

FLEX - Erlang

Simon Härer, Michael Sperber

Created: 2023-04-25 Tue 08:07

Live Coding

These are slides for accreditation purposes, but the best mode of presentation for this is a live coding demo, where the participants can also try out programming in Erlang.

Simple Data

```
42      % number  
true    % boolean  
false   % boolean  
foo     % atom
```

```
23 + 42  
42 == 23  
42 >= 23  
true andalso false  
false orelse true
```

Data Structures

```
{1,foo,true} % tuple  
[1,2,3]      % list  
"abc"        % string = list of numbers  
<<1,17,42>>  
<<1,17,42:16>> == <<1,17,0,42>>.
```

```
{a => "hello"} % map  
{1 => 2, b => b}
```

```
[1,2,3] ++ [4,5,6]  
"abc" ++ "ABC"
```

Maps

```
A = a,  
B = b,  
M0 = # {},  
M1 = # {a => <<"hello">>},  
M2 = # {1 => 2, b => b},  
M3 = # {k => {A,B}},  
M4 = # {{ "w", 1 } => f() }.
```

Map Expressions

$M = \# \{ 1 \Rightarrow a \} .$

$M \# \{ 1.0 \Rightarrow b \} .$

$\# \{ 1 \Rightarrow a, 1.0 \Rightarrow b \} .$

$M \# \{ 1 := b \} .$

$\# \{ 1 \Rightarrow b \}$

Functions

```
double (X) ->  
    times (X, 2) .
```

```
times (X, N) ->  
    X * N.
```

Functions with Status

```
divide(N, M) ->  
  if  
    M == 0 -> divide_by_zero;  
    true -> {ok, N / M}  
  end.
```


Pattern Matching

```
dogs_per_legs(Legs) ->  
  case divide(Legs, 4) of  
    {ok, People} -> People;  
    divide_by_zero -> io:format("this can't happen")  
  end.
```

Pattern Guard

```
divide(_N, M) when M == 0 ->  
    divide_by_zero;  
divide(N, M) ->  
    {ok, N / M}.
```

Functions

```
area({square, Side}) ->  
    Side * Side;  
area({circle, Radius}) ->  
    3.14159265 * Radius * Radius;  
area({triangle, A, B, C}) ->  
    S = (A + B + C) / 2,  
    math:sqrt(S * (S-A) * (S-B) * (S-C));  
area(Other) ->  
    {invalid_object, Other}.
```

Functions

```
length([]) ->  
    0;  
length([H|T]) ->  
    1 + length(T).
```

Modules

```
-module(animals) .  
  
-export([map/2, map_iterative/3]) .  
  
-include_lib("eunit/include/eunit.hrl") .
```

Records

```
-record(dillo, {alive :: boolean(), weight :: pos_integer()}).  
-record(rattlesnake,  
        {thickness :: non_neg_integer(), length :: pos_integer()}).
```

```
d1() -> #dillo{alive=true, weight=10}.  
d2() -> #dillo{alive=false, weight=12}.  
r1() -> #rattlesnake{thickness=10, length=100}.  
r2() -> #rattlesnake{thickness=0, length=120}.
```

Functions

```
-type animal() :: #dillo{} | #rattlesnake{}.  
  
% run over one animal  
-spec run_over_animal(animal()) -> animal().  
  
run_over_animal(#dillo{weight=Weight}) ->  
    #dillo{alive=false, weight=Weight};  
run_over_animal(#rattlesnake{length=Length}) ->  
    #rattlesnake{thickness=0, length=Length}.
```

Test

```
run_over_animal_test() ->  
  #dillo{alive=false, weight=10} = run_over_animal(d1()),  
  ?assert(d2() == run_over_animal(d2())),  
  #rattlesnake{thickness=0, length=100} = run_over_animal(r1()),  
  R2 = r2(),  
  R2 = run_over_animal(r2()).
```


Functions over Lists

```
% run over a list of animals
-spec run_over_animals([animal()]) -> [animal()].

run_over_animals_test() ->
  I = [d1(), d2(), r1(), r2()],
  O = [#dillo{alive=false, weight=10}, d2(),
       #rattlesnake{thickness=0, length=100}, r2()],
  O = run_over_animals(I).

run_over_animals([]) ->
  [];
run_over_animals([A|As]) ->
  [run_over_animal(A) | run_over_animals(As)].
```

Higher-Order Function

```
map(_F, []) ->
    [];
map(F, [First|Rest]) ->
    [F(First) | map(F, Rest)].

run_over_animals(As) ->
    map(fun run_over_animal/1, As).
```

Tail-Recursive Function

```
map_iterative(_F, [], Acc) ->  
    lists:reverse(Acc);  
map_iterative(F, [First|Rest], Acc) ->  
    map_iterative(F, Rest, [F(First)|Acc]).
```

Process and Message Reception

```
format_server_0() ->
  spawn(fun () ->
    receive
      Bla -> io:format(Bla)
    after 10000 ->
      io:format("timeout~n")
    end
  end) .
```

Message Passing in Erlang

- asynchronous (every process has a queue)
- queues are unbounded
- ordering is preserved
- cross-node message-passing may lose messages

Process as Server

```
format_server() ->  
  receive  
    Bla -> io:format(Bla),  
          format_server()  
end.  
  
start_format_server() ->  
  spawn(process, format_server, []).
```

Replies, records as messages

```
-record(get, {sender_pid :: pid()}).  
-record(inc, {i :: number()}).
```

```
inc_loop(N) ->  
  receive  
    #inc{i = I} ->  
      io:format("incrementing ~w by ~w~n", [N, I]),  
      inc_loop(N + I);  
    #get{sender_pid = SenderPid} ->  
      SenderPid ! N,  
      inc_loop(N)  
  end.
```

Interface via functions

```
start_inc(N) ->
    spawn(process, inc_loop, [N]).

inc(Pid, I) ->
    Pid ! #inc{i = I}.

get(Pid) ->
    Pid ! #get{sender_pid = self()},
    receive
        N -> N
    end.
```


gen_server

```
-module (inc_gen_server) .  
  
-export ([start/1, start_link/1,  
         inc/1, inc/2, get/0, get/1,  
         init/1, handle_cast/2, handle_call/3]) .  
  
-behavior (gen_server) .
```

Inc gen_server

```
start(N) ->  
    gen_server:start({global, global_inc}, ?MODULE, N, []).
```

```
start_link(N) ->  
    gen_server:start_link(?MODULE, N, []).
```

```
-record(get, {sender_pid :: pid()}).  
-record(inc, {i :: number()}).
```

Inc gen_server calls

```
inc(I) ->  
    inc(global_name(), I).
```

```
inc(Pid, I) ->  
    gen_server:cast(Pid, #inc{i = I}).
```

```
get(Pid) ->  
    gen_server:call(Pid, #get{}).
```

```
get() ->  
    get(global_name()).
```

Inc gen_server callbacks

```
init(N) -> {ok, N}.
```

```
handle_cast(#inc{i = I}, N) ->  
    io:format("incrementing ~w by ~w~n", [N, I]),  
    {noreply, N + I}.
```

```
handle_call(#get{}, _From, N) ->  
    {reply, N, N}.
```

Link

```
die_process() ->
    receive
        Msg -> io:format("~w~n", [10 / Msg]),
                die_process()
    end.

start_die_process() ->
    Pid = spawn(?MODULE, die_process, []),
    link(Pid),
    process_flag(trap_exit, true), % try removing this line
    Pid.
```

Link

- `link(Pid)` creates *bidirectional* link between `self()` and `Pid`
- `spawn_link` **is** `spawn + link`, atomically

Supervisors

```
-module(supervisor_demo).  
  
-export([start/0, start_link/0, init/0, die_process/0]).  
  
start() ->  
    spawn(?MODULE, init, []).  
  
start_link() ->  
    spawn_link(?MODULE, init, []).  
  
die_process() ->  
    receive  
        Msg -> io:format("~w~n", [10 / Msg]),  
  
        die_process()  
  
    end.
```

Supervisors

```
init() ->
    process_flag(trap_exit, true),
    loop().

loop() ->
    Pid = spawn_link(?MODULE, die_process, []),
    register(die_process, Pid),
    receive
        {'EXIT', _From, shutdown} ->
            exit(shutdown); % will kill the child too
        {'EXIT', Pid, Reason} ->
            io:format("Process ~p exited for reason ~p~n",
                      [Pid, Reason]),

            loop()

    end.
```