**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

**THE ERLANG ENVIRONMENT**

```
unix% erl
1> 1 + 2.3
3
2> c(demo).
{ok, demo}
3> demo:double(25).
50
4> date().
{2004,2,24}
5> halt().
```

**THE ERLANG ENVIRONMENT**

```
unix% cat demo.erl
-module(demo).
-export([double/1]).
double(X) -> 2 * X.
```

**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] = [1,2,3,4,5,6,7]
- [A|B] = [abc]
- [A|B] = []
- (A, ,B) = {123, 456, 789}

Functions:
Function definition (in a module)
-module(math).
-export([factorial/1]).

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

Function use
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:

\[ \text{F} = \text{fun}(X) \rightarrow X \times 2 \text{ end.} \]
\[ \text{F}(2). \]

Punctuation:
Easiest way to think about it:
\[ \rightarrow \text{ is AND} \]
\[ \Rightarrow \text{ is OR} \]
\[ \blacksquare \text{ is END} \]

Example:
\[ \text{factorial}(0) \rightarrow 1; \% \text{ OR} \]
\[ \text{factorial}(N) \rightarrow \]
\[ \text{io:format}(\text{"factorial \~w\~n", \[N\]}, \% \text{ AND} \]
\[ N \times \text{factorial}(N-1). \% \text{ END} \]

Processes:
\[ \text{Pid} = \text{spawn}(\text{Mod, Func, Args}) \]

Creates a new process that evaluates the given function with
the given arguments
\[ \text{Pid} = \text{spawn(math, factorial, \[12\]).} \]

With anonymous functions (most useful):
\[ \text{F} = \text{fun()} \rightarrow \text{io:format}("Hello!") \text{ end.} \]
\[ \text{Pid} = \text{spawn(F)}. \]

Message Passing:
A does:
\[ B ! \{\text{self()}, \text{hello}, \text{you}\} \]

This sends a message \((A, \text{hello}, \text{you})\) to process \(B\).
In order to receive the message \(B\) does:
\[ \text{receive} \]
\[ \{\text{From, Msg1, Msg2}\} \rightarrow \ldots \]
\[ \text{end} \]

Processing messages:
\[ \rightarrow \text{ queue messages in arrival order} \]
\[ \rightarrow \text{ test each message against all receive clauses -- until match} \]
\[ \rightarrow \text{ wait for more messages if no match} \]
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", [A, B, C, D]) end}.
do_receive() ->
  receive
    {msg, F} -> F("woohoo")
  end.

0 is special
flush() ->
    receive
      Any -> flush()
    after
      0 -> true
    end.

0 means:
→ Check message buffer
→ If empty execute the given code (true)
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:

```
io:format(FormatString, ArgList)
```

Examples

1> io:format("Hello world!\n", []). Hello world! ok
2> io:format("arg1:~w, arg2:~w, arg3:~w", [1,2,3]). arg1:1, arg2:2, arg3:3 ok

Guarded Function Clauses:

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

Examples

- `is_number(X)` - X is a number
- `is_atom(X)` - X is an atom
- `is_tuple(X)` - X is a tuple
- `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]).
```

```erlang
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:

- `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

- stlib:
  - 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:

```
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:

```
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 = "++[a:a()]), sleep(), dyn_loop().
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

Homework

Client-Server in Erlang:
- Simple address database server and client
- 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?

Erlang Resources

http://www.erlang.org

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

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

Man pages http://www.erlang.org/documentation/doc-5.9.1/doc/man_index.html

Erlang Books http://learnyousomeerlang.com


Watch the Movie!

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