Introduction to Erlang

1st Semester, A.Y. 2020-2021

Erlang is a **functional programming language** that has a large emphasis on ***concurrruncey*** and ***high reliability***. It uses the ***actor model***, and each actor is a separate process in the virtual machine.

Installation

To install erlang on your Ubuntu, open the terminal and execute the following command:

```
$ sudo apt-get install erlang
```

To start the Erlang shell in Linux, type the following in the terminal:

```
$ erl
```

To end the Erlang shell, type

```
1> q()
```

1 Syntax

- Erlang shell can be used to execute valid Erlang Statements.
- Each expression in the Erlang shell must end with a ***period (.)***.
- Multiple expressions in a single line must be separated by ***commas (,)***.

A. Numbers

Erlang doesn't care if you enter floating point numbers or integers: both types are supported when dealing with arithmetic.

```
1> 1 + 10.
11
2> 2 * 33.3.
66.6
3> 2 - 4.99.
-2.99
4> 7 / 9.
0.7777777777777778
5> 10 div 3.                    % div operator is for integer division
3
6> 10 rem 3.                    % rem operator is for modulo
1
7> (2 + 3) rem 3 * 4.           % will be evaluated using PEMDAS
8
8> (2.4 + 3.6) * 2 + 4 / 2 - 7.
7.0
9> 2#1010.                      % integers in different bases (other than 10)
10                              % can also be used using the Base#Value syntax
10> 2#1000 + 2#111.
15
11> 8#77.
63
12> 16#F.
15
13> 16#AA + 8#77.
233
```

B. Variables

- Variable names in Erlang must begin with an ***uppercase*** letter.
- Assigning a value to a variable can only be done ***once***. Changing the value of any variable will result to an error.
- The = operator compares the values of the left-hand side and the right-hand side. If they are the same, it returns the value. Else, it will result to an error (line **19**).
- If the left-hand side term is a variable and is unbound (has no value associated to it), Erlang will automatically bind the value in the right-hand side to the variable in the left-hand side (lines **15, 16, and 17**).
- If you want to erase the value of a variable, use the **f(Variable)** function. We can also use **f()** to clear all variable names. However, *these functions are only available in the shell.*

```
14> Isa.
* 1: variable 'Isa' is unbound
15> Isa = 1.
1
16> Ichi = Uno = One = Isa = 1.
1
17> Two = Ichi + Isa.
2
18> Two = 2.
2
19> Two = Two + 1.
** exception error: no match of right hand side value 3
20> two = 2.          % will produce an error since two started with a
                      % lowercase letter
** exception error: no match of right hand side value 2
```

C. Atoms

Atoms are literals or constants with their own name for value. They always start with a lowercase letter or are enclosed with single quotes.

```
21> conan.
conan
22> ran.
ran
23> 'Edogawa'
'Edogawa'
24> haibara = 'haibara'.
haibara
25> atomsrule@erlang.
atomsrule@erlang
```

D. Boolean Algebra

Examples of boolean algebra in Erlang:

```
26> true and false.
false
27> false or true.
true
28> true xor false.
true
29> not false.
true
30> not (true and false).
true
```

The ***and*** and ***or*** operators will always evaluate arguments on both sides of the operator. Short-circuit operator counterparts are ***andalso*** and ***orelse***.

E. Comparison Operators

Operator	Description
==	equal to
/=	not equal to
=:=	exactly equal to
=/=	not exactly equal to
=<	less than or equal to
<	less than to
>=	greater than or equal to
>	greater than to

Examples of comparison operators:

```
31> 5 =:= 5.
true
32> 5 =/= 5.
false
33> 3 =:= 2.
false
34> 5 /= 5.0.
false
35> 5 == 5.0.
true
36> 1 < 2.
true
37> 1 >= 1.
true
```

Comparing expressions with different data types follows this hierarchy:

$$number < atom < reference < fun < port < pid < tuple < list < binary$$

F. Tuples

Allows you to group together N items. Tuples are written in the form:

{Element1, Element2,..., ElementN}

```
38> X = 5, Y = 3.
3
39> Point = {X,Y}.          % declare a tuple containing the value of X and Y
{5,3}
40> X.
5
41> {X,_} = Point.          % we used the anonymous _ variable which is
{5,3}                       % always unbound and acts as a wildcard for
42> {_,_} = {12,13}.        % pattern matching
{12,13}
43> {_,_} = {1,2,3}.
** exception error: no match of right hand side value {1,2}
44> {Z,W,V} = {4,5,6}.      % we assign the numbers 4,5,6 to the variables
{4,5,6}                     % Z,W,V respectively
45> Z.
4
46> {Z,V}.
{4,6}
```

We can also use tuples to represent single values that can have different types.

```
47> Temp = 23.213.
23.213
48> PreciseTemp = {celsius,Temp}.
{celsius,23.213}
```

```
49> {kelvin,T} = PreciseTemp.
** exception error: no match of right hand side value {celsius,23.213}
```

The = operator compares the format of **{kelvin, T}** and **{celsius, 23.213}** even though T is unbound. Erlang will see that the atom celsius is different compared to the atom kelvin.

The PreciseTemp is an example of a tuple for single values. A tuple which contains an atom and another element is called a ***tagged tuple***.

G. Lists

Lists are the most used data structure in Erlang. They can contain anything and follows this basic notation:

[***Element1, Element2,..., ElementN***]

```
50> [1,$A,{base,[2#11,8#34]},5.4,kat].
[1,65,{base,[3,28]},5.4,kat]
51> [$m,$e,$w].      % lists are used to represents strings since there is no
"mew"                % string data type in erlang
52> [109,101,119].
"mew"
53> [$m,$e,$w]=:=[109,101,119].
true
```

A list is composed of two parts, a head and a tail:

[***Head*** | ***Tail***]

The pipe | is the cons operator (constructor). Any list can be built with only cons and values.

```
54> List = [2,3,4].
[2,3,4]
55> NewList = [1|List].
[1,2,3,4].
56> [Head|Tail] = NewList.    % assigns the head of the list to the Head
[1,2,3,4]          % variable and the tail of the list to the tail variable
57> Head.
1
58> Tail.
[2,3,4]
59> [NewHead|NewTail] = Tail.
[2,3,4]
60> NewHead.
2
61> NewTail.
[3,4]
62> [1|[]].                   % [] denotes an empty list
[1]
63> [5|[4|[3|[2|[1]]]]].
[5,4,3,2,1]
64> [5|[4|[3|[2|[1|[]]]]]].
[5,4,3,2,1]
65> [1,2,3] ++ [4,5].         % ++ concatenates two lists
[1,2,3,4,5]
66> [1,2,3,4,5] -- [4,5].     % -- removes the second list from the first one
[1,2,3]
```

2 Modules

Modules are a bunch of functions regrouped in a single file, under a single name. All functions in Erlang must be defined in modules.

A. Module Declaration

- A module file uses the file extension **.erl**. Every module file is **required** to have the name of the module declared using the format ***-module(Name)*** where name is an atom.
- You can declare two things in a module: *attributes* and *functions*. Attributes are metadata describing the module itself and follows the form -Name(Attribute).
- Another important attribute is the **export** attribute where functions must be declared together with their corresponding arities.
- The export attribute must follow this format: **-export([Func1/Arity,...,FuncN/Arity])**
- Arity is the number of parameters passed on to the function. A module can have functions with the same name but with different number of arities.

The syntax of a function follows the form:

functionName(arguments) − > Body.

functionName has to be an atom,
arguments are the parameters, and the
Body can be one or more Erlang statements separated by commas and ended by periods.

The last logical expression of a function to be executed will have its value returned to the caller automatically.

sample.erl

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

hello() ->
   io:format("Hello World!~n"). % io:format is for printing text

add(X,Y) ->
   X + Y.

both(X) ->
   hello(),
   add(5,X).
```

B. Compiling a module

```
1> cd("/path/to/the/erl/file/").
"path to the erl file"
ok
2> c(sample).
{ok,sample}
3> sample:hello().
Hello World!
ok
4> sample:add(5,4).
9
5> sample:both(10).
Hello World!
15
```

3 Pattern Matching and Functions

Pattern matching is used in Erlang to know which function to execute. Example of pattern matching in functions:

```
greet(male,Name) ->      %for males
   io:format("Hi Mr. ~s!~n",[Name]),
   io:format("Have a good day ~n");
```

```
greet(female,Name) ->   %for females
   io:format("Hi Ms. ~s!~n",[Name]),
   io:format("Have a good day ~n");
greet(_,Name) ->        %default
   io:format("Hi ~s!~n",[Name]),
   io:format("Have a good day ~n").
```

The function above is an example of a function that has several clauses separated by a comma. It also has different cases separated by a semicolon. Erlang will match the pattern of the first parameter to either male, female, or to a default wildcard value.

Functions can also be recursive:

```
factorial(0) -> 1;
factorial(N) -> N*factorial(N-1).
```

Additional constraints can also be added using **guard clauses**.

```
factorial(N) when N>0 ->
   N*factorial(N-1);
factorial(0) -> 1.
```

4 Conditional Statements

A. If -Construct

```
if
  guard1 -> expr1,...;
  guard2 -> expr2,...;
  ...;
  guardN -> exprN,...
end.
```

B. Case-construct

```
case condition of
  Pattern1 -> expr1,...;
  Pattern2 -> expr2,...;
  ...;
  PatternN -> exprN,...
end.
```

References

[1] *Erlang Reference Manual User's Guide.*

[2] Hébert, F. Learn You Some Erlang (for great good!).