Erlang Workshop

El Monasterio de Erlang
2018-09-20

Que es Erlang?

- Lenguaje de programación funcional
- Compilado a bytecode y con tipado dinámico
- Estructuras de datos inmutables
- Soporte nativo para computación distribuida y concurrente
- Permite hacer software altamente escalable y resistente a fallos

Quienes lo usan?

Setup

1. Instalar Erlang usando tu manejador de paquetes:
    - Para **Homebrew**: `brew install erlang`
    - Para **MacPorts**: `port install erlang`
    - Para **Ubuntu** y **Debian**: `apt-get install erlang`
    - Para **Fedora**: `yum install erlang`
    - Para **FreeBSD**: `pkg install erlang`
2. Clonar el repositorio de “erlings-workshop”

    $ git clone https://github.com/lambdaclass/erlings-workshop

Atomos

Son valores literales únicos.

Permiten únicamente la operación de comparación.

Son similares a los tipos enumerados de C y Java y a los symbols de Ruby.

Comienzan en minúscula, pueden contener cualquier secuencia de caracteres alfanumericos junto a los símbolos ‘_’ y ‘@’.

```
1> hello.

2> true.

3> false.

4> a_day_of_the_week.

5> user@localhost.

6> ‘an atom with spaces’.

7> ‘user@192.168.1.1’.
```

Tuplas

Conjunto de valores heterogeneos.

Tienen longitud fija.

Similar a las tuplas de Python y Haskell

Tienen la forma:

{Elem1, Elem2, … , ElemN}

```
1> {1, 2}.

2> {nombre, "pepito"}.

3> {"user", 22, true}.

4> {{name, "usr"}, {age, 22},
{admin, true}}.
```

Funciones

Están compuestas por una cabecera (con cero o más parámetros) y un cuerpo. Tiene la forma:

nombre(Arg1, Arg2, … , ArgN) ->
Expr1,
...
ExprN.

Funciones con distinta aridad (número de parámetros) son funciones distintas.

El cuerpo de la función consta de 1 o más expresiones separadas por coma (‘,’). Siempre devuelve el valor de la ultima expresion.

```
foo() ->
  42.

foo(Arg2, Arg3) ->
  foo(Arg2, Arg3).

foo(Arg1, Arg2, Arg3) ->
  foo(Arg2, Arg3).

sumar_2(Numero) ->
  Numero + 2.

sumar(X, Y) ->
  X + Y.
```

Pattern Matching

En Erlang no existe el concepto de asignación de variables en el sentido tradicional.

En su lugar, las variables toman valores como resultado del pattern matching, esto se llama ***binding***.

En Erlang el operador “=” no es de asignación, es el operador de ***pattern matching***, el evalua el lado derecho y compara con el izquierdo. Al mismo tiempo hace ***binding*** de valores a las variables que no tengan valores.

```
Edad = 34.

Persona = {{nombre, "usuario"},
           {edad, Edad}}.

{Nombre, _Edad} = Persona.

area({triangulo, Ancho, Alto}) ->
   (Ancho * Alto) / 2;
area({rectangulo, Ancho, Alto}) ->
   Ancho * Alto.
```

Pattern Matching

Uno de los utilidades de ***pattern matching*** es al usarlo en los argumentos de una función.

Esto permite tener para una misma función distintos comportamientos sin necesidad de un ***if*** que agrega complejidad al código.

Para separar estas mismas funciones se usa el punto y coma (“;”), la última terminando con punto (“.”) como de costumbre.

```
area(Figura, Ancho, Alto) ->
  if
    Figura = triangulo ->
      (Ancho * Alto) / 2;
    Figura = rectangulo ->
      Ancho * Alto
  end.


area(triangulo, Ancho, Alto) ->
   (Ancho * Alto) / 2;
area(rectangulo, Ancho, Alto) ->
   Ancho * Alto.
```

Guardas

Expresiones booleanas que se utilizan para hacer comparaciones sobre las variables ligadas.

No cualquier función puede ser utilizada en una guarda. Solamente pueden utilizarse en guardas:

- Funciones nativas específicas
- Funciones de chequeo de tipos
- Operadores de comparación y booleanos
- Expresiones aritméticas
- Expresiones booleanas

En la guarda la coma (“,”) actúa como un ***and*** y el punto y coma (“;”) como un ***or***.

```
mayor_edad(X) when is_integer(X), X > 17 ->
  true;
mayor_edad(_) ->
  false.
```

Módulos

Forma de agrupar funciones, para hacerlas públicas (usables por otros módulos) se utiliza ***export***.

Módulos más usados:

- **erlang**
- **lists**
- **io**
- **maps**
- **math**
- **re**
- **timer**

El módulo **erlang** está semi-importando implícitamente, por lo que no es necesario escribirlo para algunas de sus funciones.

```
-module(geometria).
-export([area/1]).

area({cuadrado, Lado}) ->
   math:pow(Lado, 2).
```

Ejercicio!
hello_pattern

Escribir una función ***hello_pattern:hello/1*** que recibirá una tupla y responderá de las siguiente formas:

- {morning, Name} : morning
- {evening, Name} : {good, evening, Name}
- {night, Name} : night
- {math_class, Number, Name} : si Number es menor a cero retorna none, de lo contrario retorna {math_class, Name}

Para probar su solución utilice make.

Listas

Tipo de dato compuesto de tamaño dinámico, sus elementos son heterogéneos.

Su construcción es de la siguientes formas:

[Elem1, Elem2, … , ElemN]

[Elem1 | [Elem2 | []]]

Pueden ser descompuestos (***pattern matched***) de la siguiente forma:

[Cabeza | Cola] = [1, 2, 3]

Cabeza tendra el valor **1** y ***Cola*** tendrá el valor **[2,3]**

```
1> ListaVacia = [].

2> Vocales = [a, e, i, o, u].

3> [PrimeraVocal | Resto] =
Vocales.
% PrimerVocal == a.
% Resto == [e, i, o ,u].
```

Recursión

Erlang, dada su naturaleza funcional, carece de herramientas de iteración como: while, for, etc.

En su lugar, debemos utilizar funciones de orden superior o recursión.

```
sumatoria([]) ->
  0;
sumatoria([X | Xs]) ->
  X + sumatoria(Xs).

lists:map(fun sumar_2/1, [1, 2, 3]).
% [3, 5, 6]

lists:filter(fun es_par/1, [1, 2, 3]).
% [2]

lists:all(fun es_par/1, [1,2,3]).
% false
```

Ejercicio!
run_length_encoding

Implementar *run-length encoding* para compresion de datos.

Duplicados consecutivos de elementos son guardados como [N, E] donde N es la cantidad de veces que aparece E consecutivamente.

```
1> lists_exercises:run_length_encode([a,a,a,a,b,c,c,a,a,d,e,e,e,e]).
% [[4,a],[1,b],[2,c],[2,a],[1,d],[4,e]]
```

Proplists

Son listas de tuplas cuya primer componente actúa como clave y la segunda como valor

Se utilizan frecuentemente para pasar opciones de configuración a las funciones

Se pueden utilizar como cualquier lista pero también poseen un módulo propio: **proplists**

```
Conf = [{level, info}, {file, “error.log”}].

proplists:lookup(level, Conf).
% info

proplists:delete(level, Conf).
% [{file, “error.log”}]

Conf ++ [{formatter, default}]
% [{level, info}, {file, “error.log”},
%  {formatter, default}]
```

Mapas

Estructura de datos formada por pares clave valor

Muy similar a los diccionarios de Python y objetos de Javascript

Poseen un módulo con funciones útiles: **maps**

```
% Creación:
Map = #{} = maps:new().

% Agregar par clave valor:
Map2 = maps:put(Map, edad, 5).
Map2 = Map#{edad => 5}.
% #{edad => 5}

% Actualizar valor:
Map3 = maps:update(Map, edad, 18).
Map3 = Map#{edad := 18}.
% #{edad => 5}

% Obtener valor:
maps:get(edad, Map2).
% 5

% Buscar valor:
maps:find(edad, Map3).
#{edad := Edad} = Map3.

% Remover un par:
maps:remove(Map3, edad).
```

Ejercicio!
proplist_to_map

Escribir una función recursiva **maps_exercises:proplist_to_map/1** que recibe una proplist y construye un map con ella. Utiliza el primer componente de cada tupla como la llave y el segundo como el valor.

```
1> maps_exercises:proplist_to_map([]).
%% #{}

2> maps_exercises:proplist_to_map([{firstname, "Pedro"},
{lastname, "Sanches"}, {age, 11}]).
%% #{age => 11,firstname => "Pedro",lastname => "Sanches"}
```

Concurrencia en Erlang

Modelo de Actores

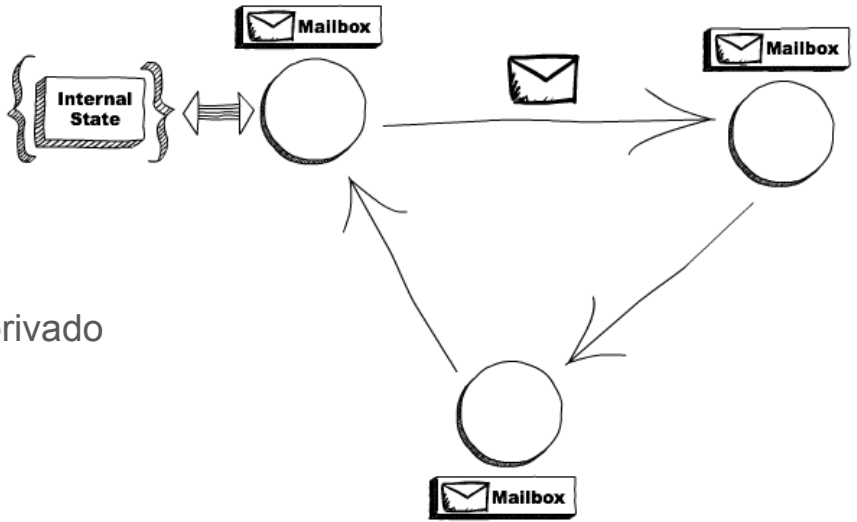
Erlang está “basado” en el modelo de actores.

El modelo postula que la unidad primitiva de computación es el “actor”.

Este se comunica mediante el pasaje de mensajes asincrónicos con otros actores.

Al recibir un mensaje, un actor puede:

- Crear actores
- Enviar mensajes
- Realizar una computación y modificar su estado privado



“OOP to me means only **messaging**, local retention and protection and **hiding of state-process**, and extreme late-binding of all things…”

-Alan Kay on the Meaning of “Object-Oriented Programming”

Similitudes y diferencias con OOP

OOP (objetos)

- Los objetos se comunican mediante pasaje de mensajes
- Poseen identidad propia (referencia)
- Poseen estado interno y aislado
- Pasaje de mensajes sincronico

Erlang (procesos)

- Los procesos se comunican mediante pasaje de mensajes
- Poseen identidad propia (PID)
- Poseen un estado interno y aislado
- Pasaje de mensajes asíncrono

Procesos

- Poseen un identificador único, el PID
- Los mensajes son términos de Erlang (átomos, tuplas, listas, PIDs, etc)
- Poseen un estado interno
- Poseen un mailbox, cola donde guardan todos los mensajes que el proceso recibe y que ha de procesar
- El estado de cada proceso es independiente del de los demás.

```
% Para crear un proceso se usa la función spawn
% Module: Módulo de la función que se va a ejecutar
% Function: Función que ejecutará el proceso
% Args: Lista de argumentos para la función Function

1> spawn(Module, Function, Args).
% <0.233.0>
2> spawn(lists, seq, [1, 10]).
% <0.258.0>
```

Enviar mensajes

Un proceso de Erlang puede mandar mensajes a cualquier otro proceso de Erlang.

Para dirigir el mensaje se utiliza el PID.

El mensaje puede ser cualquier dato de Erlang, es decir: enteros, átomos, tuplas, listas, etc.

El envío de mensajes es asíncrono, al mandar un mensaje no se espera respuesta (similar a UDP)

El envío de mensaje retorna como valor el mensaje.

```
1> Pid = spawn(...).
% <0.123.0>
2> Pid ! 55.
% 55
3> self() ! {msg, "hello world"}.
% {msg, "hello world"}
4> self() ! Pid ! [1,2,3]
% [1,2,3]
```

Recibir mensajes

Para recibir mensajes se usa ***receive***, en el se declaran varias expresiones contra las que hace *pattern matching* al recibir un mensaje y ejecuta el brazo que haga ***match***.

El ***receive*** también puede tener un ***after***, a este se le da un valor de milisegundos y si después de ese tiempo ningún mensaje ha hecho ***match*** se ejecuta su brazo.

Los mensajes son procesados en forma FIFO.

```
receive
 {add_5, From, Num} ->
   From ! Num + 5;
 {msg, Msg} ->
   io:format("~p~n", [Msg])
after 2000 ->
 io:format("No hay mensajes")
end.
```

Ejercicio!
Calculator

Escriban una calculadora simple (suma, resta, multiplicación y división) usando procesos y envío/recepción de mensajes. Para esto escriban un modulo **calculator** con las siguientes funciones:

- start_calculator/0 : esta función creará un proceso con la siguiente función calculator:calculator_server/0
- calculator_server/0 : Recibirá a través de mensajes la operación a ejecutar y sus argumentos, después devolverá el resultado en un mensaje.
- turn_off/1: Apaga el servidor de la calculadora.

Adicionalmente crea los funciones add/3, subtract/3, multiply/3, division/3 para abstraer las operaciones. Estas funciones recibirán el PID del servidor de la calculadora y los dos valores sobre los que se quieren operar y devolverán el valor devuelto por la calculadora.

Temas no cubiertos

OTP

Supervisión de procesos

Behaviours

gen_server, gen_statem, gen_event

Mnesia, ETS, DETS

Herramientas del lenguaje para tracing y debugging

Ports

¡Gracias por su atencion!