The Interworking's of Erlang

Yuwei

Abstract

Erlang is a general purpose functional concurrent programing language. Erlang is used mainly for telecom services. This research primarily focuses on exploring what defines and separates erlang from other languages as well as it's applications on multithread plain text parsing.

Introduction

Erlang first developed in 1986 by Joe Armstrong for the company Ericsson. The language was first designed to make telecom communication easier and more efficient. Since availability always needed to be high, there needed to be a language and system built on concurrency and speed. Erlang provided an easy general-purpose language that specialized in transferring data.

Since most messages sent through telecom does not need to be modified, for this reason erlang was made a functional language. This makes everything much simpler and decreases errors. This research focuses on what makes erlang different and show some of the pros and cons of some of the design choices of the language.

Syntax

Erlang syntax is very different from most languages. All lines must end with a period and a white space character. For arithmetic operators, the primitives that erlang supports directly is integers and floats. All operators must come between two

numbers. However, integer division is not directly supported by the division symbol. The function div must take the place the division symbol. This is also true for remainders. Remainders is implemented into the language through the function rem. A cool mathematic operator that should be included into other languages is the use of the # sign for the change of base. For example, 2 # 1000. will give you 8. This operator converted the binary sequence 1000 into base 10 which is 8. This also works with Hex. 16 # AE. Will give you 174. This allows for quick conversions between bases.

Variables

Functional languages do not allow variables. However, erlang allows non mutable variables. This means that all variables created in erlang can only be bounded once. All variables must start with a capital letter. For example, One = 1. Will bound the value 1 to the variable One. We then can not say One = 2. Erlang will complain about this line. Another interesting thing is that we can assign multiple variables at the same time. Un = Uno = One = 1. will assign the value 1 to all three variables. You can also assign variables to other variables. For example, Un = Uno. would compile because Un and Uno has the same value. Un = Uno + 1. would not compile. The assignment operator uses pattern matching to assign the left side to the right. There is a keyword variable called _. This variable is for testing purposes and will not store any values after it is used in an operation.

Atoms

The reason all variables must start with a capital letter is because of atoms. Atoms are literals. Atoms are not mutable and starts with lowercase letters. If one wants to use special characters or uppercase letters in their atoms, they must surround the string with single quotes. For example, atom. will return atom. ‘Atom12@gmail.com’ will return Atom12@gmail. Atoms are made this way so that you will never have an undefined value. This makes atom comparisons really easy as atoms can never change, and you don’t need a specific function to compare two atoms. Although Atoms are very useful, they are not garbage collected. This means that all declared atoms will stay in memory until the end of the program.

Tuples

Tuples are used in many languages to organize data for the user. For example,

coordinate points. In erlang tuples are surrounded by curly braces. The coordinate tuple {3,4} shows that 3 represents the x coordinate and 4 represents the y coordinate. However, in erlang you can use the underscore as a place holder or a wild card. If you wanted coordinates on the x = 3 coordinate line, one can assign {3, _} equals to coordinate tuple. It will replace the underscore with the value of the right-side tuple. The only restriction is that the number of values I the tuples must match.

Lists and Strings

Lists and Strings cause a large problem in erlang. The reason being that they have the same syntax. Both strings and lists are surrounded by brackets. This can cause many problems when interpreting the code. Lists in erlang supports multiple types in the same list. This can cause many problems but is very powerful. For example, you can have a mix a list of numbers with atoms and floats and doubles. This increases the amount of applications a list can have. However, this can lead to many problems. Take this list for example [97,98,99].. This will return abc. The reason being the ascii numbers of abc is 97,98, and 99. Erlang will only recognize a list as a list if at least one element in the list cannot be converted to a character.

Some cool features of lists in erlang is the ++ and -- operator. Using this operator on two lists will append lists and finds the difference between two lists. A example would be the [1,2,3] ++ [5,6,7]. This expression would return [1,2,3,4,5,6,7]. You can also take the difference of two lists. [1,2,3] -- [1,2] would return [3].

Some other things one can do with lists in erlang is the use of hd and tl. This is the same as car and cdr in lisp. It takes the head and the tail of a list. One can also construct a new list using the pipe symbol. [Head | Tail] = [1,2,3,4,5]. Would assign the variable Head the value 1 and Tail the list [2,3,4,5]. You can extend this to construct any variation of a list. [1, | [2, 3 | []]] constructions a list by appending the separate lists of [1] [2,3] and a empty list. A empty list indicates that there can be more elements in the list.

List Comprehensions

There are many things one can do with lists in functional languages. Erlang has many of the same list comprehensions that other function languages has. For example the expression

[X || X <- [1,2,3,4,5,6,7,8,9,10],

X rem 2 =:= 0]. This gets all even elements on said list. In a longer sense the statement says assign X the list [1,2,3,4,5,6,7,8,9,10] then get each element in each individually. If the X modulus 2 is 0 then return that number. The list of atoms Weather = [{toronto, rain},{montreal, storms}, {london, fog}, {paris, sun}, {boston, fog}, {vancouver, snow}]. can be used in very useful ways. With the statement [X || {X, rain} <- Weather]. One can get all cities that are raining. Such a simple statement are very powerful in functional languages.

Modules

Modules are like classes in java. They hold several functions with a number of attributes. Modules allow developers to organize and write custom functions into one space. This makes importing new functions much easier. Here is an example a module.

```
-module(useless).
-export([add/2, hello/0, greet_and_add_two/1]).

add(A,B) ->
    A + B.

%% Shows greetings.
%% io:format/1 is the standard function used to output text.
hello() ->
    io:format("Hello, world!~n").

greet_and_add_two(X) ->
    hello(),
    add(X,2).
```

The module is called useless and has three functions. Each function export statement must have the name of the function and number of parameters for said function.

Control Statements

Controls statements in erlang are similar to languages in that time period. All if statements must be followed by a then statement. There are also optional statements of else if and else statements. These keywords are the same as in Java. However, all if statements must end with the end keyword.

```
function greet(Gender,Name)
    if Gender == male then
        print("Hello, Mr. %s!", Name)
    else if Gender == female then
        print("Hello, Mrs. %s!", Name)
    else
        print("Hello, %s!", Name)
end
```

Another control statement in Erlang is guards. Here is an example of guards.

```
old_enough(0) -> false;
old_enough(1) -> false;
old_enough(2) -> false;
...
old_enough(14) -> false;
old_enough(15) -> false;
old_enough(_) -> true.
```

It is a series of Boolean conditionals. In this case if you are between the ages of 0 and 15 you are not old enough. As one can see, the underscore symbol is used again for a wild card. It is very important that it is at the end of the list of statements. This is because it the wild card can be any value. A better

way of doing this is in the example below.

```
old_enough(X) when X >= 16 -> true;
old_enough(_) -> false.
```

By using the when keyword, we can get rid of the redundancy of all the guard statements.

Switch case is another control structure that erlang implements. In the example below we are trying to see if we should go to the beach.

```
beach(Temperature) ->
    case Temperature of
        {celsius, N} when N >= 20, N =< 45 ->
            'favorable';
        {kelvin, N} when N >= 293, N =< 318 ->
            'scientifically favorable';
        {fahrenheit, N} when N >= 68, N =< 113 ->
            'favorable in the US';
        _ ->
            'avoid beach'
    end.
```

The reason the erlang implementation of switch cases is more powerful then average switch case is that we can implement functions and complex list comprehension statements within the case statements. In this case we took case of Celsius, kelvin and Fahrenheit all in one switch case statement. Since we have the ability to use complex pattern matching statements in switch statements, we can implement rather complex situations in a fairly simple way.

Typing

Erlang enforces dynamically type binding. This means that on compile time the compiler does not check that all your variables are initiated, or all your comparisons are correctly written. This does pose a series of problems for most languages that are dynamically typed. However, erlang is the exception to this. Erlang was designed in a way that enforces that no variable can be declared without initializing it. This combined with the independent based functions in erlang. The combination of these and several other design choices ensures that an erlang program will not crash even if sections of a erlang program runs into errors.

Erlang is also a strongly typed language. This means that the language enforces predefined types for all variables in the language. This means does limit the power that the developer has, but it does cut back on errors and variability immensely. Since availability is essential to the language, a large amount of variability is a bad thing.