}
> erlang:tuple_to_list({1,2,3,4}).
[1,2,3,4]
```

2.Printing on screen

Many times, we will need to display data on the screen. At the moment, all the information we see is because the console shows it to us, as a result of the output of the code we are writing. However, there are moments, in which it will be necessary to make a concrete exit of a data with more complete information.

To do this, we have the ***io*** module, from which we will only use the `io:format/2` function for now. This function allows us to print on the screen the information we want to show, based on a specific format that is passed as the first parameter.

Note

For those who have programmed with languages such as C, Java, PHP, ... this function is equivalent and very similar to ***printf***, that is, the function is based on a string with a specific format (adding parameters) that will be replaced by the values indicated in the following parameters.

For example, if you want to display a string on the screen, we can write the following:

```
> io:format("Hello world!").
Hello world!ok
```

This goes like this because the return of the function is ***ok***, so the string is printed and then the return of the function (the function return is always printed in the console). To make a car return, we must insert a special character. Unlike other languages where special characters are used, Erlang does not use the backslash, but uses the tilde (~), and after this symbol, the characters are interpreted in a special way. We have:

~

Print the symbol of the tilde.

c

Represents a character that will be replaced by the corresponding value passed in the list as the second parameter. Before the letter ***c*** you can add a pair of numbers separated by a full stop. The first number indicates the size of the field and the justification to left or right according to the positive or negative sign of the number. The second number indicates how many times the character will be repeated. For example:

```
> io:format("[~c,~5c,~5.3c,~-5.3c]~n", [$a,$b,$c,$d]).
[a,bbbbb,  ccc,ddd  ]
ok
```

e / f / g

They are responsible for presenting floating point numbers. The format of ***e*** is scientific (X.Ye+Z) while ***f*** is presented in a fixed-point format. The format ***g*** is a mixture since it presents the scientific format (***e***) if the number is out of the range [0.1.10000.0], and otherwise presents the format of a fixed point (***f***). The numbers that can be prefixed to each letter indicate, the size you want to represent and justification (as seen before). After the point the precision. Some examples:

```
> io:format("[~7.2e,~7.2f,~7.4g]", [10.1,10.1,10.1]).
[ 1.0e+1,  10.10,  10.10]ok
> Args = [10000.67, 10123.23, 1220.32],
> io:format("~11.7e | ~11.3f | ~11.7g ", Args).
1.000067e+4 |   10123.230 |    1220.320 ok
```

s

Print a character string. Similar to ***c***, but the meaning of the second number in this case is the number of characters in the list that will be displayed. Let's see some examples:

```
> Hello = "Hello world!",
> io:format("[~s,~-7s,~-7.5s]", [Hello, Hello, Hello]).
[Hello world!,Hello w,Hello  ]ok
```

w / W

Print any data with its standard syntax. It is used above all to be able to print tuples, but it also prints lists, numbers, atoms, etc. The only caveat is that a character string will be considered as a list. The prefix numbers are used in the same way as in ***s***. An example:

```
> Data = [{hello,world},10,"hello",world],
> io:format("[~w,~w,~w,~w]~n", Data).
[{hello,world},10,[104,101,108,108,111],world]
ok
```

The ***W*** version is similar to the previous one although it takes two parameters from the parameter list. The first is the data that will be printed, the second is the depth. If we print a list with many elements, we can show only a certain number of them. From that number add ellipses. An example:

```
> io:format("[~W]", [[1,2,3,4,5],3]).
[[1,2|...]]ok
```

p / P

It is the same as ***w***, but it tries to detect if a list is a character string to print it like that. If the impression is too large, it will be divided in several lines. The uppercase version is also the same as its ***W*** namesake, accepting an extra parameter for depth.

b / B / x / X / + / #

They print numbers according to the indicated base. The previous numbers to each letter (or symbol) indicate, the first the magnitude and the justification of the presentation and the second the base in which the number will be expressed. They all print numbers, but there are differences between them.

B prints the numeric parameter. If the base is greater than 10 the letters will be printed in capital letters. With ***b*** the letters are printed in lowercase letters.

With ***X*** and ***x*** they are equal to the previous ones adding the possibility of using a prefix taken from the next parameter in the list of parameters, consecutive to the value to be represented.

The number sign (#) always give preference to the base in Erlang format: 10#20 (decimal), 8#65 (octal), 16#1A (hexadecimal). The plus sign (+) is the same but writing the letters in lowercase. An example:

```
> io:format("[~.2b,~.16x,~.16#]", [21,21,"0x",21]).
[10101,0x15,16#15]ok
```

i

It ignores the parameter that you use. It is useful if the format of the parameters that is passed is always the same and if you want to ignore a specific one in a specific format.

n

Carriage return, makes a line break, so that you can separate by different lines what you want to print on the screen.

Note

There is also the `io_lib` module that also has the function `format/2`. The only difference it presents is that instead of presenting the resulting string on the screen, it returns it as a string of characters.