Introduction to Erlang

Sébastien Tixeuil
Sebastien.Tixeuil@lip6.fr

Erlang History

- **Erlang**: Ericsson Language
- Designed to implement large-scale real-time telecommunication switching systems
- Open source version
- http://www.erlang.org

Main Characteristics

- Declarative
- Concurrent
- Real-time
- Continuous operation
- Robust
- Memory management
- Distribution
- Integration

Basics

- '%' starts a comment
- '.' ends a declaration
- Every function must be in a module
  - source file name is module name + ".erl"
- ':' used for calling functions in other modules

Basics

```
-module(tut).
-export([double/1]).

double(X) ->
    2 * X.
```

The Erlang Shell

```
% erl
Erlang (BEAM) emulator version 5.2 [source] [hipe]

Eshell V5.2  (abort with ^G)
1>
```

```
1> 2 + 5.
7
2> (42 + 77) * 66 / 3.
2618.00
```

io:format

```
32> io:format("hello world~n", []).
hello world
ok
33> io:format("this outputs one Erlang term: ~w~n", [hello]).
this outputs one Erlang term: hello
ok
34> io:format("this outputs two Erlang terms: ~w~w~n", [hello, world]).
this outputs two Erlang terms: helloworld
ok
35> io:format("this outputs two Erlang terms: ~w ~w~n", [hello, world]).
this outputs two Erlang terms: hello world
ok
```

Manual

```
% erl -man io
ERLANG MODULE DEFINITION                                        io(3)

MODULE
     io - Standard I/O Server Interface Functions

DESCRIPTION
     This module provides an  interface  to  standard  Erlang  IO
     servers. The output functions all return ok if they are suc-
     ...
```

Pattern Matching

- The expression "`Pattern = Expression`" causes "`Expression`" to be evaluated and the result matched against "`Pattern`"
- If the match succeeds, "`Pattern`" is then *bound*

Pattern Matching

```
{A, B} = {12, apple}

{C, [Head|Tail]} = {{222, man}, [a,b,c]}

[{person, Name, Age, _}|T] =
        [{person, fred, 22, male},
         {person, susan, 19, female}, ...]
```

Recursive Functions

- *Variables* start with upper-case characters
- '*;*' separates function clauses
- Variables are *local* to the function clause
- Pattern matching and guards to select clauses
- Runtime error if no clause matches

Recursive Functions

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

fac(N) when N > 0 ->  N * fac(N-1);
fac(0)            ->  1.

> math:fac(25).
15511210043330985984000000
```

Lists

- Pattern-matching selects components of the data
- '_' is a "don't care" pattern (not a variable)
- '[]' is the empty list
- '[X,Y,Z]' is a list with exactly three elements
- '[X,Y,Z|Tail]' has three or more elements

Lists

```
18> [First |TheRest] = [1,2,3,4,5].
[1,2,3,4,5]
19> First.
1
20> TheRest.
[2,3,4,5]
```

Lists

```
21> [E1, E2 | R] = [1,2,3,4,5,6,7].
[1,2,3,4,5,6,7]
22> E1.
1
23> E2.
2
24> R.
[3,4,5,6,7]
```

Lists

```
25> [A, B | C] = [1, 2].
[1,2]
26> A.
1
27> B.
2
28> C.
[]
```

Lists

```
append([H|T], L) -> [H|append(T, L)];
append([],    L) -> L.
```

```
list_length([]) ->
    0;
list_length([First | Rest]) ->
    1 + list_length(Rest).
```

List Comprehension

- Left of the '||' is an expression template
- 'Pattern <- List' is a generator
  - elements are picked from the list in order
- The other expressions are boolean filters
- If there are multiple generators, you get all combinations of values

Lists

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

Numbers

- Arbitrary-size integers (but usually just one word)
- #-notation for base-N integers
- $-notation for character codes (ISO-8859-1)
- Floating-point numbers
  - == *vs.* =:= and /= *vs.* =/=

Arithmetic Expressions

Operator	Description	Type	Operands	Prio
+ X	+ X	unary	mixed	1
- X	- X	unary	mixed	1
X * Y	X * Y	binary	mixed	2
X / Y	X / Y (floating point division)	binary	mixed	2
X div Y	integer division of X and Y	binary	integer	2
X rem Y	integer remainder of X divided by Y	binary	integer	2
X band Y	bitwise and of X and Y	binary	integer	2
X + Y	X + Y	binary	mixed	3
X - Y	X - Y	binary	mixed	3
X bor Y	bitwise or of X and Y	binary	integer	3
X bxor Y	arithmetic bitwise xor X and Y	binary	integer	3
X bsl N	arithmetic bitshift left of X by N bits	binary	integer	3
X bsr N	bitshift right of X by N bits	binary	integer	3

Atoms

- Must start with lower case character or be quoted
- Single-quotes are used to create arbitrary atoms
- Similar to hashed strings
  - use only one word of data
  - constant-time equality test

Quoted Atoms

Characters	Meaning
\b	backspace
\d	delete
\e	escape
\f	form feed
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\\	backslash
\^A .. \^Z	control A to control Z (i.e. 0 .. 26)
\'	single quote
\"	double quote
\OOO	The character with octal representation OOO

Atoms

```
-module(tut2).
-export([convert/2]).

convert(M, inch) ->
    M / 2.54;

convert(N, centimeter) ->
    N * 2.54.
```

```
10> tut2:convert(3, inch).
1.18110
11> tut2:convert(7, centimeter).
17.7800
```

Comparisons

Operator	Description	Type
`X > Y`	`X` greater than `Y`	coerce
`X < Y`	`X` less than `Y`	coerce
`X =< Y`	`X` equal to or less than `Y`	coerce
`X >= Y`	`X` greater than or equal to `Y`	coerce
`X == Y`	`X` equal to `Y`	coerce
`X /= Y`	`X` not equal to `Y`	coerce
`X =:= Y`	`X` equal to `Y`	exact
`X =/= Y`	`X` not equal to `Y`	exact

Tuples

- Tuples are the main data constructor in Erlang
- A tuple whose first element is an atom is called a tagged tuple
- The elements of a tuple can be any values

Tuples

```
-module(tut3).
-export([convert_length/1]).

convert_length({centimeter, X}) ->
    {inch, X / 2.54};
convert_length({inch, Y}) ->
    {centimeter, Y * 2.54}.
```

```
15> tut3:convert_length({inch, 5}).
{centimeter,12.7000}
16> tut3:convert_length(tut3:convert_length({inch, 5})).
{inch,5.00000}
```

Tuples

```
lookup(Key, {Key, Val, _, _}) ->
  {ok, Val};
lookup(Key,{Key1,Val,S,B}) when Key<Key1->
  lookup(Key, S);
lookup(Key, {Key1,Val,S,B}) ->
  lookup(Key, B);
lookup(Key, nil) ->
  not_found.
```

Record Syntax

- Records are just a syntax for working with tagged tuples
- You don't have to remember element order and tuple size
- Good for internal work within a module
- Not so good in public interfaces (users must have same definition)

Built-in Functions

- Implemented in C
- All the type tests and conversions are BIFs
- Most BIFs (not all) are in the module "`erlang`"
- Many common BIFs are auto-imported (recognized without writing "`erlang:...`")
- Operators ('+','-','*','/',...) are also really BIFs

Fun Expressions

- Anonymous functions (lambda expressions)
- Can have several clauses
- All variables in the pattern are new
  - All variable bindings in the fun are local
- Variables bound in the environment can be used in the fun-body

Fun Expressions

```
90> Xf = fun(X) -> X * 2 end.
#Fun<erl_eval.5.123085357>
91> Xf(5).
10
```

Fun Expressions

```
map(Fun, [First|Rest]) ->
    [Fun(First)|map(Fun,Rest)];
       map(Fun, []) ->
    [].
```

```
92> Add_3 = fun(X) -> X + 3 end.
#Fun<erl_eval.5.123085357>
93> lists:map(Add_3, [1,2,3]).
[4,5,6]
```

Case-switches

- Choice between alternatives within the body of a clause

```
case Expr of
    Pattern1 [when Guard1]  -> Seq1;
    Pattern2 [when Guard2]  -> Seq2;
    ...
    PatternN [when GuardN] -> SeqN
end
```

If-switches

- Like a case-switch without the patterns and the '`when`' keyword
- Use '`true`' as catch-all
- Guards are special
  - comma-separated list
- only specific built-in functions (and all operators)
- no side effects

If-switches

```
-module(tut9).
-export([test_if/2]).

test_if(A, B) ->
    if
        A == 5 ->
            io:format("A = 5~n", []),
            a_equals_5;
        B == 6 ->
            io:format("B = 6~n", []),
            b_equals_6;
        A == 2, B == 3 ->                    %i.e. A equals 2 and B equals 3
            io:format("A == 2, B == 3~n", []),
            a_equals_2_b_equals_3;
        A == 1 ; B == 7 ->                   %i.e. A equals 1 or B equals 7
            io:format("A == 1 ; B == 7~n", []),
            a_equals_1_or_b_equals_7
    end.
```

Catching Exceptions

- Three classes of exceptions
  - `throw`: user-defined
  - `error`: runtime errors
  - `exit`: end process
- only catch `throw` exceptions normally
- Re-thrown if no catch-clause matches
- '`after`' part is always run (side effects only)

Preprocessor

- C-style token-level preprocessor
- Record definitions often put in header files, to be included
- Use macros mainly for constants
- Use functions instead of macros if you can (compiler can inline)

Processes

- Code is executed by a *process*
- A process keeps track of the program pointer, the stack, the variables values, etc.
- Every process has a *unique* process *identifier*
- Processes are *concurrent*
- Processes do *not* share data

Processes: Implementation

- Virtual machine layer processes
- Preemptive multitasking
- Little overhead (*e.g.* 100.000 processes)
- Can use multiple CPUs on multiprocessor machines

Starting Processes

- The "`spawn`" function creates a new process
- The new process will run the specified function
- The spawn operation always returns immediately
  - The return value is the `Pid` of the "child"

Concurrency

- Several processes may use the same program code at the same time
  - each has own program counter, stack, and variables
  - programmer need not think about other processes updating the variables

Processes

```
-module(tut14).

-export([start/1, say_something/2]).

say_something(What, 0) ->
    done;
say_something(What, Times) ->
    io:format("~p~n", [What]),
    say_something(What, Times - 1).

start() ->
    spawn(tut14, say_something, [hello, 3]),
    spawn(tut14, say_something, [goodbye, 3]).
```

```
9> tut14:start().
hello
goodbye
<0.63.0>
hello
goodbye
hello
goodbye
```

Message Passing

```
Pid ! Message
```

- "!" is the *send operator*
  - *Pid* of the receiver is used as the address
- Messages are sent *asynchronously*
- Any value can be sent as a message

Message Queues

- Each process has a *message queue* (mailbox)
  - incoming messages are placed in the queue (no size limit)
- A process *receives* a message when it extracts it from the mailbox
  - need *not* take the first message in the queue

Receiving a Message

- `receive`-expressions are similar to `case` switches
  - patterns are used to match messages in the mailbox
  - messages in the queue are tested in order
  - only one message can be extracted each time

Receiving a Message

```
receive
  Message1 ->
    Actions1;
  Message2 ->
    Actions2;
    ...
after Time ->
  TimeOutActions
end
```

Selective Receive

- Patterns and guards permit message selection
- `receive`-clauses are tried in order
- If *no* message matches, the process *suspends* and waits for a new message

Send and Reply

- `Pid`s are often included in messages (`self()`), so that the receiver can reply to the sender
  - If the reply includes the `Pid` of the second process, it is easier for the first process to recognize the reply

Send and Reply

```
Call (RPC)
  A ! {self(), B},
  receive
    {A, Reply} ->
      Reply
  end
```

Registered Processes

- A process can be registered under a name
- Any process can send a message to a registered process, or look up the `Pid`
- The `Pid` might change (if the process is restarted and re-registered), but the name stays the same

Ping-pong

```
-module(tut16).

-export([start/0, ping/1, pong/0]).
```

```
start() ->
    register(pong, spawn(tut16, pong, [])),
    spawn(tut16, ping, [3]).
```

Ping-pong

```
pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.
```

Ping-pong

```
ping(0) ->
    pong ! finished,
    io:format("ping finished~n", []);

ping(N) ->
    pong ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1).
```

Receive with Timeout

- A receive-expression can have an `after`-part
  - can be an integer (milliseconds) or "`infinity`"
- The process waits until a matching message arrives, or the timeout limit is exceeded
  - soft real-time: **no** guarantees

Receive with Timeout

```
receive
    Message1 [when Guard1] ->
        Actions1 ;
    Message2 [when Guard2] ->
        Actions2 ;
    ...
after
    TimeOutExpr ->
        ActionsT
end
```

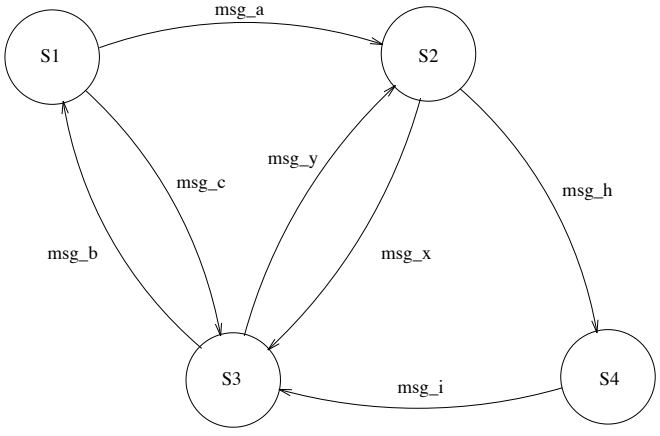
Message Order

- The **only** guaranteed message order is when both the sender and the receiver are the same for both messages (first-in, first-out)
- Using selective receive, it is possible to choose which messages to accept, even if they arrive in a different order

Process Termination

- A process *terminates* when:
  - it finishes the function call that it started with
  - there is an exception that is not caught
- All messages sent to a terminated process will be thrown away
- Same `Pid` will not be used before long

Implementing Automata



Implementing Automata

```
s1() ->
    receive
        msg_a ->
            s2();
        msg_c ->
            s3()
    end.

s2() ->
    receive
        msg_x ->
            s3();
        msg_h ->
            s4()
    end.
```

```
s3() ->
    receive
        msg_b ->
            s1();
        msg_y ->
            s2()
    end.

s4() ->
    receive
        msg_i ->
            s3()
    end.
```

Distribution

- Running "`erl`" with the flag "`-name xxx`"
  - starts the Erlang network distribution system
  - makes the virtual machine emulator a "node" ('`xxx@host.domain`')
- Erlang nodes can communicate over the network (but must find each other first)

Connecting Nodes

- Nodes are connected the first time they try to communicate
- The function "`net_adm:ping(Node)`" is the easiest way to set up a connection between nodes
  - returns "`pong`" or "`pang`"
- Send a message to a registered process using "`{Name,Node} ! Message`"

Distribution is Transparent

- Possible to send a `Pid` from one node to another (`Pid`s are unique across nodes)
- You can send a message to any process through its `Pid` (even on another node)
- You can run several Erlang nodes (with different names) on the same computer

Running Remote Processes

- Variants of the spawn function can start processes directly on another node
- The module '`global`' contains functions for
  - *registering* and *using* named processes over the whole network of connected nodes
  - setting global locks

Bit Syntax

```
-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN,5).

DgramSize = size(Dgram),
case Dgram of
 <<?IP_VERSION:4, HLen:4,
  SrvcType:8,TotLen:16,ID:16,Flgs:3,
  FragOff:13,TTL:8,Proto:8,HdrChkSum:16,
  SrcIP:32,DestIP:32,Body/binary>> when
   HLen >= 5,4*HLen =< DgramSize ->
    OptsLen = 4*(HLen-?IP_MIN_HDR_LEN),
    <<Opts:OptsLen/binary,Data/binary>>
    = Body,
  ...
end.
```

Token Passing Example

- Unidirectional ring of `N` nodes
- Each node `i` has a state `v(i)` (integer)
- Top :
  - `<v(n-1) = v(0)> -> v(0):=v(0)+1 mod N`
- Bottom :
  - `<v(i-1) != v(i)> -> v(i):=v(i-1)`

Token Ring Example

```
bottom( Child, State ) ->
     io:format("> Bottom: ~w, Seq#: ~w, State#: ~w ..~n", [self(), Child, State]),
     receive
          N when N =/= State ->
               Child ! N,
               bottom( Child, N );
          N when N =:= State ->
               Child ! State,
               bottom( Child, State)
     end.
```

Token Ring Example

```
top( Child, State, Size ) ->
     io:format("> Top: ~w, Seq#: ~w, State#: ~w ..~n", [self(), Child, State]),
     receive
          N when N =:= State ->
               Child ! ((N + 1) rem Size) ,
               top( Child, ((N + 1) rem Size), Size );
          N when N =/= State ->
               Child ! State,
               top( Child, State, Size )
     after 2000 ->
               Child ! State,
               top( Child, State, Size )
     end.
```

Token Ring Example

```
-module(dijkstra).
-export([init/1]).

init( N ) when N > 1 ->
      Top = spawn( fun() -> wait() end ),
      Top ! { init( N - 1, Top, N ), random:uniform( N ), N }.

init( N, Last, Size ) when N > 0 ->
      spawn( fun() -> bottom( init( N - 1, Last, Size ), random:uniform( N ) ) end );
init( N, Last, _Size ) when N =:= 0 ->
      Last.

wait() ->
      receive { Child, State, Size } ->
           top( Child, State, Size )
      end.
```