**Introduction to Erlang**

**Erlang:** Functional language with built in concurrency support

**OTP:** A large collection of libraries for Erlang

**Features:**
- Concurrency and asynchronous message passing
- Lightweight processes. Fast context switches
- Virtual machine
- Not suitable for low-level system software

**History:**
- Named after mathematician Agner Erlang
- Originated from Ericsson (maybe Erlang actually stands for ERicsson LANGUAGE?)
- Used for a lot of telecoms applications: e.g. switches
- Open sourced in 1998

**Basics: Sequential Programming**

- Numbers: Integers (1, -10), Floats (3.1415, -0.23)
  - Hex: 16#AB123 Binary: 2#100110
  - ASCII: $A$ (65), $Z$ (122), etc.
- Atoms: hello, how are you, ’I am fine’
- Variable: Counter, Good_server, BadServer
  - Only bound once. Value cannot be changed once bound!!!
- Operators: +, -, *, /, >, >=, <, <=, ==, /=
Data Structures:

- **Tuples:** `{123, hello, 'Good Morning', {super, 456}}, {}`
- **Lists:** `[123, hello, 'Welcome'], [], "abcdefg", ""`
- **Combinations:** `[(123, house), guest, {friends, family}], {123, [1,2,3,4], "building"}`
- **Others (dict, process dictionary, etc.):** see documentation

Pattern Matching:

Binding variables to values

- `A = 10`
- `(B, C, D) = {10, foo, bar}`
- `{A, A, B} = {abc, abc, foo}`
- `(A, A, B) = {abc, def, 123}`
- `[A,B,C] = [1,2,3]`
- `[A,B,C,D] = [1,2,3]`
- `[A,B,C,D] = [1,2,3,4,5,6,7]`
- `[A|B] = [abc]`
- `[A|B] = []`
- `(A,.., B) = {123, 456, 789}`

Functions:

Function definition (in a module)

```erlang
-module(math).
-export([factorial/1]).
```

```erlang
% this calculates factorial
factorial(0) ->
    1;
factorial(N) ->
    N * factorial(N-1).
```

Function use

```erlang
2> math:factorial(5).
120
```

Function Evaluation Rules:

- Clauses scanned until a match is found
- All variables in function head are bound
- Variables are local to each clause
- Body evaluated sequentially

Built In Functions:

- In module `erlang`
- Do what you cannot (easily) do in Erlang
Anonymous Functions:

```erlang
F = fun(X) -> X*2 end.
F(2).
```

Punctuation:

Easiest way to think about it:

- `;` is AND
- `.` is OR
- `.` is END

Example:

```erlang
factorial(0) ->
   1; % OR
factorial(N) ->
   io:format("factorial ~w~n", [N]), % AND
   N * factorial(N-1). % END
```

Concurrent Programming

Processes:

```erlang
Pid = spawn(Mod, Func, Args)
```

Creates a new process that evaluates the given function with the given arguments

```erlang
Pid = spawn(math, factorial, [12]).
```

With anonymous functions (most useful):

```erlang
F = fun() -> io:format("Hello!") end.
Pid = spawn(F).
```

Message Passing:

A does:

```erlang
B ! {self(), hello, you}
```

This sends a message `(A, hello, you)` to process `B`

In order to receive the message `B` does:

```erlang
receive
   {From, Msg1, Msg2} -> ...
end
```

Processing messages:

- queue messages in arrival order
- test each message against all receive clauses – until match
- wait for more messages if no match
Selective Message Reception:

A: C!foo

B: C!bar

C:

receive
    foo -> true
end,
receive
    bar -> true
end

→ foo is received before bar no matter what order they were sent in (or how they were queued).

Timeouts:

Wait a given amount of time (milliseconds)

sleep(T) ->
    receive
    after
        T -> true
    end.

Wait forever

suspend() ->
    receive
    after
        infinity -> true
    end.

Closures (very useful)

Values of bound variables are passed along in messages
-module(closures).
-export([do_send/4, do_receive/0]).

do_send(Dest, A, B, C) ->
    Dest ! {msg, fun(D) ->
        io:format("A: ~s, B: ~s, C: ~s, D: "+s"-n", [A, B, C, D]) end}.

do_receive() ->
    receive
        {msg, F} -> F("woohoo")
    end.

1> B = spawn(fun() -> closures:do_receive() end).
2> closures:do_send(B, "hello", "there", "friend")
A: hello, B: there, C: friend, D: woohoo
**Why is Erlang Good for Distributed Systems?**

1. Built-in support for message passing
2. Light-weight processes
3. Functional language:
   - no global state
   - no concurrent access of global state
   - Note: it’s possible to have global state, but avoid this!
4. Error handling

---

**Output:**

\[
\text{io:format(FormatString, ArgList)}
\]

**Examples**

1. \( \text{io:format("Hello world!\n", []).} \)
   \text{Hello world!} \text{ok}
2. \( \text{io:format("arg1:~w, arg2:~w, arg3:~w", [1,2,5]).} \)
   \text{arg1:1, arg2:2, arg3:5} \text{ok}

---

**Guarded Function Clauses:**

\[
\text{factorial(N) when N > 0} - \text{factorial(N - 1);} \\
\text{factorial(0) -> 1.}
\]

**Examples**

- \( \text{is_number(X)} \) - X is a number
- \( \text{is_atom(X)} \) - X is an atom
- \( \text{is_tuple(X)} \) - X is a tuple
- \( \text{is_list(X)} \) - X is a list
- See documentation for more (http://www.erlang.org/documentation/doc-5.9.1/doc/index.html)
Case and If:

```erlang
case X of
    {yes, _} -> ...;
    {no, _} -> ...;
    _Else -> ...
end,
... 
```

Recursion and List Traversal:

Common patterns

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

```erlang
double_list([H|T]) -> [2*H|double_list(T)];
double_list([]) -> [].
```

What happens:

```erlang
double_list([1,2,3]).
double_list([1,2,3]) => [2|double_list([2,3])]
double_list([2,3]) => [4|double_list([3])]
double_list([3]) => [6|double_list([])]

[2,4,6]
```

List Comprehensions:

```erlang
List = [ X || X <- L, Filter ].
```

Example:

```erlang
Y = [ 1/X || X <- List, X > 0].
```
Useful functions for lists:

```erlang
lists:filter(fun(E) -> E rem 2 == 0 end, List).
lists:map(fun(E) -> E * 2 end, List).
lists:flatten([[1,2,3],[4,5,6],[[7,8], 9, [10]]]).
lists:unzip([[1,a], [2,b], [3,c]]). -> {{1,2,3},[a,b,c]}
lists:zip([1,2,3],[a,b,c]). -> [{1,a},{2,b},{3,c}]
```

**Some Useful Libraries**

- **stdlib:**
  
  - `io`: read, write, format, etc.
  - `lists`: append, concat, flatten, reverse, sort, member, etc.
  - `string`: len, equal, concat, substr, strip, etc.
  - `dict`: new, find, store, fetch, update, etc.
  - `math`: sin, cos, tan, exp, log, pow, sqrt, etc.

**Error Handling**

**Try - Catch:**

```erlang
catch_error(N) ->
  try error_func(N) of
  {ok, Ret} -> io:format("SUCCES: ~w\n", [Ret])
  catch
  throw:Err -> io:format("THROW: ~w\n", [Err]);
  exit:Err -> io:format("EXIT: ~w\n", [Err]);
  error:Err -> io:format("ERROR: ~w\n", [Err])
  after
  io:format("All Done\n")
  end.
error_func(1) -> throw(woops);
error_func(2) -> exit(woops);
error_func(3) -> erlang:error(woops);
error_func(N) -> {ok, N}.
```

**Trap Exit:**

```erlang
trapper(N) ->
  process_flag(trap_exit, true),
  Pid = spawn(fun() -> exiter(N) end),
  link(Pid),
  receive
  {'EXIT', Pid, Why} -> io:format("~w exited with ~w\n", [Pid, Why])
  end.
exiter(1) -> exit(1);
exiter(2) -> 1/0;
exiter(N) -> true.
```
Dynamic Code Loading

- `module(dyn).
- `export([start/0]).

`start() -> spawn(fun() -> dyn_loop() end).

`dyn_loop() -> io:format("a = "++"n",[dyn_a:a()]),`sleep() -> `receive after 3000 -> true end.

- `module(dyn_a).
- `export([a/0]).

`a() -> 1.

3> `dyn:start().
a = 1
a = 1
% change `dyn_a.erl to return 2
4> `c(dyn_a).
{ok,dyn_a}
a = 2

Erlang Resources

http://www.erlang.org

Documentation http://www.erlang.org/doc.html

Introductory Course (Do This!) http://www.erlang.org/course/course.html

1/doc/man_index.html

Erlang Books http://learnyousomeerlang.com


Homework

Client-Server in Erlang:

See Exercises: Client server exercise (Erlang), Part A.

Hacker’s edition: Performance of Erlang:

Evaluate how long it takes to create processes in Erlang
How about processes on another machine?
Evaluate how long it takes to send messages in Erlang
Local: same core? different cores?
Remote: same cluster, same LAN? over WAN?

Watch the Movie!

http://www.youtube.com/watch?v=uKfKtXYLG78