Distributed and Parallel Computing

Erlang

Alan P. Sexton

University of Birmingham

Spring 2018

Erlang

Erlang is a functional programming language with some peculiarities stemming from its original implementation in Prolog.

```
-module(hello).
-export([hello_world/0, start/0, echo/0, echo/1]).

hello_world() ->
    io:fwrite("hello, world~n").

start() ->
    io:fwrite("start/0 called~n").

echo()->
    io:fwrite("echo/0 called~n").

echo([]) ->
    ok;

echo([Hd|Tl]) ->
    io:format("~s~n", [Hd]),
    echo(Tl).
```

Compiling and Running

If this program is in file `hello.erl`, it can be compiled and run from the (Linux) command line as follows:

```
$ erlc hello.erl
$ erl -noshell -run hello hello_world -run init stop
hello, world
$ erl -noshell -run hello -run init stop
start/0 called
$ erl -noshell -run hello echo -run init stop
echo/0 called
$ erl -noshell -run hello echo arg1 2 b -run init stop
arg1
2
b
```

Resources

- A good book on Erlang, which is freely accessible online:
  `http://learnyousomeerlang.com/`
  For this course you need at most only the chapters up to and including "Designing a Concurrent Application" (Chapter 13)
- The official Erlang documentation page:
  `http://erlang.org/doc/`
- The Erlang reference manual:
  `http://erlang.org/doc/reference_manual/users_guide.html`
- The Standard Library documentation:
  `http://erlang.org/doc/apps/stdlib/index.html`
- The official Erlang tutorial:
  `http://erlang.org/doc/getting_started/users_guide.html`.
  This covers nearly everything you need for this module and covers quite a lot that you do not need. It is also much shorter than the book above.

Basics of Erlang

- Variables start with upper case letters (X, Var, Lst)
- Atoms start with lower case letters or are enclosed in single quotes (atom, y, 'This is an atom')
- = binds (and matches):
  - `X = 2` if X is free, binds X to 2. If X is already bound to 2, succeeds, if X is bound to something else, fails and throws an exception.
  - `[2|Tl]= [2,3,4]` binds Tl to [3,4]
- Exact equality: `=:=` inequality: `=/=`
- Inexact equality: `==` inequality: `/=`
- Comparison: `< > =< >=`
- Tuples: `{a, 2, "abc"}`
- Lists: `[a, 2, "abc"], [Hd | Tl], hd([1,2,3]), tl([1,2,3])`

More Basics of Erlang

- Functions:

```
fn(Pattern1) ->
    Expr1;
fn(Pattern2) ->
    Expr2;
fn(PatternN) ->
    ExprN.
```

- Sequence of expressions separated by “,”

Sending and Receiving Messages

```
-module(t1).
-compile(export_all).

sen()->
    self() ! 1,
    self() ! 2,
    self() ! 3.


rec() ->
    receive X ->
            io:format("Received: ~p~n", [X])
    after 2000 ->
            timeout
    end.
```

Running t1

The t1 program of the previous slide is not intended to be run as a command line program, but rather to be run interactively using the Erlang shell:

```
$ erl
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:20:20]...

Eshell V7.3  (abort with ^G)
1> t1:sen().
3
2> t1:rec().
Received: 1
ok
3> t1:rec().
Received: 2
ok
4> t1:rec().
Received: 3
ok
5> t1:rec().
timeout
6> q().
ok
7> $
```

Running the Erlang Shell

- The `c(t1).` command compiles and loads the `t1` module
- An erlang module with name “`x`” must be in a file with name “`x.erl`” and have the `-module(x).` command at the start of the file
- Within the shell you can use the arrow keys to select previous/next lines or the edit the current line
- The `q().` command wil quit the shell, as will `<^C^C>`. You can call other user commands with `<^G>` (type `?<Return>` to see a list of options).
- Within the shell, after you have compiled and loaded a module, to call functions from that module you must prefix the function name with the module name and a ":"

Receiving Multiple Messages, Selectively

```
-module(t2).
-compile(export_all).

sen()->
    self() ! 1,
    self() ! 2,
    self() ! 3.

selrec() ->
    receive
        X when X > 2 ->
            io:format("Selectively Received: ~p~n", [X]),
            selrec()
    after 0 ->
            normalrec()
    end.

normalrec() ->
    receive X ->
            io:format("Normally Received: ~p~n", [X]),
            normalrec()
    after 0 ->
            ok
end.
```

Receiving Multiple Messages, Non-Selectively

```
-module(t3).
-compile(export_all).

sen()->
    self() ! 1,
    self() ! 2,
    self() ! 3.

largerec() ->
    receive
        X ->
            case X of
                X when X > 2 ->
                    io:format("Large Received: ~p~n", [X]);
                X ->
                    io:format("Small Received: ~p~n", [X])
            end,
            largerec()
    after 0 ->
            ok
    end.
```

Spawning Processes

```
-module(t4).
-compile(export_all).

start() ->
    Pid = spawn(t4, node, ["node"]),
    Pid ! {self()},
    receive X ->
            io:format("Start ~p got: ~p~n", [self(), X])
    end.

node(N) ->
    receive
        {X} ->
            io:format("Node ~p got: ~p~n", [self(), X]),
            X ! {self(), N};
        X ->
            io:format("Node ~p got bad message: ~p~n",
                      [self(), X])
    end.
```