Principle of Programming Languages

Erlang

Variables, Atoms, Tuples, Lists

Variables start with an Upper Case Letter (like in Prolog). Variables can only be bound once, the value of a variable can never be changed once it has been set.
Atoms are like symbols in Scheme. Any character code is allowed within an atom, singly quoted sequences of characters are atoms (not strings). unquoted must be lowercase, to avoid clashes with variables.
Tuples are used to store a fixed number of items.
Are like in Haskell, e.g. [1, 2, 3], ++ concatenates. Main difference: [X — L] is cons (like (cons X L)). Strings are lists, like in Haskell.

```
ACamelCaseVariableName % Variable (unmutable)
start_with_a_lower_case_letter % Atom (symbol)
{person, 'Jim', 'Austrian'} % Tuple: three
    atoms
```

Pattern Matching

Like in Prolog, = 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

```
Map = #{one => 1, "Two" => 2, 3 => three}. %
    Create
Map#{one := "I"}. % Update/insert
#{"Two" := V} = Map. % Search and assign
```

Functions and Modules

Function and **Module** names (func and module in the above) must be atoms. Functions are defined within Modules. Functions must be exported before they can be called from outside the module where they are defined.

```
-module(demo).
-export([double/1]).
double(X) -> times(X, 2). % Exported
times(X, N) -> X * N. % Not exported
```

Built-in functions in the erlang module:

```
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({1,2,3,4}) % [1,2,3,4]
```

A **Function** is defined as a sequence of clauses (separated by ";"). **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). The keyword `when` introduces a guard, like "|" in Haskell. All variables in a guard must be bound.

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

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.

Case and If

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

```
case lists:member(a, X) of
  true -> ... ;
  false -> ...
end,
%%%%%
if
  integer(X) -> ... ;
  tuple(X) -> ... ;
  true -> ... % works as an "else"
end,
```

Lambdas

```
Square = fun (X) -> X*X end. % Declaration
Square(3). % Usage
lists:map(Square, [1,2,3]). returns [1,4,9] %
    Lambdas can be passed as usual to higher
    order functions:
lists:foldr(fun my_function/2, 0, [1,2,3]). %
    To pass standard functions, we need to
    prefix their name with fun and state its
    arity:
```

Concurrent Programming: the Actor Model

Everything is an **Actor**: an independent unit of computation. Actors are inherently concurrent. Actors can only communicate through messages (async communication). Actors can be created dynamically. No requirement on the order of received messages.
There are three main primitives:

- `spawn` creates a new process executing the specified function, returning an identifier.
- `send` (written `!`) sends a message to a process through its identifier; the content of the message is simply a variable. The operation is asynchronous.
- `receive` end extract, 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

We have a process with `Pid1` (**Process Identity** or Pid). In it we perform `Pid2 = spawn(Mod, Func, Args)`. Like apply but spawning a new process. After, `Pid2` is the process identifier of the new process, this is known only to process `Pid1`.

Message Passing

```
PidB ! {self(), {mymessage, [1,2,3]}} %
    Process A sends a message to B
receive
  {From, {Msg, Data}} ->
    work_on_data(Data);
end
```

`self()` returns the Pid of the process executing it. `From`, `Msg` and `Data` become bound when the message is received. Messages can carry data and be selectively unpacked. If `From` is bound before receiving a message, then only data from that process is accepted.
Client-Server can be easily realized through a simple protocol, where requests have the syntax {`request, ...`}, while replies are written as {`reply, ...`}

Registered Processes

`register(Alias, Pid)` registers the process Pid with name Alias. Any process can send a message to a registered process.

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

Timeouts

```
% If the message foo is received within the
    time Time perform Actions1 otherwise
    perform Actions2.
wait(Time, message) ->
   receive
      foo -> Actions1;
   after
      Time -> Actions2;
sleep(T) -> % process suspends for T ms.
   receive
   after
      T -> true
   end.
suspend() -> % process suspends indefinitely.
   receive
   after
      infinity -> true
   end.
flush() -> % flushes the message buffer.
   receive
      Any -> flush()
   after
      0 -> true
   end.
```

Exams

2020-07-17

Define a "broadcaster" process which answers to the following commands:

- spawn, L, V creates a process for each element of L, passing its initial parameter in V, where L is a list of names of functions defined in the current module and V is their respective parameters (of course it must be $|L| = |V|$).
- send, V, with V a list of values, sends to each respective process created with the previous spawn command a message in V; e.g. spawn, [1,2,3] will send 1 to the first process, 2 to the second, and 3 to the third.
- stop is used to end the broadcaster, and to also stop every process spawned by it.

Solution:

```
broadcaster(Pids) ->
   receive
      {spawn, Fs, Vs} ->
         FDs = lists:zip(Fs, Vs),
         io:format("~p~n", [FDs]),
         broadcaster([spawn_link(?MODULE, F,
             V) || {F,V} <- FDs]);
      {send, Vs} ->
         FDs = lists:zip(Pids, Vs),
         io:format("~p~n", [FDs]),
         [ Pid ! V || {Pid, V} <- FDs];
      stop ->
         ok
   end.
```

2020-06-29

Define a "functional" process buffer, called fuffer, that stores only one value and may receive messages only from its creator. fuffer can receive the following commands:

- 'set' to store a new value.
- 'get' to obtain the current value.
- 'apply F' to apply the function F to the stored value.
- 'die' to end.
- 'duplicate' to create (and return) an exact copy of itself.

Solution:

```
fuffer(Value) ->
   receive
      {set, V} ->
         fuffer(V);
      {get, Sender} ->
         Sender ! {self(), Value},
         fuffer(Value);
      {apply, F} ->
         fuffer(F(Value));
      die ->
         ok;
      {duplicate, Sender} ->
         Pid = spawn(?MODULE, fuffer, [Value]),
         Sender ! {self(), Pid},
         fuffer(Value)
   end.
```

2020-02-07

We want to create a simplified implementation of the "Reduce" part of the MapReduce paradigm. To this end, define a process "reduce_manager" that keeps track of a pool of reducers. When it is created, it stores a user-defined associative binary function ReduceF. It receives messages of the form reduce, Key, Value, and forwards them to a different "reducer" process for each key, which is created lazily (i.e. only when needed). Each reducer serves requests for a unique key. Reducers keep into an accumulator variable the result of the application of ReduceF to the values they receive. When they receive a new value, they apply ReduceF to the accumulator and the new value, updating the former. When the reduce_manager receives the message print_results, it makes all its reducers print their key and incremental result.
Solution:

```
start_reduce_mgr(ReduceF) ->
   spawn(?MODULE, reduce_mgr, [ReduceF, #{}]).

reduce_mgr(ReduceF, Reducers) ->
   receive
      print_results ->
         lists:foreach(fun ({_, RPid}) -> RPid
             ! print_results end,
             maps:to_list(Reducers));
      {reduce, Key, Value} ->
         case Reducers of
            #{Key := RPid} ->
               RPid ! {Key, Value},
               reduce_mgr(ReduceF, Reducers);
            _ ->
               NewReducer = spawn(?MODULE,
                   reducer, [ReduceF, Key,
                   Value]),
               reduce_mgr(ReduceF,
                   Reducers#{Key => NewReducer})
         end
   end.

reducer(ReduceF, Key, Result) ->
   receive
      print_results ->
         io:format("~s: ~w~n", [Key, Result]);
      {Key, Value} ->
         reducer(ReduceF, Key, ReduceF(Result,
             Value))
   end.
```

2020-01-15

We want to implement something like Python's `range` in Erlang, using processes.
E.g.

```
R = range(1,5,1)  % starting value, end
    value, step
next(R)           % is 1
next(R)           % is 2
. . .
next(R)           % is 5
next(R)           % is the atom
    stop_iteration
```

Define `range` and `next`, where `range` creates a process that manages the iteration, and `next` a function that talks with it, asking the current value.
Solution:

```
ranger(Current, Stop, Inc) ->
    receive
        {Pid, next} ->
            if
                Current =:= Stop -> Pid !
                    stop_iteration;
                true -> Pid ! Current,
                        ranger(Current + Inc,
                            Stop, Inc)
            end
    end.
range(Start, Stop, Inc) ->
    spawn(?MODULE, ranger, [Start, Stop,
        Inc]).
next(Pid) ->
    Pid ! {self(), next},
    receive
        V -> V
    end.
```

2019-09-03

- 1) Define a split function, which takes a list and a number n and returns a pair of lists, where the first one is the prefix of the given list, and the second one is the suffix of the list of length n.
  E.g. split([1,2,3,4,5], 2) is [1,2,3],[4,5]
- 2) Using split of 1), define a splitmap function which takes a function f, a list L, and a value n, and splits L with parameter n, then launches two process to map f on each one of the two lists resulting from the split. The function splitmap must return a pair with the two mapped lists.

```
helper(E, {0, L}) ->
    {-1, [[E]|L]};
helper(E, {V, [X|Xs]}) ->
    {V-1, [[E|X]|Xs]}.

split(L, N) ->
    {_, R} = lists:foldr(fun helper/2, {N,
        [[]]}, L),
    R.

mapper(F, List, Who) ->
    Who ! {self(), lists:map(F, List)}.

splitmap(F, L, N) ->
    [L1, L2] = split(L, N),
    P1 = spawn(?MODULE, mapper, [F, L1,
        self()]),
    P2 = spawn(?MODULE, mapper, [F, L2,
        self()]),
    receive
        {P1, V1} ->
            receive {P2, V2} ->
                    {V1, V2}
            end
    end.
```

2019-07-24

Define a master process which takes a list of nullary (or 0-arity) functions, and starts a worker process for each of them. The master must monitor all the workers and, if one fails for some reason, must re-start it to run the same code as before. The master ends when all the workers are done.
Note: for simplicity, you can use the library function spawn_link/1, which takes a lambda function, and spawns and links a process running it.

```
listlink([], Pids) -> Pids;
listlink([F|Fs], Pids) ->
    Pid = spawn_link(F),
    listlink(Fs, Pids#{Pid => F}).

master(Functions) ->
    process_flag(trap_exit, true),
    Workers = listlink(Functions, #{}),
    master_loop(Workers, length(Functions)).

master_loop(Workers, Count) ->
    receive
        {'EXIT', Child, normal} ->
            if
                Count =:= 1 -> ok;
                true -> master_loop(Workers,
                    Count-1)
            end;
        {'EXIT', Child, _} ->
            #{Child := Fun} = Workers,
            Pid = spawn_link(Fun),
            master_loop(Workers#{Pid => Fun},
                Count)
    end.
```

2019-06-28

Being bu and the two cf spawn-linked, we need to terminate the main process with exit to kill them. If we do not do so, we get an unending sequence of "no data". Hence, we could change "P1, done -> ok" into "P1, done -> exit(done)". The output of the system is correct, because it is possible that some elements remain in the buffer, being the buffer managed as a stack.

2019-02-08

lists:nth(V,Buf) (get element V of array B) V=1,2,3,..

2019-01-16

Define a process P, having a local behavior (a function), that answer to three commands:
- **load** is used to load a new function f on P: the previous behavior is composed with f;
- **run** is used to send some data D to P: P returns its behavior applied to D;
- **stop** is used to stop P.

For security reasons, the process must only work with messages coming from its creator: other messages must be discarded.

```
cam(Beh, Who) ->
receive
    {run, Who, What} ->
        Who ! Beh(What),
        cam(Beh, Who);
    {load, Who, Code} ->
        cam(fun (X) -> Code(Beh(X)) end, Who);
    {stop, Who} ->
        ok;
    _ -> cam(Beh, Who)
end.
```

2018-09-05

Define a function create_pipe, which takes a list of names and creates a process of each element of the list, each process registered as its name in the list; e.g. with [one, two], it creates two processes called 'one' and 'two'. The processes are "connected" (like in a list, there is the concept of "next

process") from the last to the first, e.g. with [one, two, three], the process structure is the following:
`three -> two -> one -> self`,
this means that the next process of 'three' is 'two', and so on; self is the process that called create_pipe.
Each process is a simple repeater, showing on the screen its name and the received message, then sends it to the next process.
Each process ends after receiving the 'kill' message, unregistering itself.
Solution:

```
repeater(Next, Name) ->
    receive
        kill ->
            unregister(Name),
            Next ! kill;
        V ->
            io:format("~p got ~p~n", [Name,
                V]),
            Next ! V,
            repeater(Next, Name)
    end.

create_pipe([], End) -> End;
create_pipe([X|Xs], Next) ->
    P = spawn(?MODULE, repeater, [Next, X]),
    register(X, P),
    create_pipe(Xs, X).
```

2018-02-05

The fixed-point of a function f and a starting value x is the value $v = f^k(x)$, with $k > 0$, such that $f^k(x) = f^{k+1}(x)$. We want to implement a fixed-point code using two communicating actors:

- 1) Define the function for an `applier` actor, which has a state S, holding a value, and receives a function f from other actors: if S = f(S), it sends back the result S and ends it computation; otherwise sends back a message to state that the condition S = f(S) has not been reached.
- 2) Define a function called fix, which takes as input a function and a starting value, and creates and uses an applier actor to implement the fixed-point.

Solution:

```
applier(State) ->
    receive
        {Sender, F} -> NewState = F(State),
            if
                NewState =:= State ->
                    Sender!{self(), State};
                true -> Sender!{self(), no},
                    applier(NewState)
            end
    end.
loop(P, F) ->
    P!{self(), F},
    receive
        {P, V} -> if
            V =:= no -> loop(P, F);
            true -> V
        end
    end.
fix(F, V) ->
A = spawn(?MODULE, applier, [V]),
loop(A, F).
```

Exercise session
2019-12-10-a

```
-module('2019-12-10-a').
%-export([add/2, factorial/1, factorial/2]).
-compile(export_all).

add(A, B) ->
    A + B.

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

factorial(N, Acc) when N =< 0 ->
    Acc;
factorial(N, Acc) ->
    factorial(N-1, N*Acc).

greet(male, Name) ->
    io:format("Hello, Mr. ~s~n", [Name]);
greet(female, Name) ->
    io:format("Hello, Ms. ~s~n", [Name]);
greet(_, Name) ->
    io:format("Hello, ~s~n", [Name]).

car([X | _]) ->
    X.

ht([X | T]) ->
    io:format("Head: ~w~nTail: ~w~n", [X, T]).

caar([_, X2 | _]) ->
    X2.

map(_, []) ->
    [];
map(F, [X | Xs]) ->
    [F(X) | map(F, Xs)].

foldr(_, Acc, []) ->
    Acc;
foldr(F, Acc, [X | Xs]) ->
    F(X, foldr(F, Acc, Xs)).

foldl(_, Acc, []) ->
    Acc;
foldl(F, Acc, [X | Xs]) ->
    foldl(F, F(Acc, X), Xs).

filter(_, []) ->
    [];
filter(P, [X | Xs]) ->
    case P(X) of
    true ->
        [X | filter(P, Xs)];
    false ->
        filter(P, Xs)
    end.

merge([], L) ->
    L;
merge(L, []) ->
    L;
merge([X | Xs], [Y | Ys]) ->
    if
    X =< Y ->
        [X | merge(Xs, [Y | Ys])];
    true ->
        [Y | merge([X | Xs], Ys)]
    end.

merge_sort(L) ->
    spawn(?MODULE, ms_split, [self(), left, L]),
    receive
    {_, LSorted} ->
        LSorted
    end.

ms_split(Parent, Side, []) ->
    Parent ! {Side, []};
```

```
ms_split(Parent, Side, [X]) ->
    Parent ! {Side, [X]};
ms_split(Parent, Side, L) ->
    {Ll, Lr} = lists:split(length(L) div 2, L),
    spawn(?MODULE, ms_split, [self(), left, Ll]),
    spawn(?MODULE, ms_split, [self(), right,
        Lr]),
    ms_merge(Parent, Side, empty, empty).

ms_merge(Parent, Side, Ll, Lr)
    when (Ll == empty) or (Lr == empty) ->
    receive
    {left, LlSorted} ->
        ms_merge(Parent, Side, LlSorted, Lr);
    {right, LrSorted} ->
        ms_merge(Parent, Side, Ll, LrSorted)
    end;
ms_merge(Parent, Side, Ll, Lr) ->
    Parent ! {Side, merge(Ll, Lr)}.


split(_, B) when B < 0 -> err;
split(A, B) when length(A) < B -> err;
split([A|As],B) when length(As) =:= B -> {A,As}.


mapper(F, L, To) ->
    To ! {self(), lists:map(F,L)}.

split_map(F, L, N) ->
    [L1|L2] = split(L, N),
    P1 = spawn(?MODULE, mapper, [F, L1, self()]),
    P2 = spawn(?MODULE, mapper, [F, L2, self()]),
    receive {P1, V1} ->
        receive {P2, V2} ->
            {V1, V2}
        end
    end.
```

2019-12-10-b

```
-module('2019-12-10-b').
-compile(export_all).

start_broker() ->
    BrokerPid = spawn(?MODULE, broker, []),
    register(broker, BrokerPid).

broker() ->
    Topics = #{erlang => [], haskell => [],
        scheme => []},
    Messages = #{erlang => [], haskell => [],
        scheme => []},
    broker_loop(Topics, Messages).

broker_loop(Topics, Messages) ->
    receive
    {subscribe, Topic, Pid} ->
        io:format("Registering ~p for topic ~w~n",
            [Pid, Topic]),
        #{Topic := Pids} = Topics,
        broker_loop(Topics#{Topic := [Pid |
            Pids]}, Messages);
    {produce, Topic, Msg} ->
        io:format("Spreading message ~s for topic
            ~w~n", [Msg, Topic]),
        #{Topic := Pids} = Topics,
        [P ! {publish, Topic, Msg} || P <- Pids],
        #{Topic := Msgs} = Messages,
        broker_loop(Topics, Messages#{Topic :=
            [Msg | Msgs]});
    {update, Topic, Pid} ->
        io:format("Requested repost of topic ~w by
            process ~p~n",
            [Topic, Pid]),
        #{Topic := Msgs} = Messages,
        repost(Topic, Msgs, Pid),
        broker_loop(Topics, Messages)
    end.

repost(_, [], _) ->
    ok;
repost(Topic, [Msg | Rest], Pid) ->
    Pid ! {repost, Topic, Msg},
    repost(Topic, Rest, Pid).

consumer(Topics) ->
    [broker ! {subscribe, Topic, self()} ||
        Topic <- Topics],
    consumer_loop(Topics).

consumer_loop(Topics) ->
    receive
    {publish, Topic, Message} ->
        io:format("Received message ~s for topic
            ~w~n",
            [Message, Topic]),
        consumer_loop(Topics)
    after 10000 ->
        Chance = rand:uniform(100) =< 5,
        if
        Chance ->
            exit("Bye.");
        true ->
            [broker ! {update, T, self()} || T <-
                Topics]
        end,
        consumer_loop(Topics)
    end.

start_consumers() ->
    [spawn_link(?MODULE, consumer, [[haskell]])
        || _ <- [1,2]],
    [spawn_link(?MODULE, consumer, [[erlang]])
        || _ <- [1,2]],
    [spawn_link(?MODULE, consumer, [[scheme]])
        || _ <- [1,2]],
    handle_exit().

handle_exit() ->
    process_flag(trap_exit, true),
    receive
    {'EXIT', Pid, Msg} ->
        io:format("Process ~p died with message:
            ~s~n",
            [Pid, Msg]),
        handle_exit()
    end.

producer(Message, Topic) ->
    broker ! {produce, Topic, Message}.

start() ->
    start_broker(),
    spawn(?MODULE, start_consumers, []).
```

2019-12-13

```
-module('2019-12-13').
-compile(export_all).

% Make a ring of linked nodes
start_ring(N) ->
    Pids = start_ring_node(N),
    First = hd(Pids),
    First ! {init, First},
    Pids.
```

```
start_ring_node(0) ->
    NodePid = spawn(?MODULE, ring_node, [last]),
    [NodePid];
start_ring_node(N) ->
    [NextPid | Rest] = start_ring_node(N-1),
    NodePid = spawn(?MODULE, ring_node,
        [NextPid]),
    [NodePid, NextPid | Rest].

% Close the ring by sending the PID of the first
    node all the way to the last node.
ring_node(last) ->
    receive
    {init, First} ->
        ring_node_loop(First)
    end;
ring_node(NextPid) ->
    receive
    {init, First} ->
        NextPid ! {init, First},
        ring_node_loop(NextPid)
    end.

% Implement the following message handlers:
% - Round trip from any node;
% - Send message from a node to the one N
    positions ahead;
% - Make a node terminate, and fix the ring by
    restoring the link
%   between the previous node and the next one.
ring_node_loop(NextPid) ->
    receive
    {roundtrip, Msg} ->
        io:format("~p: Starting round trip for
            message ~s.~n", [self(), Msg]),
        NextPid ! {roundtrip, self(), Msg},
        ring_node_loop(NextPid);
    {roundtrip, TargetPid, Msg} when TargetPid
        == self() ->
        io:format("~p: End of round trip for
            message ~s.~n", [self(), Msg]),
        ring_node_loop(NextPid);
    {roundtrip, TargetPid, Msg} ->
        io:format("~p: Continuing round trip of
            message ~s.~n", [self(), Msg]),
        NextPid ! {roundtrip, TargetPid, Msg},
        ring_node_loop(NextPid);
    {nextn, 0, Msg} ->
        io:format("~p: Message received: ~s.~n",
            [self(), Msg]),
        ring_node_loop(NextPid);
    {nextn, N, Msg} ->
        io:format("~p: Sending message ~s to ~wth
            subsequent node.~n", [self(), Msg,
            N]),
        NextPid ! {nextn, N-1, Msg},
        ring_node_loop(NextPid);
    {die} ->
        io:format("~p: Bye.~n", [self()]),
        NextPid ! {notifydeath, self(), NextPid};
    {notifydeath, NextPid, NewNextPid} ->
        io:format("~p: RIP ~p.~n", [self(),
            NextPid]),
        ring_node_loop(NewNextPid);
    {notifydeath, DeadPid, NewNextPid} ->
        NextPid ! {notifydeath, DeadPid,
            NewNextPid},
        ring_node_loop(NextPid)
    end.

do_ringy_thingy() ->
    Nodes = start_ring(10),
    io:format("~w~n", [Nodes]),
    hd(Nodes) ! {roundtrip, "Giro!"},
    lists:nth(3, Nodes) ! {nextn, 15,
        "Quindici!"},
    timer:sleep(5000),
    lists:nth(6, Nodes) ! {die},
    timer:sleep(5000),
    hd(Nodes) ! {roundtrip, "Giro!"},
    ok.


% Exam 2017-07-05
% Define an **activate** function, which takes
    as input:

% a binary tree stored with tuples, e.g.
    {branch, {branch, {leaf, 1}, {leaf, 2}},
    {leaf, 3}}.
% a binary function F
% then **activate** creates a network of actors
    having the same structure of the given tree.

% Actors corresponding to leaves run a function
    called **leafy**,
% and answer messages {ask, P} where P is a
    process, with
% the pair {self(), V}, where V is the value
    stored in the leaf, then terminate.

% Actors for the branches run a function called
    **branchy**,
% if a branch actor receives an {ask, P} request
    it forward it to both its leaves, and
% when a branch actor receives the answers, they
    call
% F on the obtained values, then send the result
    V to P as
% {self(), V} and terminate.

% E.g. running the following code:
% test() ->
%  T1 = {branch,
%     {branch,
%        {leaf, 1},
%           {leaf, 2}},
%     {leaf, 3}},
%  A1 = activate(T1, fun min/2),
%  A1 ! {ask, self()},
%  receive
%  {A1, V} -> V
%  end.
% should return 1.


activate({leaf, V}, _) ->
    io:format("~p: Spawning a leaf ~w~n",
        [self(), V]),
    spawn(?MODULE, leafy, [V]);
activate({branch, L, R}, F) ->
    io:format("~p: Spawning a branch~n",
        [self()]),
    Tl = activate(L, F),
    Tr = activate(R, F),
    spawn(?MODULE, branchy, [F, Tl, Tr, none,
        false, false]).

leafy(V) ->
    receive
    {ask, P} ->
        io:format("~p: Sending leaf value ~w to
            ~p.~n", [self(), V, P]),
        P ! {self(), V};
        M ->
```

```
        error("Protocol error: ~w~n", [M])
    end.

branchy(F, L, R, none, _, _) ->
    receive
    {ask, P} ->
        io:format("~p: Farwarding to leaves ~p
            ~p~n", [self(), L, R]),
        L ! {ask, self()},
        R ! {ask, self()},
        branchy(F, L, R, wait, false, P);
        M ->
        error("Protocol error: ~w~n", [M])
    end;
branchy(F, L, R, wait, _, P) ->
    receive
    {L, V} ->
        branchy(F, L, R, recl, V, P);
    {R, V} ->
        branchy(F, L, R, recr, V, P);
        M ->
        error("Protocol error: ~w~n", [M])
    end;
branchy(F, _, R, recl, Vl, P) ->
    receive
    {R, Vr} ->
        P ! {self(), F(Vl, Vr)};
           M ->
        error("Protocol error: ~w~n", [M])
    end;
branchy(F, L, _, recr, Vr, P) ->
    receive
    {L, Vl} ->
        P ! {self(), F(Vl, Vr)};
    M ->
        error("Protocol error: ~w~n", [M])
end.

main() ->
    T1 = {branch, {branch, {leaf, 1}, {leaf,
        2}}, {leaf, 3}},
    A = activate(T1, fun min/2),
    io:format("Asking ~p~n", [A]),
    A ! {ask, self()},
    receive
    {A, V} ->
        io:format("The value is ~w~n", [V])
    end.
```

```
% Exam 2017-07-20
% We want to define a "dynamic list" data
    structure, where
% each element of the list is an actor storing a
    value.
% Such value can be read and set in parallel.

% 1) Define create_dlist, which takes a number n
    and
%   returns a dynamic list of length n
    initialized to 0.
% 2) Define the function dlist_to_list,
%   which takes a dynamic list and returns a list
%   of the contained values.
% 3) Define a map for dynamic list.
%   Of course this operation has side effects,
%   since it changes the content of the list.

create_dlist(0) ->
    [];
create_dlist(N) ->
    [spawn(?MODULE, dlist, [I, 0]) || I <-
        lists:seq(1, N)].

dlist(I, V) ->
    receive
    {set, NewV} ->
        dlist(I, NewV);
    {get, P} ->
        P ! V,
        dlist(I, V)
    end.

set(I, L, V) ->
    lists:nth(I+1, L) ! {set, V}.

get(I, L) ->
    lists:nth(I+1, L) ! {get, self()},
    receive
    V ->
        V
    end.


dlist_to_list([]) ->
    [];
dlist_to_list(Ps) ->
    d2l(Ps, []).


d2l([], L) ->
    L;
d2l([P | Ps], L) ->
    P ! {get, self()},
    receive
    V ->
        d2l(Ps, L ++ [V])
    end.

dmap([], _) ->
    ok;
dmap([P | Ps], F) ->
    P ! {get, self()},
    receive
    V ->
        P ! {set, F(V)},
        dmap(Ps, F)
end.

dfoldl([], Acc, _) ->
    Acc;
dfoldl([P | Ps], Acc, F) ->
    P ! {get, self()},
    receive
    V ->
        dfoldl(Ps, F(Acc, V), F)
    end.


main2() ->
    L = create_dlist(10),
    [set(N, L, N) || N <- lists:seq(0,9)],
    LL = dlist_to_list(L),
    io:format("List: ~w~n", [LL]),
    dmap(L, fun(X) -> X*2 end),
    L2 = dlist_to_list(L),
    io:format("List: ~w~n", [L2]),
io:format("Sum: ~w~n", [dfoldl(L, 0, fun(X, Y)
    -> X+Y end)]),
    ok.
```