Erlang Lecture 1

Erlang Basics
(adapted from Cooper Filby)

Assignment 4

Due: 4/22/16

Installing Erlang

http://www.erlang.org/download.html

Windows:

- Download and install the installer

OS X:

- Use brew or ports

Debian:

- sudo apt-get install erlang-base

Alternatively, build from source on OS X/Linux.

Erlang (since 198

Influenced by: Prolog, SmallTalk, PLEX

Telecom Industry

Getting Started Guide: erlang.org/download/getting_started-5.4.pdf

Erlang – Hello W

Erlang Shell: erl

```
1> io:fwrite("Hello, world!\n").
Hello World!
ok

2> halt().
```

Erlang

Main paradigms: concurrent, functional.

Motivation: fault-tolerant systems.

- Actor Model
- Some similarities to Prolog syntactically

Basic Commands

erl can be used as a basic calculator

```
> 1 + 2.
→ 3

>5 / 2.
→ 2.5

>5 div 2.
→ 2
```

Variables

Must begin with a capital letter.

```
>A = 1 + 2.
>B = 3 + 4.
>A = A - B.
```

Variables

Must begin with a capital letter.

```
>A = 1 + 2.
>B = 3 + 4.
>A = A – B.
```

```
** exception error: no match of right hand side
value -4
```

Key Points

Erlang uses single assignment

Assignment = pattern matching

```
>A=3.
>A=A.
5
```

Comparisons

== - Equal

/= - Not Equal

=< - Less than or equal to

< - Less than

>= - Greater than or equal to

> - Greater than

=:= - Exactly Equal (value and type)

=/= - Exactly not equal (value and type)

Operators

Unary operators: +, -

Arithmetic operators: +, -, *, /, div, rem

Bitwise operators: bnot, band, bor, bxor, bsl, bsr

Erlang - Modules

```
-module(test).

-export([double/1]).

double(X) ->
  X+X.
```

Erlang - Modules

In erl:

```
> c(test). % load module test
{ok, test}

> test:double(5). % function call
25
```

Erlang – Function

```
-module(fact).
-export([fact/1]).

fact(1) ->
  1;
fact(X) ->
  X*fact(X-1).
```

Functions

```
-module(fact).
-export([fact/1]).

fact(X) when X == 1 ->
  1;
fact(X) ->
  X*fact(X-1).
```

Erlang – Exercise

Write an Erlang module that computes the greatest common divisor according to Euclid's method.

if expression

Synatx:

```
if
    GuardSeq1 ->
        Body1;
    GuardSeqN ->
        BodyN
end
```

Example:

```
compare(X,Y) ->
    if
        X > Y -> 1;
        X < Y -> -1;
        true -> 0
    end.
```

Atoms

Data types that have no associated value, begin with a lowercase letter.

```
convert(X, inch) → % cm to inch
  X / 2.54;
convert(X, cm) → % inch to cm
  X * 2.54.
```

Tuples

Tuples group data.

Examples:
{2,3}
{captain, "James T. Kirk", "Enterprise"}

Exercise

Create a function that converts Centigrades to Fahrenheit and vice versa (c = (f-32)*5/9). The data should be tagged by the unit.

Pattern Matching

Does Left Side match Right Side?

- {Y, _} = {123, 51}. ?
- {atom, _} = {val, other}. ?
- {atom, _} = {atom, other}. ?

Pervasive throughout Erlang

- Assignment
- Function calls
- Recieve

Lists

Like tuples, but variable length:

```
> A = [1,2,3,4,5].
```

Accessing elements:

```
> [Head | Rest] = [1,2,3,4,5].
```

Ignore components

```
>[ _ | Rest] = [1,2,3,4,5].
```

List

Add elements:

```
> [1,2,3] ++ [4,5].
[1,2,3,4,5]
```

Remove Elements:

```
> [1,2,3] -- [1,5].
[2,3]
```

Exercise

Write a function *maxnum* that returns the maximum number in a list.

IO

Invoke IO methods

```
> io:format("Hello World!~n", []).
> io:format("Hello, ~w!~n", [joe]).

> io:format("Hello, ~w!~n", ["Joe"]).
  Hello, [74, 111, 101]!
```

Erlang FUNs

Fun ~ lambda function

```
F = fun (Arg1, Arg2, ... ArgN) ->
        ...
    end
```

```
F = fun Module:FunctionName/Arity
```

Exercise - lists

Write a function that doubles all elements in a list.

Note, lists can be represented by: [H|T]

Map

Map takes a Fun and applies it to all arguments in a list.

```
double(L)  -> map(fun(X) -> 2*X end, L).
```

Actor Model

Concurrency model that “treats ‘actors’ as the universal primitives of concurrent digital computation …

Actor = “object” + active behavior
Actors communicate through real messages

Actor Model

An actor can make local decisions”.

- send messages to other actors
- react upon next message
- make more actors

Advantages: No locking required

Concurrency

Concurrency - Having several actors working together or independently.
Parallelism - Having multiple actors running at the exact same time.

Erlang relies on spawning actors and passing messages to achieve these.

Fault Tolerance

Ability of a system to recover and continue processing when an error occurs.

Critical in Parallel and Distributed systems

Erlang - ‘Let it fail’

- Asynchronous
- Message Passing
- No assumptions about recipient

Message Passing

Commonly used paradigm in which data is sent between agents/processes.

- Send data/information
  let the recipient invoke the code
- Erlang ‘mailboxes’

Sending Messages

Syntax: PID ! message

Ex: self() ! hello.

-> hello

Empty mailbox:

-> flush().

Spawn

Creates a new process and returns PID.
(PID = process id)

```
spawn(Fun) -> pid()
spawn(Node, Fun) -> pid()
spawn(Module, Function, Args) -> pid()
spawn(Node, Module, Function, Args) -> pid()
```

Exercise

Write a function *run(X,Y)* that uses two actors to compute factorials for X and Y concurrently.

Factorial Revisited

```
-module(fact).
-export([f/1, run/0]).
f(0) ->
        io:format(“~p: ~B~n”, [self(), 1]),
        1;
f(N) ->
        io:format(“~p: ~B~n”, [self(), N],
        N * f(N-1).
run(X, Y) ->
        spawn(fact, f, [X]),
        spawn(fact, f, [Y]).
```

Communication

Spawn creates a new process (actor), but how can we communicate with it?

Communication

→ Send messages using process ID.

```
PID = spawn(fun ... end).

PID!hello.
PID!{self(), tag, data}.
```

Receive

Blocking call that waits for and processes messages.

```
receive
    pattern1 -> body1;
    pattern2 -> body2;
    patternN -> body3  % no ;
end.
```

Receive Example

```
-module(get).
-export([listen/0]).
listen() ->
    receive
        hello ->
            io:format(“Hello!~n”, []);
        goodbye ->
            io:format(“Goodbye!~n”, [])
    end.
```

Receive Example

What happens after we spawn and ..

Looping

Solution: Use recursion!

```
listen() ->
    receive
            hello ->
                io:format(“Hello!~n, []),
                listen();
            ...
```

Send and Respond

Use tuples and pattern matching, expect Pid as an argument:

```
listen() ->
    receive
        {PID, message, Data} ->
            PID ! ack;
        ...
```

Example

```
Pid = spawn(fake, listen, []).
Pid ! {self(), message, something}.
flush().
    Shell got ack.
    ok.
```

Exercise

Write a service that reads a message in the format of {PID, task, Data}, computes the tasks, and responds to PID with the result.

e.g., PID!{self(), fact, 5}.

Maintaining State

How can we keep track of the messages we receive?

Solution: Recursion

```
listen(MessageList) ->
    receive
        {Pid, message, Message} ->
            Pid ! ack,
            listen([Message|MessageList]);
        ...
    end.
```

Stateful Example

See reply.erl source

erl:

```
    Pid = spawn(reply, listen, [[]]).
    Pid ! {self(), message, hello}.
    Pid ! {self(), message, alright}.
    Pid ! {self(), get}.
    Pid ! {badinput}.
    flush().
        Shell got ack
        Shell got ack
        Shell got [alright,hello]
```

Other BIFs

register(Name, Pid)
unregister(Name)

Simple way of mapping/unmapping atom identifiers for Pids.

Distributed Actors

Involve communication across network.

Reference:

http://erlang.org/doc/reference_manual/distributed.html

Magic Cookie

Authentication of nodes.

Two Erlang nodes can communicate, if they have same magic cookie.

Magic Cookie

Option 1: Cookie is read at startup from ~/.erlang.cookie

Option 2: Cookie is set using:
erl -setcookie CookieString

Magic Cookie

Option 3: Cookie can be set for each node.

```
erlang:set_cookie(Node, CookieString)
```

Erlang Nodes

Names erlang instance:

Set at startup time:

```
erl -sname X@Host
erl -name Y@fully.qualified.name
```

Erlang Connection

a) Ping remote node

```
net_adm:ping('Y@host').
```

b) Spawn process on remote node

```
Spawn('Y@host', 't3', 'connect', [self()]).
```

Exercise

a) Create a listener that waits for a remote process to join, and then sends an acknowledgement back.

```
comm:listen() -> ...
```

b) Create a process that connects to a remote listener.

```
comm:connect(Node) -> ...
```

Erlang - Maps

Mapping of keys to values.

Reference:

http://joearms.github.io/2014/02/01/big-changes-to-erlang.html

Erlang - Maps

Create a map

```
Map = #{key1 => Val1, key2 => Val2, ...}.

Z = #{ {age, fred} => 12,
       {age, bill} => 97,
       {color, red} => {rgb, 255, 0, 0}}.
```

Erlang - Maps

Retrieve data from map:

```
Map = #{key1 => Val1, key2 => Val2, ...}.

#{ {color, red} := X1} = Z.
% X1 = {rgb, 255, 0, 0}
```

Erlang - Maps

Update map:

```
Map1 = Map#{key1 := Val1, key2 := Val2, ...}.

Z1 = Z#{{color, red} := {rgb, 0, 0, 255}}.
% red is blue in Z1
```

Erlang - Maps

Update map:

```
Map1 = Map#{key1 => Val1, key2 => Val2, ...}.
Map1 = Map#{key1 := Val1, key2 := Val2, ...}.
```

=> updates or inserts

:= updates (key must be present)

References

http://www.erlang.org/download/getting_started-5.4.pdf

http://www.tryerlang.org/
http://learnyousomeerlang.com/
http://www.erlang.org/doc/getting_started/conc_prog.html
http://savanne.be/articles/concurrency-in-erlang-scala/

References

http://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency#thanks-for-all-the-fish

http://www.erlang.org/course/concurrent_programming.html

http://www.erlang.org/download/erlang-book-part1.pdf