# Erlang Cheat Sheet

## Basics

### Numbers

```
> 2 + 15.
17
> 49 * 100.
4900
> 1892 - 1472.
420
> 5 / 2.
2.5
> 5 div 2.
2
> 5 rem 2.
1
```

### Invariable Variables

They start with an upper case letter and can only be bound once.

```
One = 1.
```

### Atoms

They are like symbols. Unquoted must be lowercase to avoid clashes with variables.

```
> atom.
atom
> atoms_rule.
atoms_rule
> atoms_rule@erlang.
atoms_rule@erlang
> 'Atoms can be cheated!'.
'Atoms can be cheated!'
> atom = 'atom'.
atom
```

### Boolean Algebra & Comparison operators

```
> true and false.
false
> false or true.
true
> true xor false.
true
> not false.
true
> not (true and true).
false

> 5 =:= 5.
true
> 1 =:= 0.
false
> 1 =/= 0.
true
> 5 =:= 5.0.
false
> 5 == 5.0.
true
> 5 /= 5.0.
false

> 1 < 2.
true
> 1 < 1.
false
> 1 >= 1.
true
> 1 =< 1. % !!!
true
```

## Tuples

Tuples are used to store a fixed number of items.

```
{123, bcd}
{123, def, abc}
{person, 'Jim', 'Austrian'} % three atoms!
{abc, {def, 123}, jkl}
```

## Lists

```
AList = [1, 2, 3, {numbers, [4,5,6]}, 5.34, atom].
```

### ++ and -- operators

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

Both `++` and `--` are right-associative. This means the elements of many `--` or `++` operations will be done from right to left, as in the following examples:

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

### hd and tl

```
> hd([1,2,3,4]).
1
> tl([1,2,3,4]).
[2,3,4]
```

### Cons operator

```
> List = [2,3,4].
[2,3,4]
> NewList = [1|List].
[1,2,3,4]
> [Head|Tail] = NewList.
[1,2,3,4]
> Head.
1
> Tail.
[2,3,4]
> [NewHead|NewTail] = Tail.
[2,3,4]
> NewHead.
2

% any list can be built with
% only cons and values
> [1 | []].
[1]
> [2 | [1 | []]].
[2,1]
> [3 | [2 | [1 | []] ] ].
[3,2,1]
```

### List Comprehensions

```
> [2*N || N <- [1,2,3,4]].
[2,4,6,8]

> [X || X <- [1,2,3,4,5,6,7,8,9,10], X rem 2 =:= 0].
[2,4,6,8,10]

> [X+Y || X <- [1,2], Y <- [2,3]].
[3,4,4,5]
```

## Pattern Matching

= is for pattern matching; _ is "don't care".

```
A = 10
    Succeeds - binds A to 10
{A, A, B} = {abc, abc, foo}
    Succeeds - binds A to abc, B to foo
{A, A, B} = {abc, def, 123}
    Fails
[A,B|C] = [1,2,3,4,5,6,7]
    Succeeds - binds A = 1, B = 2, C = [3,4,5,6,7]
[H|T] = [abc]
    Succeeds - binds H = abc, T = []
{A,_, [B|_], {B}} = {abc, 23, [22, x], {22}}
    Succeeds - binds A = abc, B = 22
```

## Maps

### Definition

```
> Map = #{one => 1, "Two" => 2, 3 => three}.
#{3 => three,one => 1,"Two" => 2}
```

### Update / Insert

```
> Map#{one := "I"}.
#{3 => three,one => "I","Two" => 2}
```

```
> Map.
#{3 => three,one => 1,"Two" => 2} % unchanged
```

### Get value

```
> #{"Two" := V} = Map.
#{3 => three,one => 1,"Two" => 2}
> V.
2
```

## Built In Functions

```
date()
time()
length([1,2,3,4,5])
size({a,b,c})
atom_to_list(an_atom) % "an_atom"
list_to_tuple([1,2,3,4]) % {1,2,3,4}
integer_to_list(2234) % "2234"
tuple_to_list({}) ...
```

## Function Syntax & Evaluation

A function is defined as a sequence of clauses.

```
func(Pattern1, Pattern2, ...) -> ... ;
func(Pattern1, Pattern2, ...) -> ... ;
...
func(Pattern1, Pattern2, ...) -> ... .
```

Clauses are scanned sequentially until a match is found.
When a match is found all variables occurring in the head become bound.
Variables are local to each clause, and are allocated and deallocated automatically.
The body is evaluated sequentially (use "," as separator).

```
-module(mathStuff).
-export([factorial/1, area/1]).
factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).
area({square, Side}) ->
Side * Side;
area({circle, Radius}) ->
3.14 * Radius * Radius;
area({triangle, A, B, C}) ->
S = (A + B + C)/2,
math:sqrt(S*(S-A)*(S-B)*(S-C));
area(Other) ->
{invalid_object, Other}.
```

## Guards

The keyword `when` introduces a guard.

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

### Examples of Guards

```
number(X)       % X is a number
integer(X)      % X is an integer
float(X)        % X is a float
atom(X)         % X is an atom
tuple(X)        % X is a tuple
list(X)         % X is a list
X > Y + Z       % X is > Y + Z
X =:= Y         % X is exactly equal to Y
X =/= Y         % X is not exactly equal to Y
X == Y          % X is equal to Y
    % (with int coerced to floats,
    % i.e. 1 == 1.0 succeeds
    % but 1 =:= 1.0 fails)
length(X) =:= 3 % X is a list of length 3
size(X) =:= 2   % X is a tuple of size 2.
```

### Notice

There is a guard sub-language, because guards must be evaluated in constant time. **Therefore, you cannot use your own predicates in them**.

## Apply

```
apply(Mod, Func, Args)
```

Apply function `Func` in module `Mod` to the arguments in the list `Args`.
Mod and Func must be atoms (or expressions which evaluate to atoms).
Any Erlang expression can be used in the arguments to apply.
`?MODULE` uses the preprocessor to get the current module's name.

```
%% Example
apply(?MODULE, min_max, [[4,1,7,3,9,10]]).
{1, 10}
```

## Case

```
%% Simple example
case lists:member(a, X) of
    true -> ... ;
    false -> ...
end,

%% Complex example
beach(Temperature) ->
    case Temperature of
    {celsius, N} when N >= 20, N =< 45 ->
        'favorable';
    {kelvin, N} when N >= 293, N =< 318 ->
        'scientifically favorable';
    {fahrenheit, N} when N >= 68, N =< 113 ->
        'favorable in the US';
    _ ->
        'avoid beach'
end.
```

## If

```
if
    integer(X) -> ... ;
    tuple(X) -> ... ;
    true -> ... % works as an else
end,
```

### If vs Case

`if` needs **guards**, so **for user defined predicates it is customary to use** `case`.

## Recursion

Let's implement some functions recursively.

### Factorial (again)

```
%% Tail recursive version
tail_fac(N) -> tail_fac(N,1).

tail_fac(0,Acc) -> Acc;
tail_fac(N,Acc) when N > 0 -> tail_fac(N-1,N*Acc).
```

### Length

```
%% Normal version
len([]) -> 0;
len([_|T]) -> 1 + len(T).

%% Tail recursive version
tail_len(L) -> tail_len(L,0).

tail_len([], Acc) -> Acc;
tail_len([_|T], Acc) -> tail_len(T,Acc+1).
```

### Duplicate

```
%% Normal version
duplicate(0,_) ->
[];
duplicate(N,Term) when N > 0 ->
[Term|duplicate(N-1,Term)].

%% Tail recursive version
tail_duplicate(N,Term) ->
    tail_duplicate(N,Term,[]).

tail_duplicate(0,_,List) ->
    List;
tail_duplicate(N,Term,List) when N > 0 ->
    tail_duplicate(N-1, Term, [Term|List]).
```

### Reverse

```
%% Normal version
reverse([]) -> [];
reverse([H|T]) -> reverse(T)++[H].

%% Tail recursive version
tail_reverse(L) -> tail_reverse(L,[]).

tail_reverse([],Acc) -> Acc;
tail_reverse([H|T],Acc) -> tail_reverse(T, [H|Acc]).
```

### Sublist

```
%% Normal version
sublist(_,0) -> [];
sublist([],_) -> [];
sublist([H|T],N) when N > 0 -> [H|sublist(T,N-1)].

%% Tail recursive version
tail_sublist(L, N) -> tail_sublist(L, N, []).

tail_sublist(_, 0, SubList) -> SubList;
tail_sublist([], _, SubList) -> SubList;
tail_sublist([H|T], N, SubList) when N > 0 ->
tail_sublist(T, N-1, [H|SubList]).
    % We use a list as an accumulator
    % like we did to reverse our list.
    % So we have to reverse it.
tail_sublist(L, N) -> reverse(tail_sublist(L, N, [])).
```

### Zip

```
%% Lists with same length
zip([],[]) -> [];
zip([X|Xs],[Y|Ys]) -> [{X,Y}|zip(Xs,Ys)].

%% Lists with different lengths
lenient_zip([],_) -> [];
lenient_zip(_,[]) -> [];
lenient_zip([X|Xs],[Y|Ys]) ->
    [{X,Y}|lenient_zip(Xs,Ys)].
```

### Quicksort

```
%% Simple version
% easier to read, but has to traverse
% the list twice to partition it in two parts
lc_quicksort([]) -> [];
lc_quicksort([Pivot|Rest]) ->
lc_quicksort([Smaller || Smaller <- Rest,
    Smaller =< Pivot])
++ [Pivot] ++
lc_quicksort([Larger || Larger <- Rest,
    Larger > Pivot]).

%% Two-function version
quicksort([]) -> [];
quicksort([Pivot|Rest]) ->
    {Smaller, Larger} =
        partition(Pivot,Rest,[],[]),
    quicksort(Smaller) ++ [Pivot] ++
        quicksort(Larger).

partition(_,[], Smaller, Larger) ->
    {Smaller, Larger};
partition(Pivot, [H|T], Smaller, Larger) ->
if H =< Pivot ->
    partition(Pivot, T, [H|Smaller], Larger);
H > Pivot ->
    partition(Pivot, T, Smaller, [H|Larger])
end.
```

### Mergesort

```
mergesort([L]) -> [L];
mergesort(L) ->
    {L1, L2} = lists:split(length(L) div 2, L),
    merge(mergesort(L1), mergesort(L2)).

merge(L1, L2) -> merge(L1, L2, []).
merge([], L2, A) -> A ++ L2;
merge(L1, [], A) -> A ++ L1;
merge([H1|T1], [H2|T2], A) when H2 >= H1 ->
    merge(T1, [H2|T2], A ++ [H1]);
merge([H1|T1], [H2|T2], A) when H1 > H2 ->
    merge([H1|T1], T2, A ++ [H2]).
```

## Lambdas

Syntax for lambdas is, e.g.:

```
Square = fun (X) -> X*X end.
```

We can use it like this:

```
Square(3).
```

Lambdas can be passed as usual to higher order functions.

## Some HOF implementations

```
%% Map
map(_, []) -> [];
map(F, [H|T]) -> [F(H)|map(F,T)].

%% Filter
filter(Pred, L) ->
lists:reverse(filter(Pred, L, [])).

filter(_, [], Acc) -> Acc;
filter(Pred, [H|T], Acc) ->
case Pred(H) of
true  -> filter(Pred, T, [H|Acc]);
false -> filter(Pred, T, Acc)
end.

%% Find the maximum of a list
max([H|T]) -> max2(T, H).

max2([], Max) -> Max;
max2([H|T], Max) when H > Max -> max2(T, H);
max2([_|T], Max) -> max2(T, Max).

%% Find the minimum of a list
min([H|T]) -> min2(T,H).

min2([], Min) -> Min;
min2([H|T], Min) when H < Min -> min2(T,H);
min2([_|T], Min) -> min2(T, Min).

%% Sum of all the elements of a list
sum(L) -> sum(L,0).

sum([], Sum) -> Sum;
sum([H|T], Sum) -> sum(T, H+Sum).

%% Left fold
fold(_, Start, []) -> Start;
fold(F, Start, [H|T]) -> fold(F, F(H,Start), T).
```

## STDLIB: lists

### map/2

```
lists:map(Fun, List1) -> List2
```

### filter/2

```
lists:filter(Pred, List1) -> List2
```

### foldl/3

```
lists:foldl(Fun, Acc0, List) -> Acc1
```

### foldr/3

```
lists:foldr(Fun, Acc0, List) -> Acc1
```

### zip/2

```
lists:zip(List1, List2) -> List3
```

### append/1

Returns a list in which all the sublists of `ListOfLists` have been appended.

```
lists:append(ListOfLists) -> List1
```

### partition/2

It takes a list and returns two: one that has the terms which satisfy a given predicate, and one list for the others.

```
lists:partition(Pred, List) ->
    {Satisfying, NotSatisfying}
```

### all/2

It takes a predicate and a list and tests if all its elements return true.

```
lists:all(Pred, List) -> boolean()
```

### any/2

It takes a predicate and a list and tests if at least one of its elements returns true.

```
lists:any(Pred, List) -> boolean()
```

### flatten/1

It returns a flattened version of `DeepList`.

```
lists:flatten(DeepList) -> List
```

### split/2

It takes a list and splits it into two list. The first contains the first N elements and the second the remaining ones.

```
lists:split(N, List1) -> {List2, List3}
```

### nth/2

It returns the nth element of a list.

```
lists:nth(N, List) -> Elem
```

### seq/2

It returns a sequence of integers from `From` to `To`.

```
lists:seq(From, To) -> Seq
```

### seq/3

It returns a sequence of integers from `From` to `To`, increasing by `Incr`. If `To` is passed, then it is not an element of the sequence.

```
lists:seq(From, To, Incr) -> Seq
```

### sublist/2

Returns the sublist of `List1` starting at position 1 and with (maximum) `Len` elements. It is not an error for Len to exceed the length of the list, in that case the whole list is returned.

```
lists:sublist(List1, Len) -> List2
```

### sublist/3

Returns the sublist of `List1` starting at `Start` and with (maximum) `Len` elements. It is not an error for `Start`+`Len` to exceed the length of the list.

```
lists:sublist(List1, Start, Len) -> List2
```

### keystore/4

Returns a copy of `TupleList1` where the first occurrence of a tuple `T` whose `N`th element compares equal to `Key` is replaced with `NewTuple`, if there is such a tuple `T`. If there is no such tuple `T`, a copy of `TupleList1` where [`NewTuple`] has been appended to the end is returned.

```
lists:keystore(Key, N, TupleList1, NewTuple) ->
    TupleList2
```

### keyfind/3

Searches the list of tuples `TupleList` for a tuple whose `N`th element compares equal to `Key`. Returns `Tuple` if such a tuple is found, otherwise `false`.

```
lists:keyfind(Key, N, TupleList) -> Tuple | false
```

## IO Format specifiers

1. `~n`: newline
2. `~c`: ASCII code (characters)
3. `~f`: float
4. `~s`: string
5. `~w`: Erlang terms (e.g., atoms)

## Concurrent programming

### Primitives

There are three main primitives:

1. `spawn`, that creates a new process executing the specified function, returning an identifier.
2. `!`, that sends a message to a process through its identifier; the content of the message is simply a variable; the operation is asynchronous.
3. `receive ... end`, that extracts, going from the first, a message from a process's mailbox queue matching with the provided set of patterns (this is blocking if no message is in the mailbox); the mailbox is persistent until the process quits.

### Creating a New Process

Let's assume we are in the process with identifier `Pid1`. In it we perform `Pid2 = spawn(Mod, Func, Args)`, where: `Mod` is the current module (obtainable with the `?MODULE` macro; `Func` is the function the new process will execute; `Args` is the list of arguments (e.g.: `[Arg1, Arg2, Arg3]`). After, `Pid2` is bound to the identifier of the new process (known only to process `Pid1`).

#### Alternative spawn compact notation

```
Pid = spawn(fun() -> myFun(arg1, arg2, ...) end)
```

### Message Passing

1. Process $A$ sends a message to $B$ (it uses `self()` to identify itself):

   ```
   PidB ! {self(), {mymessage, [1,2,3]}}
   ```

   `self()` returns the identifier of the process executing it. Messages can carry data and be selectively unpacked.

2. `B` receives it with:

   ```
   receive
       {A, {mymessage, D}} ->
           work_on_data(D);
   end
   ```

   Variables `A` and `D` become bound when receiving the message (if `A` is bound before receiving a message, then only data from that process is accepted).

### Example

```
%% An Echo process
-module(echo).
-export([go/0, loop/0]).

go() ->
    %% spawn(Mod, Func, Args)
    %% the current module can be obtained
    %% with the ?MODULE macro
    Pid2 = spawn(echo, loop, []),
    Pid2 ! {self(), hello},
    receive
        {Pid2, Msg} ->
            io:format("P1 ~w~n",[Msg])
    end,
    Pid2 ! stop.

loop() ->
    receive
        {From, Msg} ->
            From ! {self(), Msg},
            loop();
        stop ->
            true
    end.
```

### Selective Message Reception

1. $A$ performs `PidC ! foo`.
2. $B$ performs PidC ! bar.
3. Code in $C$:

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

4. `foo` is received, then `bar`, irrespective of the order in which they were sent.

### Selection of any message

1. $A$ performs `PidC ! foo`.
2. $B$ performs PidC ! bar.
3. Code in $C$:

   ```
   receive
       Msg -> ... ;
   end
   ```

4. The first message to arrive at the process $C$ will be processed; the variable `Msg` in the process $C$ will be bound to one of the atoms `foo` or `bar` depending on which arrives first.

### Registered Processes

```
register(Alias, Pid)
```

Registers the process `Pid` with name `Alias`. E.g.:

```
start() ->
    Pid = spawn(?MODULE, server, [])
    register(analyzer, Pid).
analyze(Seq) ->
    analyzer ! {self(), {analyze, Seq}},
    receive
        {analysis_result, R} -> R
    end.
```

Any process can send a message to a registered process.

## Client Server Model

Client-Server can be easily realized through a simple protocol, where requests have the syntax {`request, ...`}, while replies are written as {`reply, ...`}.
Server code:

```
-module(myserver).
server(Data) -> % note: local data
    receive
        {From,{request,X}} ->
            {R, Data1} = fn(X, Data),
            From ! {myserver,{reply, R}},
            server(Data1)
    end.
```

Interface Library:

```
-export([request/1]).
request(Req) ->
    myserver ! {self(),{request,Req}},
    receive
        {myserver,{reply,Rep}} -> Rep
    end.
```

## Timeouts

Consider this code in process $B$:

```
receive
    foo -> Actions1;
after
    Time -> Actions2;
```

If the message `foo` is received from $A$ within the time `Time` perform `Actions1`. Otherwise perform `Actions2`.

### Uses of Timeouts

- `sleep(T)`: process suspends for `T` ms.

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

- `suspend()`: process suspends indefinitely.

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

- The message `What` is sent to the current process in `T` ms from now.

```
set_alarm(T, What) ->
    spawn(timer, set, [self(), T, What]).

set(Pid, T, Alarm) ->
    receive
    after
        T -> Pid ! Alarm
    end.
receive
    Msg -> ... ;
end
```

- `flush()`: flushes the message buffer

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

A value of 0 in the timeout means check the message buffer first and if it is empty execute the following code.

## "Let it crash": an example

We are going to see a simple supervisor **linked** to a number of workers.
Each worker has a state (a natural number, 0 at start), can receive messages with a number to add to it from the supervisor, and sends back its current state. When its local value exceeds 30, a worker ends its activity.
The supervisor sends "add" messages to workers, and keeps track of how many of them are still active; when the last one ends, it terminates.
We are going to add code to simulate random errors in workers: the supervisor must keep track of such problems and re-start a new worker if one is prematurely terminated.

```
%% MAIN FUNCTION
main(Count) ->
    register(the_master, self()), % I'm the master, now
    start_master(Count),
    unregister(the_master),
    io:format("That's all.~n").
```

**When two process are linked, when one dies or terminates, the other is killed, too.** To transform this kill message to an actual manageable message, we need to set its `trap_exit` process flag.

```
%% STARTING THE MASTER AND ITS CHILDREN
start_master(Count) ->
    % The master needs to trap exits:
    process_flag(trap_exit, true),
    create_children(Count),
    master_loop(Count).

% This creates the linked children
create_children(0) -> ok;
create_children(N) ->
    % spawn + link
    Child = spawn_link(?MODULE, child, [0]),
    io:format("Child ~p created~n", [Child]),
    Child ! {add, 0},
    create_children(N-1).

%% MASTER'S LOOP
master_loop(Count) ->
    receive
        {value, Child, V} ->
            io:format("child ~p has value ~p ~n",
                [Child, V]),
            Child ! {add, rand:uniform(10)},
            master_loop(Count);
        {'EXIT', Child, normal} ->
            io:format("child ~p has ended ~n",
                [Child]),
            if
            Count =:= 1 -> ok; % this was the last
            true -> master_loop(Count-1)
            end;
            % "unnormal" termination
        {'EXIT', Child, _} ->
            NewChild =
                spawn_link(?MODULE, child, [0]),
            io:format("child ~p has died,
                now replaced by ~p ~n",
            [Child, NewChild]),
            NewChild ! {add, rand:uniform(10)},
            master_loop(Count)
    end.

%% CHILDREN'S MAIN LOOP
child(Data) ->
    receive
        {add, V} ->
            NewData = Data+V,
            BadChance = rand:uniform(10) < 2,
            if
                % random error in child:
                BadChance -> error("I'm dying");
                % child ends naturally:
                NewData > 30 -> ok;
                % there is still work to do:
                true ->
                    the_master !
                        {value, self(), NewData},
                    child(NewData)
            end
    end.
```

# Past exams

## 2023/09/12

### *#TREES*

**Text**

Consider the infinite list of binary trees of Exercise 2: instead of infinite lists, we want to create processes which return the current element of the "virtual infinite list" with the message *next*, and terminate with the message *stop*.

1. Define a function *btrees* to create a process corresponding to the infinite tree of Exercise 2.1.
2. Define a function *incbtrees* to create a process corresponding to the infinite tree of Exercise 2.2.

Notes: for security reasons, processes must only answer to their creating process; to define trees, you can use suitable tuples with atoms as customary in Erlang (e.g. *branch, leaf, 1, leaf, 1*).

**Solution**

```
btrees_body(T, Pid) ->
    receive
        next ->
            T1 = {branch, T, T},
            Pid ! T1,
            btrees_body(T1, Pid);
        stop ->
            T
    end.

btrees(N, Pid) ->
    spawn(?MODULE, btrees_body,
        [{leaf, N}, Pid]).

inctree({leaf, X}) ->
    {leaf, X+1};
inctree({branch, X, Y}) ->
    {branch, inctree(X), inctree(Y)}.

incbtrees_body(T, Pid) ->
    receive
        next ->
            T1 = inctree(T),
            T2 = {branch, T1, T1},
            Pid ! T2,
            incbtrees_body(T2, Pid);
        stop ->
            T
    end.

incbtrees(Pid) ->
    spawn(?MODULE, incbtrees_body,
        [{leaf, 0}, Pid]).
```

## 2023/07/03

***#LISTS***

**Text**

1. Define a "deep reverse" function, which takes a "deep" list, i.e. a list containing possibly lists of any depths, and returns its reverse. E.g. *deeprev([1,2,[3,[4,5]],[6]]) is [[6],[[5,4],3],2,1].*
2. Define a parallel version of the previous function.

**Solution**

```
%%% STANDARD VERSION

deeprev([]) ->
    [];
deeprev([X | Xs]) ->
    V = deeprev(X),
    Vs = deeprev(Xs),
    % V doesn't need the square brackets,
    % because it will never fall in the case below,
    % but just on the empty-list one above.
    Vs ++ [V];
deeprev(X) ->
    X.

%%% PARALLEL VERSION

% Wrapper
deeprevp(L) ->
    P = self(),
    dp(P, L),
    receive
        {P, R} -> R
    end.

dp(Pid, []) ->
    Pid ! {self(), []};
dp(Pid, [X | Xs]) ->
    Self = self(),
    P1 = spawn(fun() -> dp(Self, X) end),
    P2 = spawn(fun() -> dp(Self, Xs) end),
    receive
        {P1, V} ->
            receive
                {P2, Vs} ->
                    Pid ! {Self, Vs ++ [V]}
            end
    end;
dp(Pid, X) ->
    Pid ! {self(), X}.
```

## 2023/06/12

***#LISTS #MERGESORT***

**Text**

Consider the following implementation of *mergesort* and write a parallel version of it.

```
mergesort([L]) -> [L];
mergesort(L) ->
{L1, L2} = lists:split(length(L) div 2, L),
merge(mergesort(L1), mergesort(L2)).
merge(L1, L2) -> merge(L1, L2, []).
merge([], L2, A) -> A ++ L2;
merge(L1, [], A) -> A ++ L1;
merge([H1|T1], [H2|T2], A) when H2 >= H1 ->
    merge(T1, [H2|T2], A ++ [H1]);
merge([H1|T1], [H2|T2], A) when H1 > H2 ->
    merge([H1|T1], T2, A ++ [H2]).
```

**Solution**

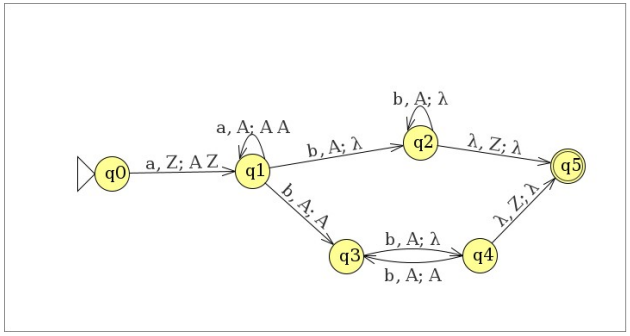
```
% initial method
mergesortp(L) ->
    Me = self(),
    msp(Me, L),
    receive
        % the spawn is not here:
        % the first process is the one executing
        % the function for the first time,
        % so it knows it's over when it is
        % the author of the last step
        {Me, R} -> R
    end.

% base case:
% list with only one element is sorted
msp(Pid, [L]) ->
    Pid ! {self(), [L]};
msp(Pid, L) ->
    % split in two and form a tuple
    {L1, L2} = lists:split(length(L) div 2, L),
    Me = self(),
    Pid1 = spawn(fun() -> msp(Me, L1) end),
    Pid2 = spawn(fun() -> msp(Me, L2) end),
    receive
        {Pid1, Sorted1} ->
            receive
                {Pid2, Sorted2} ->
                    Pid !{self(),
                        merge(Sorted1, Sorted2)}
            end
    end.

% add empty list
merge(L1, L2) -> merge(L1, L2, []).
% when left half is empty,
% concatenate with right half
merge([], L2, A) -> A ++ L2;
% when right half is empty,
% concatenate with left half
merge(L1, [], A) -> A ++ L1;
merge([H1 | T1], [H2 | T2], A) when H2 >= H1 ->
    merge(T1, [H2 | T2], A ++ [H1]);
merge([H1 | T1], [H2 | T2], A) when H1 > H2 ->
    merge([H1 | T1], T2, A ++ [H2]).
```

## 2023/02/15

***#AUTOMATON***

**Text**

Consider the following non-deterministic pushdown automaton (PDA), where Z is the initial stack symbol and $\lambda$ represents the empty string:




Write a concurrent Erlang program that simulates only the given PDA, and each state of the PDA is implemented as an independent parallel process.

**Solution**

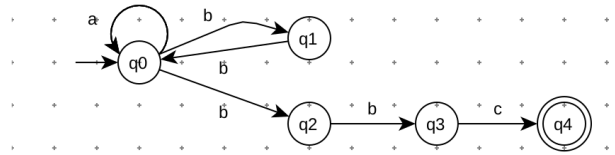
```
q0() ->
    receive
        {S, [a | Xs], [z | T]} ->
            q1 ! {S, Xs, [a, z] ++ T}
    end,
    q0().

q1() ->
    receive
        {S, [a | Xs], [a | T]} ->
            q1 ! {S, Xs, [a, a] ++ T};
        {S, [b | Xs], [a | T]} ->
            q2 ! {S, Xs, T},
            q3 ! {S, Xs, [a | T]}
    end,
    q1().

q2() ->
    receive
        {S, [b | Xs], [a | T]} ->
            q2 ! {S, Xs, T};
        {S, Xs, [z | T]} -> q5 ! {S, Xs, T}
    end,
    q2().

q3() ->
    receive
        {S, [b | Xs], [a | T]} ->
            q4 ! {S, Xs, T}
    end,
    q3().

q4() ->
    receive
        {S, [b | Xs], [a | T]} ->
            q3 ! {S, Xs, [a | T]};
        {S, Xs, [z | T]} -> q5 ! {S, Xs, T}
    end,
    q4().

q5() ->
    receive
        {S, [], _} ->
            io:format("~w accepted~n", [S])
    end,
    q5().

start() ->
    % to avoid exporting qs
    register(q0, spawn(fun() -> q0() end)),
    register(q1, spawn(fun() -> q1() end)),
    register(q2, spawn(fun() -> q2() end)),
    register(q3, spawn(fun() -> q3() end)),
    register(q4, spawn(fun() -> q4() end)),
    register(q5, spawn(fun() -> q5() end)).

stop() ->
    unregister(q0),
    unregister(q1),
    unregister(q2),
    unregister(q3),
    unregister(q4),
    unregister(q5).

read_string(S) ->
    q0 ! {S, S, [z]},
    ok.
```

## 2023/01/25

***#AUTOMATON***

**Text**

Consider the following non-deterministic finite state automaton (FSA):




Write a concurrent Erlang program that simulates the previous FSA, where each state is implemented as a process.

**Solution**

```
q0() ->
    receive
        {S, [a | Xs]} ->
            q0 ! {S, Xs};
        {S, [b | Xs]} ->
            q1 ! {S, Xs},
            q2 ! {S, Xs}
    end,
    q0().

q1() ->
    receive
        {S, [b | Xs]} -> q0 ! {S, Xs}
    end,
    q1().

q2() ->
    receive
        {S, [b | Xs]} -> q3 ! {S, Xs}
    end,
    q2().

q3() ->
    receive
        {S, [c | Xs]} -> q4 ! {S, Xs}
    end,
    q3().

q4() ->
    receive
        {S, []} ->
            io:format("~w accepted~n", [S])
    end,
    q4().

start() ->
    register(q0, spawn(fun() -> q0() end)),
    register(q1, spawn(fun() -> q1() end)),
    register(q2, spawn(fun() -> q2() end)),
    register(q3, spawn(fun() -> q3() end)),
    register(q4, spawn(fun() -> q4() end)).

stop() ->
    unregister(q0),
    unregister(q1),
    unregister(q2),
    unregister(q3),
    unregister(q4).

read_string(S) ->
    q0 ! {S, S},
    ok.
```

## 2022/09/01

***#PARTITIONS #FOLD***

**Text**

We want to implement a parallel foldl, *parfold(F, L, N)*, where the binary operator F is associative, and N is the number of parallel processes in which to split the evaluation of the fold. Being F associative, parfold can evaluate foldl on the N partitions of L in parallel. Notice that there is no starting (or accumulating) value, differently from the standard foldl.
You may use the following libray functions:

`lists:foldl(<function>, <starting value>, <list>)`

`lists:sublist(<list>, <init>, <length>)`

**Solution**

```
%% you need this as a separate function
%% from parthelp just because the three
```

```
%% lines below are preparatory and needs
%% to be done just once at the beginning
partition(L, N) ->
    M = length(L),
    %% integer division
    Chunk = M div N,
    %% length of the last partition
    %% (in case the reminder is not 0)
    End = M - Chunk * (N - 1),
    %% recurring helper function
    %% (the third argument is the start
    %%of the current chunk
    parthelp(L, N, 1, Chunk, End, []).

parthelp(L, 1, P, _, E, Res) ->
    %% stop recurring, but add
    %% the last sublist to the list
    Res ++ [lists:sublist(L, P, E)];
parthelp(L, N, P, C, E, Res) ->
    %% sublist from P of length C
    R = lists:sublist(L, P, C),
    %% recursive call with N decreased
    %% by 1 (basically acting as a counter)
    %% and updated start of sublist
    parthelp(L, N - 1, P + C, C, E, Res ++ [R]).

parfold(F, L, N) ->
    %% Ls contains the list of sublists
    Ls = partition(L, N),
    %% W is the list of the processes' IDs
    W = [spawn(?MODULE, dofold,
        [self(), F, X]) || X <- Ls],
    %% store the first sublist (order is not important,
    %% since F is associative) in R and the rest in Rs
    [R | Rs] = [
        receive
            %% V is the result of the foldl
            %% of each partition
            {P, V} -> V
        end
     || P <- W
    ],
    %% comulative foldl
    lists:foldl(F, R, Rs).

%% X is the initial value needed for the foldl
dofold(Proc, F, [X | Xs]) ->
    Proc ! {self(), lists:foldl(F, X, Xs)}.
```

## 2022/07/06

***#ZIP***

**Text**

We want to implement an interface to a server protocol, for managing requests that are lists of functions of one argument and lists of data on which the functions are called.
The interface main function is called *multiple_query* and gets a list of functions *FunL*, and a list of data, *DataL*, of course of the same size.
The protocol works as follows (assume that there is a registered server called *master_server*):

1. First, we ask the master server for a list of slave processes that will perform the computation. The request has the following form:
   `slaves_request, identity, <id_of_the_caller>, quantity, <number_of_needed_slaves>`
2. The answer has the following form:
   `slaves_id, <list_of_ids_of_slave_processes>`
3. Then, the library sends the following requests to the slave processes: `compute_request, identity, <id_of_the_caller>, <function>, <data>`, where `<function>` is one of the elements of *FunL*, and `<data>` is the corresponding element of *DataL*.
4. Each process sends the result of its computation with a message: `compute_result, identity, <slave_id>, value, <result_value>`
5. *multiple_query* ends by returning the list of the computed results, that must be ordered according to *FunL* and *DataL*.

If you want, you may use *lists:zip/2* and *lists:zip3/3*, that are the standard zip operations on 2 or 3 lists, respectively.

**Solution**

```
multiple_query(FunL, DataL) ->
    master_server ! {slaves_request,
        {identity, self()}, {quantity, length(FunL)}},
    receive
        {slaves_id, Slaves} -> ok
    end,
    [
        S ! {compute_request, {identity, self()}, F, D}
     || {S, F, D} <- lists:zip3(Slaves, FunL, DataL)
    ],
    [
        receive
            {compute_result, {identity, S},
                {value, V}} -> V
        end
     || S <- Slaves
    ].
```

## 2022/06/16

***#PARTITION***

**Text**

Define a program *tripart* which takes a list, two values x and y, with x ¡ y, and three functions, taking one argument which must be a list.

- *tripart* first partitions the list in three sublists, one containing values that are less than both x and y, one containing values v such that $x \leq v \leq y$, and one containing values that are greater than both x and y.
- Three processes are then spawned in parallel, running the three given functions and passing the three sublists in order (i.e. the first function must work on the first sublist and so on).
- Lastly, the program must wait the termination of the three processes in the spawning order, assuming that each one will return the pair P, V, where P is its PID and V the resulting value.
- *tripart* must return the three resulting values in a list, with the resulting values in the same order as the corresponding sublists.

**Solution**

```
helper([X | L], P1, P2, L1, L2, L3)
    when (X < P1) and (X < P2) ->
    helper(L, P1, P2, [X | L1], L2, L3);
helper([X | L], P1, P2, L1, L2, L3)
    when (X >= P1) and (X =< P2) ->
    helper(L, P1, P2, L1, [X | L2], L3);
helper([X | L], P1, P2, L1, L2, L3)
    when (X > P1) and (X > P2) ->
    helper(L, P1, P2, L1, L2, [X | L3]);
helper([], _, _, L1, L2, L3) ->
    [L1, L2, L3].

part(L, X, Y) ->
    helper(L, X, Y, [], [], []).

tripart(L, X, Y, F1, F2, F3) ->
    [D1, D2, D3] = part(L, X, Y),
    P1 = spawn(?MODULE, F1, [D1]),
    P2 = spawn(?MODULE, F2, [D2]),
    P3 = spawn(?MODULE, F3, [D3]),
    receive
        {P1, V1} ->
            receive
                {P2, V2} ->
                    receive
                        {P3, V3} ->
                            [V1, V2, V3]
                    end
            end
    end.
```

## 2022/02/10

***#LISTSMAP #ASCII***

**Text**

Define a parallel lexer, which takes as input a string x and a chunk size n, and translates all the words in the strings to atoms, sending to each worker a chunk of x of size n (the last chunk could be shorter than n). You can assume that the words in the string are separated only by space characters (they can be more than one — the ASCII code for ‘ ’ is 32); it is ok also to split words, if they overlap on different chunks.
E.g. `plex("this is a nice test", 6)` returns *[[this,i],[s,a,ni],[ce,te],[st]]*
For you convenience, you can use the library functions:

- *lists:sublist(List, Position, Size)* which returns the sublist of List of size Size from position Position (starting at 1);
- *list_to_atom(Word)* which translates the string Word into an atom.

**Solution**

```
% clean function to partition a list
% into a list of sublists
split(List, Size, Pos, End) when Pos < End ->
    [lists:sublist(List, Pos, Size)] ++
        split(List, Size, Pos + Size, End);
split(_, _, _, _) ->
    [].

% 32 is ' '
lex([X | Xs], []) when X =:= 32 ->
    lex(Xs, []);
lex([X | Xs], Word) when X =:= 32 ->
    [list_to_atom(Word)] ++ lex(Xs, []);
lex([X | Xs], Word) ->
    lex(Xs, Word ++ [X]);
lex([], []) ->
    [];
lex([], Word) ->
    [list_to_atom(Word)].

run(Pid, Data) ->
    % each worker send to the original
    % process (Pid here) the result of calling
    % 'lex' on its own sublist
    Pid ! {self(), lex(Data, [])}.

plex(List, Size) ->
    Part = split(List, Size, 1, length(List)),
    % as many workers as the number of sublists
    W = lists:map(
        fun(X) ->
            % each worker gets its
            % respective sublist (X)
            spawn(?MODULE, run, [self(), X])
        end,
        Part
        % W will then contain the IDs
        % of the spawned workers
    ),
    lists:map(
        % for each worker, receive its atoms
        fun(P) ->
            receive
                {P, V} -> V
            end
        end,
        W
    ).
```

## 2022/01/21

### *#HASHTABLE #KEYSTORE #KEYFIND*

**Text**

Create a distributed *hash table* with *separate chaining*. The hash table will consist of an agent for each bucket, and a master agent that stores the buckets' PIDs and acts as a middleware between them and the user. Actual key/value pairs are stored into the bucket agents.
The middleware agent must be implemented by a function called `hashtable_spawn` that takes as its arguments (1) the hash function and (2) the number of buckets. When executed, `hashtable_spawn` spawns the bucket nodes, and starts listening for queries from the user. Such queries can be of two kinds:

- Insert: `insert, Key, Value` inserts a new element into the hash table, or updates it if an element with the same key exists;
- Lookup: `lookup, Key, RecipientPid` sends to the agent with PID "`RecipientPid`" a message of the form `found, Value`, where Value is the value associated with the given key, if any. If no such value exists, it sends the message `not_found`.

The following code:

```
main() ->
    HT = spawn(?MODULE, hashtable_spawn,
        [fun(Key) -> Key rem 7 end, 7]),
    HT ! {insert, 15, "Apple"},
    HT ! {insert, 8, "Orange"},
    timer:sleep(500),
    HT ! {lookup, 8, self()},
    receive
        {found, A1} -> io:format("~s~n", [A1])
    end,
    HT ! {insert, 8, "Pineapple"},
    timer:sleep(500),
    HT ! {lookup, 8, self()},
    receive
        {found, A2} -> io:format("~s~n", [A2])
    end.
```

should print the following:

```
Orange
Pineapple
```

**Solution**

```
hashtable_spawn(HashFun, NBuckets) ->
    % THIS IS A SMART WAY TO HAVE
    % SOMETHING SIMILAR TO A FOR LOOP
    BucketPids = [spawn(?MODULE, bucket, [[]])
        || _ <- lists:seq(0, NBuckets)],
    hashtable_loop(HashFun, BucketPids).

% SEPARATE FUNCTION NEEDED DUE TO RECURSION
hashtable_loop(HashFun, BucketPids) ->
    receive
        {insert, Key, Value} ->
            lists:nth(HashFun(Key) + 1,
                BucketPids) ! {insert, Key, Value},
            hashtable_loop(HashFun, BucketPids);
        {lookup, Key, AnswerPid} ->
            lists:nth(HashFun(Key) + 1,
                BucketPids) ! {lookup, Key, AnswerPid},
            hashtable_loop(HashFun, BucketPids)
    end.

bucket(Content) ->
    receive
        {insert, Key, Value} ->
            NewContent = lists:keystore(Key, 1,
                Content, {Key, Value}),
            bucket(NewContent);
        {lookup, Key, AnswerPid} ->
            case lists:keyfind(Key, 1, Content) of
                false ->
                    AnswerPid ! not_found;
                {_, Value} ->
                    AnswerPid ! {found, Value}
            end,
            bucket(Content)
    end.
```

---

Made by Antonio Sgarlata for the *Principles of Programming Languages* course held at Politecnico di Milano.
Template by Dave Richeson, Dickinson College,
http://divisbyzero.com/