Advanced Programming

Introduction to Erlang

Oleks Shturmov
oleks@oleks.info

Slides adapted from Ken Friis Larsen
kflarsen@diku.dk

Department of Computer Science
University of Copenhagen

September, 2021

Part I

Introduction

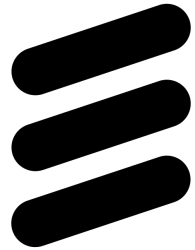
Erlang — What's in a name?

"We deliberately encouraged this ambiguity" – Joe Armstrong, HOPL III

The Man

- Agner Krarup Erlang
- 1878–1929
- Invented queueing theory and traffic engineering, which is the basis for telecommunication network analysis.

The Company



ERICSSON

- Swedish multinational networking and telecommunications company
- Founded by Lars Magnus Ericsson in 1876
- Birthplace of the Erlang Programming Language

The Erlang Programming Language

- Developed in the early 1980s, by the guys above[1]
- All while working at Ericsson, programming telephone switches
- Useful for distributed, fault-tolerant systems, in general
- Open-sourced in 1998, still maintained by Ericsson
- Used today by WhatsApp (Facebook), Nintendo, Discord, etc.

[1] Image source: `https://www.youtube.com/watch?v=rYkI0_ixRDc`

Erlang Customer Declaration

Erlang is a:

- a concurrency-oriented language
- dynamically typed
- with a strict functional core language

On the matter of Erlang syntax

- Syntax heavily inspired by the Prolog programming language
- **Semantically, Erlang bears little resemblance to Prolog**
  - Prolog is a *logic programming language*
  - A distinct programming paradigm, beyond the scope of this course
- We won't delve on this history, except to justify peculiar syntax

On the matter of Erlang syntax

- Syntax heavily inspired by the Prolog programming language
- **Semantically, Erlang bears little resemblance to Prolog**
  - Prolog is a *logic programming language*
  - A distinct programming paradigm, beyond the scope of this course
- We won't delve on this history, except to justify peculiar syntax
- For example:
  - The Erlang shell prompts for inputs of the form:

$$Body_1 \text{ ‘,’ } Body_2 \text{ ‘,’ } \ldots \text{ ‘,’ } Body_n \text{ ‘.’} \quad \text{(where } n \geq 1\text{)}$$

  - We may read comma (‘,’) as “and then”
  - Note in particular, the period (‘.’) at the end
  - ‘,’ and ‘.’ play similar roles in functions declarations

Part II

Basic Concepts

`erl`

On Windows, the Erlang executable

- Starts an Erlang runtime system (i.e., an Erlang node)
- Drops you into an Erlang shell for that node

```
$ erl
Erlang/OTP 24 [erts-12.0.3] [source] [64-bit] [...]

Eshell V12.0.3  (abort with ^G)
1> 21+21.
42
2>
```

- Use one of the following options to quit/kill `erl`:
  - Enter the command `q()`.
  - Press `Ctrl+g`, and enter the command `q` (quit)
  - Press `Ctrl+c`, and enter the command `a` (abort)

Fundamental Stuff

- We have integers, floating-point numbers, and arithmetic:

```
1> 21+21.
42
2> 3/4.
0.75
3> 5 div 2.
2
```

- We have lists:

```
4> [21,32,67] ++ [100,101,102].
[21,32,67,100,101,102]
```

- Strings are just lists of characters, and characters are just integers:

```
5> "Sur" ++ [112, 114, $i, $s, $e].
```

Fundamental Stuff

- We have integers, floating-point numbers, and arithmetic:

```
1> 21+21.
42
2> 3/4.
0.75
3> 5 div 2.
2
```

- We have lists:

```
4> [21,32,67] ++ [100,101,102].
[21,32,67,100,101,102]
```

- Strings are just lists of characters, and characters are just integers:

```
5> "Sur" ++ [112, 114, $i, $s, $e].
"Surprise"
```

Fundamental Stuff

- We have booleans and comparison operators:

```
1> 1 > 2.
false
```

Fundamental Stuff

- We have booleans and comparison operators:

```
1> 1 > 2.
false
```

- However, Erlang sometimes coerses integers to floats!

```
2> 0.9999999999999999 >= 1. % That's 17 9's.
true
```

Operator	Meaning	Strict?
>	..	No
<	..	No
>=	..	No
=<	**Less than or equal to**	No
==	Equal to	No
/=	Not equal to	No
=:=	Equal to	**Yes!**
=/=	Not equal to	**Yes!**

Names (Variables)

- Names (variables) start with an upper-case letter

```
1> Homer = "Homer".
Homer
2> X=5, Y=2, X*Y.
10
```

- Once assigned, variables cannot be re-assigned[2]

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

[2] Perhaps another relic of the Prolog past, or part of the "strict functional core"

What to do if you mess up in `erl`?

- `Eshell` is quite forgiving — you can just let exceptions happen
- In non-exceptional cases[3], you might be tempted to kill `erl`...
- A better option is to press `Ctrl+g`:

```
4> X/0
4> % Pressing Ctrl+g ...
User switch command
 -> h
  c [nn]            - connect to job
  i [nn]            - interrupt job
  ...
  q                 - quit erlang
  ? | h             - this message
 -> i
 -> c
** exception exit: killed
4> X/2.
2.5
```

- Bonus: you get to keep your names (variables)!

[3] Long-running, live-, or dead-locked commands

Tuples and Atoms

- Erlang uses curly braces for tuples:

```
1> {"Bart", 9}.
{"Bart",9}
```

- Atoms are used to represent non-numerical constant values (like `enums` in C and Java). Atom is a sequence of alphanumeric characters (including `@` and _) that starts with a lower-case letter (or is enclosed in single-quotes):

```
2> bloody_sunday_1972.
bloody_sunday_1972
3> [{bart@simpsons, "Bart", 9}, {'HOMER', "Homer", 42}].
[{bart@simpsons,"Bart",9},{'HOMER',"Homer",42}]
```

Patterns

- As in Haskell, we can use patterns to take things apart:

```
1> P = {point, 10, 42}.
2> [ C1, C2, C3 | Tail ] = "Homer".
"Homer"
3> C2.
111
4> Tail.
"er"
5> {point, X, Y} = P.
{point,10,42}
6> X.
10
7> Y.
42
```

List Comprehensions

```
1> Digits = [0,1,2,3,4,5,6,7,8,9].
[0,1,2,3,4,5,6,7,8,9]
2> Evens = [ X || X <- Digits, X rem 2 =:= 0].
[0,2,4,6,8]
3> Cross = [{X,Y} || X <- [1,2,3,4], Y <- [11,22,33,44]].
[{1,11}, {1,22}, {1,33}, {1,44},
 {2,11}, {2,22}, ... ]
4> EvenXs = [{X,Y} || {X,Y} <- Cross, X rem 2 =:= 0].
[{2,11},{2,22},{2,33},{2,44},{4,11},{4,22},{4,33},{4,44}]
```

Maps

```
1> M = #{ name => "Ken", age => 45}.
#{age => 45, name => "Ken"}
2> ClunkyName = maps:get(name, M).
"Ken"
3> #{name := PatternName} = M.
4> PatternName.
"Ken"
5> #{name := Name, age := Age} = M.
#{age => 45, name => "Ken"}
6> {Name, Age}.
{"Ken", 45}
7> Wiser = M#{age := 46}.
#{age => 46, name => "Ken"}
8> WithPet = M#{pet => {cat, "Toffee"}}.
#{age => 46, name => "Ken", pet => {cat, "Toffee"}}
```

Functions

- Remember the `move` function from Exercise Set 0 (Haskell)?

```
move :: Direction -> Pos -> Pos
move North (x,y) = (x, y+1)
move West  (x,y) = (x-1, y)
```

Functions

- Remember the `move` function from Exercise Set 0 (Haskell)?

```
move :: Direction -> Pos -> Pos
move North (x,y) = (x, y+1)
move West  (x,y) = (x-1, y)
```

- In Erlang:

```
move(north, {X, Y}) -> {X, Y+1};
move(west, {X, Y}) -> {X-1, Y}.
```

(note that we use semicolon to separate clauses, and period to terminate a declaration).

Functions

- Remember the `move` function from Exercise Set 0 (Haskell)?

```
move :: Direction -> Pos -> Pos
move North (x,y) = (x, y+1)
move West  (x,y) = (x-1, y)
```

- In Erlang:

```
move(north, {X, Y}) -> {X, Y+1};
move(west, {X, Y}) -> {X-1, Y}.
```

(note that we use semicolon to separate clauses, and period to terminate a declaration).

- Or naming a function literal:

```
Move = fun(Dir, {X,Y}) ->
               case Dir of
                   north -> {X, Y+1};
                   west  -> {X-1, Y}
               end
       end.
```

Modules

- If we want to declare functions (rather than naming literals) then we need to put them in a module.
- Modules are defined in `.erl` files, for example `somemodule.erl`:

```
-module(somemodule).
-export([move/2, qsort/1]).

move(north, {X, Y}) -> {X, Y+1};
move(west, {X, Y}) -> {X-1, Y}.

qsort([]) -> [];
qsort([Pivot|Rest]) ->
    qsort([X || X <- Rest, X < Pivot])
    ++ [Pivot] ++
    qsort([X || X <- Rest, X >= Pivot]).
```

- Note, how we **specify the arity** of functions on export

Compiling Modules

- Using the function `c`, we can compile and load modules in the Erlang shell:

```
1> c(somemodule).
{ok,somemodule}
```

- We can now call functions from our module:

```
2> somemodule:qsort([101, 43, 1, 102, 24, 42]).
[1,24,42,43,101,102]
```

- Or use them with functions from the standard library:

```
3> Moves = [{north, {1,1}}, {west, {43,42}},
            {north, {23,22}}].
4> lists:map(fun({Dir,Pos}) ->
                  somemodule:move(Dir, Pos) end,
                Moves).
[{1,2},{42,42},{23,23}]
```

Student Activation: Define your favourite function