

MPP 2026/27 (0077A, 9CFU)

Models for Programming Paradigms

Roberto Bruni Filippo Bonchi
http://www.di.unipi.it/~bruni/

https://didawiki.di.unipi.it/doku.php/magistraleinformatica/mpp/start

16a - Erlang

Erlang

concurrency oriented programming

Erlang: a concurrent programming language

http://www.erlang.org/

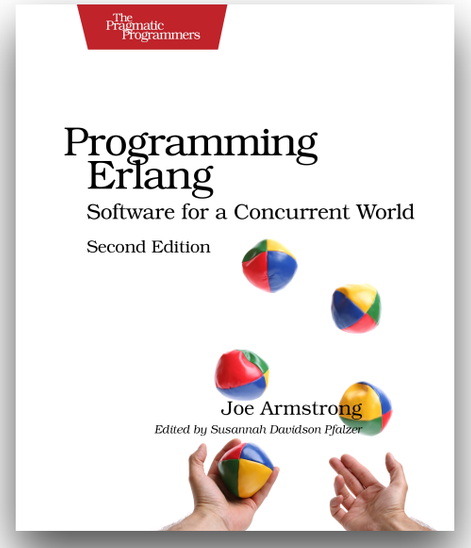


Erlang: origins

named after Danish mathematician A. K. Erlang

1986: first experimentation at Ericsson, Sweden
1989: internal use only
1990: sold as a product
1998: open source

Joe Armstrong, “Programming Erlang”, ch.1-5, 11-12



Features

declarative (functional, Prolog) programming

arbitrary size integers, tuples, lists, functions, higher-order

atoms everywhere

dynamically typed

open source

unfriendly syntax

variables are assigned only once

left-to-right evaluation, no pointers, no object-orientation

Features: concurrency

concurrent and distributed programming

asynchronous message passing
(no locks, no mutexes)

fault tolerance

hot swapping code

erlang processes are cheap

automatic memory allocation and garbage collection

can handle large telecom applications

Erl

Erlang: erl

erl is the Erlang VM emulator

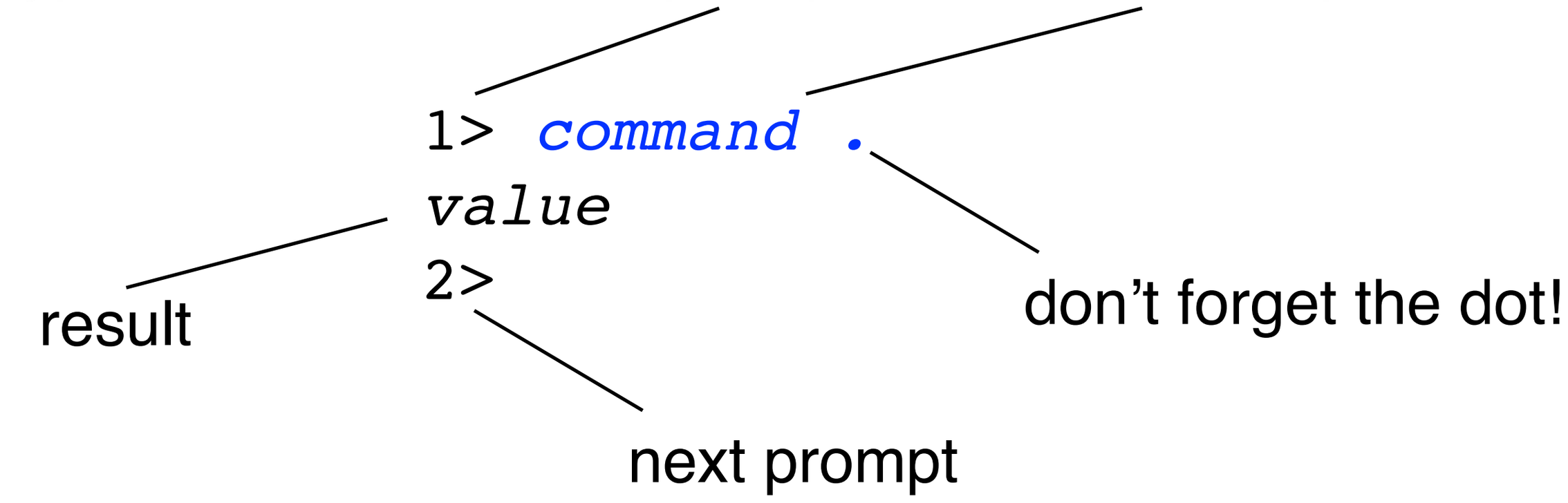
interactive shell or interpreter, executing read-eval-print loop

programmers enter expressions / declarations one at a time

they are compiled / executed

erl expressions

typical interaction:



`halt().` to exit the emulator

Erlang modules

functions are organised in modules

one module for source file

filename is module name with suffix `.erl`

a comment — arity — declarations end with a dot

```
% filename hello.erl
-module(hello).
-export([hello/0]).

hello() -> io:format("Hello, world!~n").
```

function def

module name — separator — function name — argument

erl: module loading

compile and load the module

```
1> c(hello) .
{ok,hello}
2> hello:hello() .
Hello, world!
ok
3>
```

invoke the function

return value

next prompt

if you edit `hello.erl` and do `c(hello)` again
the new version of the module replaces the old one

Erlang basics

Function definition

separates function clauses with ;
last clause ends with .

variables start with upper-case letters X Head Tail
variables are local to function clauses

function definitions cannot be nested
non-exported functions are local to the module

pattern matching allowed

guards allowed (keyword `when`)

type-checking is done at runtime

Atoms, tuples, lists

numbers: arbitrary size integers, floating point values
(cannot start with .)

atoms: start with lower-case character
(can be single-quoted if needed, don't use camelCase)

```
true  ok  hello_world.  'this is an atom'
```

tuples: main data constructor
tagged tuples: the first element of the tuple is an atom
we can use pattern matching

```
{}  {movie,"Matrix"}   {movie,Title}
```

lists: can contain elements of any type
we can use pattern matching

```
[]  [1,2,ok]  [H|T]  [X,Y,Z]  [X,Y,Z| Tail]
```

Funs

funs: anonymous functions (lambda expressions)
can have several arguments and clauses

```
fun () -> 42 end
```

```
fun (X) -> X+1 end
```

```
fun (X,Y) -> {X, fun (Z) -> Z+Y end} end
```

```
fun (F,X) -> F(X) end
```

Type test & conversion

```
is_integer(X)
is_float(X)
is_number(X)
is_atom(X)
is_tuple(X)
is_list(X)
is_function(X)
is_pid(X)
…
```

```
atom_to_list(A)
list_to_atom(L)
tuple_to_list(T)
list_to_tuple(L)
…
```

Erlang concurrency

Processes

every Erlang code is executed by a process

processes are implemented by the VM (not by OS threads)

multitasking is preemptive (VM switching and scheduling)

processes need very little memory

switching between processes is very fast

the VM can handle a large number of processes

on multiprocessor/multicore machines, processes can be scheduled to run in parallel on separate CPUs/cores using multiple schedulers

different processes may be reading the same program code at the same time (no variable updates!)

Pids

each process has a process identifier

```
Pid = self()
```

new Erlang processes can be spawned to run functions

```
Pid = spawn(module,function,arguments)
```

```
Pid = spawn(fun () -> … end)
```

```
Pid = spawn(fun f/0)
```

```
Pid = spawn(fun m:f/0)
```

the spawn operation returns immediately
(the return value is the pid of the process)

children pids are available to parent process,
not vice versa (unless passed)

Communication

Messages can be sent to pids

```
Pid ! message
```



called bang

Processes can wait to receive (and select) some message

```
receive
    Pattern1 when Cond1 -> Exp1;
    Pattern2 when Cond2 -> Exp2;
    ...
    Patternk when Condk -> Expk
end
```

Communication

Messages can be sent to pids

```
Pid ! {1,2,3}
```

called bang

Processes can wait to receive (and select) some message

```
receive
    {X} when X>0 -> X;
    {X,Y} when Y>X -> X+Y;
    {X,Y,Z} when Y>X andalso Z>Y -> X+Z;
end
```

First matching clause for first message,
if none, first matching clause for second message,
if none, ...
if none it blocks (all messages are kept)

Communication

Messages can be sent to pids

```
Pid ! {1,2,3}
```

called bang

Processes can wait to receive (and select) some message

```
receive
    {X} when X>0 -> X;
    {X,Y} when Y>X -> X+Y;
    {X,Y,Z} when Y>X andalso Z>Y -> X+Z;
    _ -> 0
end
```

First matching clause for first message
(the last clause, called catch-all, will match anyway)

Communication

Messages can be sent to pids

```
Pid ! {1,2,3}
```

called bang

Processes can wait to receive (and select) some message

```
receive
    {X} when X>0 -> X;
    {X,Y} when Y>X -> X+Y;
    {X,Y,Z} when Y>X andalso Z>Y -> X+Z;
    after 0 -> 0
end
```

timeout

First matching clause for first message,
if none, first matching clause for second message,
if none, ...
if none it evaluates to 0 (all messages are kept)

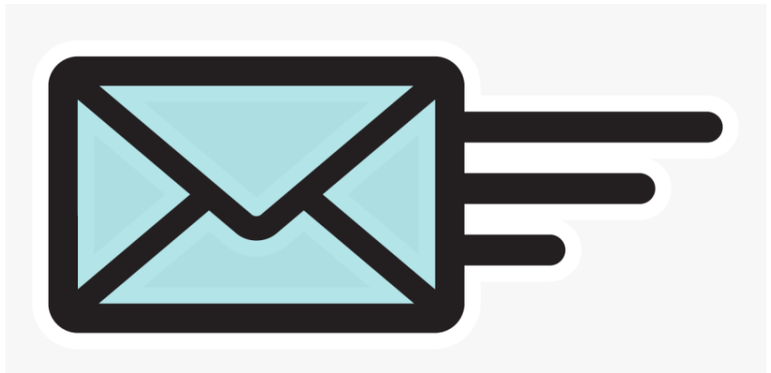
Message passing

`receive … end`



Pid

Pid `!` *message*



Message passing

messages are sent asynchronously
(the sender continues immediately)

any value can be sent as a message

each process has a message queue (mailbox)
no size limit, messages are kept until extracted

a message is received when it is extracted from the mailbox

messages are ordered from oldest to newest in the mailbox

the message that is extracted is not necessarily the oldest
(pattern matching can be used, if there is no match
the receiver suspends and keeps waiting)

Reply

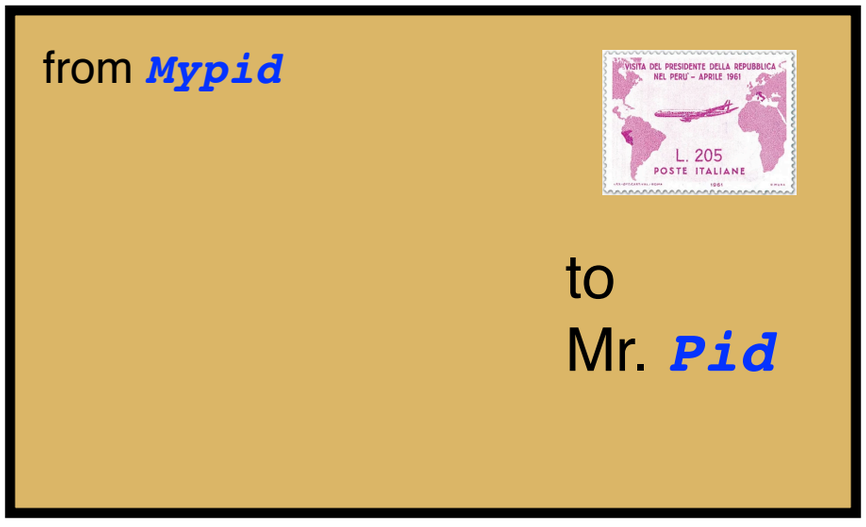
To reply a message, its sender must be known

its pid can be inserted in the message

syntax for tuples

```
Pid ! { Mypid , message }
```

now the receiver *Pid* can reply to *Mypid*



erl session

bruni — beam.smp -- -root /usr/local/Cellar/erlang/21.2.2/lib/erlang -progname erl -- -home ~ -- ▸ erl_child_setup — 106×24

```
Last login: Fri Apr 17 11:42:07 on ttys001
host-131-114-219-127:~ bruni$ erl
Erlang/OTP 21 [erts-10.2.1] [source] [64-bit] [smp:12:12] [ds:12:12:10] [async-threads:1] [hipe] [dtrace]

Eshell V10.2.1  (abort with ^G)
1>
```

erl session

```
Erlang/OTP 26 [erts-14.0.2] [source] [64-bit] [smp:12:12]
[ds:12:12:10] [async-threads:1] [jit:ns] [dtrace]

Eshell V14.0.2 (press Ctrl+G to abort, type help(). for help)

% First of all, Erlang is a functional programming language.

% Expressions must be terminated with a period
% followed by whitespace (line break, a space etc.),
% otherwise they won't be executed.
% Legacy: Erlang was implemented directly in Prolog.

1> 2 + 15.
17
2> 49 * 100.
4900
3> 5 / 2.
2.5
4> 5 div 2.
2
5> 5 rem 2.
1
```

erl session

```
% NUMBERS
% - integers
% - floating point (64-bit IEEE 754-1985 double precision)
% mathematical operators will handle transparently for you.

% ATOMS
% Atoms are literals, constants with their own name for value.
% You can't play with it,
% you can't change it,
% you can't smash it to pieces
% An atom should be enclosed in single quotes (')
% if it does not begin with a lower-case letter
% or if it contains characters other than alphanumeric, _, @
% an atom with single quotes is exactly the same as a
% similar atom without them.

6> monday.
monday
7> 'monday'.
monday
8> 'hello \n my friend'.
'hello \n my friend'
```

erl session

```
% Some atoms are reserved words. These are:
% after                                case                                of
% and                                  catch                               or
% andalso                              cond                                orelse
% band                                 div                                 query
% begin                                end                                 receive
% bnot                                 fun                                 rem
% bor                                  if                                  try
% bsl                                  let                                 when
% bsr                                  not                                 xor
% bxor
%
% TERMS
% * Constant data types
%   - Numbers
%   - Atoms
%   - Pids (process identifiers)
%   - References (for storing system unique references)
%
% * Compound data types
%   - Tuples (fixed number of comma separated terms, curly brackets)
%   - Lists (variable number of comma separated terms, square brackets)
```

erl session

```
% Tuples (fixed number of comma separated terms, curly brackets)

9> {a,b,c}.
{a,b,c}

% Lists (variable number of comma separated terms, square brackets)

10> [a,b,c].
[a,b,c]

% Strings

11> "abc".
"abc"
12> [97,98,99].
"abc"
13> [97,98,99,7].
[97,98,99,7]
```

erl session

```
% BOOLEAN ALGEBRA: true and false are just atoms!
% and / or will always evaluate both arguments.
% andalso / orelse evaluates the second argument if needed.

14> true.
true
15> false.
false
16> true and false.
false
17> false or true.
true
18> not false.
true
19> true xor false.
true
20> true and pippo.
** exception error: bad argument
     in operator  and/2
        called as true and pippo
21> true orelse pippo.
true
22> false andalso pippo.
false
```

erl session

```
% VARIABLES
% variables begin with an uppercase letter or _
% use _ when you do not care about their value
%
% the variable _ won't ever store any value
%
% you can assign a value to a variable exactly once;
% then you can 'pretend' to assign a value to a variable
% if it's the same value it already has.
% If it's different, Erlang will complain.
```

```
23> One.
* 1:1: variable 'One' is unbound
24> One = 1.
1
25> One = 2.
** exception error: no match of right hand side value 2
```

erl session

```
% The = operator has the role of comparing values
% and complaining if they're different.
% If they're the same, it returns the value
%
% The following parses Un = (Ein = (Uno = (One = 1))).
26> Un = Ein = Uno = One = 1.
1

% to display all variable bindings: b()
27> b().
Ein = 1
One = 1
Un = 1
Uno = 1
ok

% To 'erase' the value assigned to a variable: f(.)
28> f(Ein).
ok
29> b().
One = 1
Un = 1
Uno = 1
ok
```

erl session

```
30> Two = One + One.
2
31> Two = 2.
2

% To clear all variable names: f()

32> f().
ok
33> b().
ok
```

erl session

```
% To see the list of commands: help()

34> help().
** shell internal commands **
b()         -- display all variable bindings
e(N)        -- repeat the expression in query <N>
f()         -- forget all variable bindings
f(X)        -- forget the binding of variable X
h()         -- history
h(Mod)      -- help about module
h(Mod,Func)-- help about function in module
h(Mod,Func,Arity) -- help about function with arity in module
ht(Mod)     -- help about a module's types
ht(Mod,Type) -- help about type in module
ht(Mod,Type,Arity) -- help about type with arity in module
hcb(Mod)    -- help about a module's callbacks
hcb(Mod,CB) -- help about callback in module
hcb(Mod,CB,Arity) -- help about callback with arity in module
history(N) -- set how many previous commands to keep
results(N) -- set how many previous command results to keep
catch_exception(B) -- how exceptions are handled
v(N)        -- use the value of query <N>
rd(R,D)     -- define a record
rf()        -- remove all record information
rf(R)       -- remove record information about R
rl()        -- display all record information
rl(R)       -- display record information about R
rp(Term)    -- display Term using the shell's record information
rr(File)    -- read record information from File (wildcards allowed)
rr(F,R)     -- read selected record information from file(s)
rr(F,R,O)   -- read selected record information with options
```

erl session

```
** commands in module c **
bt(Pid)     -- stack backtrace for a process
c(Mod)      -- compile and load module or file <Mod>
cd(Dir)     -- change working directory
flush()     -- flush any messages sent to the shell
help()      -- help info
h(M)        -- module documentation
h(M,F)      -- module function documentation
h(M,F,A)    -- module function arity documentation
i()         -- information about the system
ni()        -- information about the networked system
i(X,Y,Z)    -- information about pid <X,Y,Z>
l(Module)   -- load or reload module
lm()        -- load all modified modules
lc([File]) -- compile a list of Erlang modules
ls()        -- list files in the current directory
ls(Dir)     -- list files in directory <Dir>
m()         -- which modules are loaded
m(Mod)      -- information about module <Mod>
mm()        -- list all modified modules
memory()    -- memory allocation information
memory(T)   -- memory allocation information of type <T>
nc(File)    -- compile and load code in <File> on all nodes
nl(Module) -- load module on all nodes
pid(X,Y,Z) -- convert X,Y,Z to a Pid
pwd()       -- print working directory
q()         -- quit - shorthand for init:stop()
regs()      -- information about registered processes
nregs()     -- information about all registered processes
uptime()    -- print node uptime
xm(M)       -- cross reference check a module
y(File)     -- generate a Yecc parser
** commands in module i (interpreter interface) **
ih()        -- print help for the i module
true
```

erl session

```
% TERM COMPARISON
% greater than (coerce integers to float if needed)
35> 4 > 3.
true

% less than (coerce integers to float if needed)
36> 4 < 5.
true

% greater than or equal to (coerce integers to float if needed)
37> 4 >= 3.
true

% equal to or less than !! (coerce integers to float if needed)
% (be careful: source of many syntax errors!!)
38> 4 <= 5.
* 1:3: syntax error before: '<='
38> 4 =< 5.
true
```

erl session

```
% equal to (coerce integers to float if needed)
39> 3 == 3.
true

% not equal to (coerce integers to float if needed)
40> 3 /= 3.
false

% equal to (coerce integers to float if needed)
41> 3 == 3.0.
true

% not equal to (coerce integers to float if needed)
42> 3 /= 3.0.
false

% exact equal to (no coercion)
43> 3 =:= 3.0.
false

% exact not equal to (no coercion)
44> 3 =/= 3.0.
true
```

erl session

```
% Erlang will let you compare anything with anything.
% Erlang has no such things as boolean true and false.
% The terms true and false are atoms
%
% The ordering of each element in a comparison is the following:
%
% number < atom  < fun  < pid < tuple < list
%
% Tuples are ordered first by their size then by their elements.
% Lists are ordered by comparing heads, then tails.
```

```
45> 1 < a.
true
46> a < {a}.
true
47> {a} < [a].
true

% Full order
%
% number < atom < reference < fun < port < pid < tuple < list
% < bit string
```

erl session

```
% PATTERN MATCHING
% Pattern = Expression

48> X=10, Y=4.
4
49> Point = {X,Y}.
{10,4}
50> {A,B} = Point.
{10,4}
51> b().
A = 10
B = 4
Point = {10,4}
X = 10
Y = 4
ok
52> {A1,_} = Point.
{10,4}
53> A1.
10
54> {_,_} = {1,2,3}.
** exception error: no match of right hand side value {1,2,3}
```

erl session

```
% LISTS
% Lists can contain anything!

55> [1, 2, 3, {numbers,[4,5,6]}, 5.34, atomo].
[1,2,3,{numbers,[4,5,6]},5.34,atomo]

% concatenation: ++ operator
56> [1,2,3] ++ [4,5].
[1,2,3,4,5]

% The operator -- removes elements from a list:
57> [1,2,3,4,5] -- [1,4].
[2,3,5]
58> [1,2,3,1,2,3] -- [1,1,2].
[3,2,3]
59> [1,2,3,1,2,3] -- [1,1,2,5].
[3,2,3]
60> [1,2,3] -- [1,2] -- [3].
[3]
% the above reads [1,2,3] -- ([1,2] -- [3]).
61> ([1,2,3] -- [1,2]) -- [3].
[]
```

erl session

```
% BIF (built-in functions)
% head of a list
62> hd([1,2,3,4,5]).
1

% tail of a list
63> tl([1,2,3,4,5]).
[2,3,4,5]

% length of a list
64> length([1,2,3,4,5]).
5

% pattern matching on lists [Head|Tail]
65> List = [2,3,4].
[2,3,4]
66> NewList = [1|List].
[1,2,3,4]
67> [Head|Tail] = NewList.
[1,2,3,4]
```

erl session

```
68> b().
A = 10
A1 = 10
B = 4
Head = 1
List = [2,3,4]
NewList = [1,2,3,4]
Point = {10,4}
Tail = [2,3,4]
X = 10
Y = 4
ok
```

erl session

```
% The | we used is named the cons operator (constructor).
% In fact, any list can be built with only cons and values:

69> [1 | [2 | [3 | [] ] ] ].
[1,2,3]
70> [1|2].
[1|2]
71> [I,J] = [1|[2]].
[1,2]
72> [I,J] = [1|2].
** exception error: no match of right hand side value [1|2]
73> [I1,J1] = [1|2].
** exception error: no match of right hand side value [1|2]
74> [I1|J1] = [1|2].
[1|2]

% [1 | 2] gives what we call an 'improper list':
% syntactically valid, are of very limited use.
```

erl session

```
75> b().
A = 10
A1 = 10
B = 4
Head = 1
I = 1
I1 = 1
J = 2
J1 = 2
List = [2,3,4]
NewList = [1,2,3,4]
Point = {10,4}
Tail = [2,3,4]
X = 10
Y = 4
ok
```

erl session

```
% List Comprehensions: use ||

76> [2*N || N <- [1,2,3]].
[2,4,6]
77> Z = [ {X,Y} || X <- [a,b,c], Y <- [1,2,3,4] ].
[{a,1},
 {a,2},
 {a,3},
 {a,4},
 {b,1},
 {b,2},
 {b,3},
 {b,4},
 {c,1},
 {c,2},
 {c,3},
 {c,4}]
78> [ X || {a,X} <- Z ].
[1,2,3,4]
```

erl session

```
% -module(name).
% This is always the first attribute and statement of a file.
% it's the name of the current module, where name is an atom.
% This is the name you'll use to call functions from other modules.
% The calls are made with the m:f(a) form,
% where m is the module name, f the function, and a the arguments.

-module(adders).
-export([add/3,add/4]).

add(X,Y) -> X + Y.
add(X,Y,Z) -> X + Y + Z.
add(X,Y,Z,U) -> add(X,Y) + add(Z,U).
```

adders.erl

```
1 -module(adders).
2 -export([add/3,add/4]).
3
4 add(X,Y) -> X + Y.
5
6 add(X,Y,Z) -> X + Y + Z.
7
8 add(X,Y,Z,U) -> add(X,Y) + add(Z,U).
9
```

```
% Erlang code is compiled to bytecode to be used by the VM.
% You can call the compiler from many places:
% $ erlc flags file.erl when in the command line,
% compile:file(FileName) when in the shell or in a module,
% c(FileName) when in the shell, etc.

-module(functions).
-compile(export_all).
% export all functions defined in the module
```

erl session

adders.erl

```erlang
-module(adders).
-export([add/3,add/4]).

add(X,Y) -> X + Y.

add(X,Y,Z) -> X + Y + Z.

add(X,Y,Z,U) -> add(X,Y) + add(Z,U).
```

```
83> c(adders).
{ok,adders}
84> adders:add(1,2,3).
6
```

erl session

functions.erl

```erlang
-module(functions).
-compile(export_all).

hello() -> io:format("Hello, world!~n").

```

```
85> c(functions).
functions.erl:2:2: Warning: export_all flag enabled - all functions
will be exported
{ok,functions}
```

erl session

```
19  greet(male, Name) ->
20      io:format("Hello, Mr. ~s!~n", [Name]);
21  greet(female, Name) ->
22      io:format("Hello, Mrs. ~s!~n", [Name]);
23  greet(_, Name) ->
24      io:format("Hello, ~s!~n", [Name]).
```

```
% Defining functions with pattern matching
% Each of these function declarations is called a function clause.
% Function clauses must be separated by semicolons (;)
% together form a function declaration.
%
% the clauses are scanned sequentially, in the order in which occur
% until the first one of them matches the call.
% When a match occurs,
% the expression on the right-hand side of '->' is evaluated,
% variables occurring in the function definition are substituted
% in the right-hand side of the clause before it is evaluated.
```

erl session

```
19  greet(male, Name) ->
20      io:format("Hello, Mr. ~s!~n", [Name]);
21  greet(female, Name) ->
22      io:format("Hello, Mrs. ~s!~n", [Name]);
23  greet(_, Name) ->
24      io:format("Hello, ~s!~n", [Name]).
```

```
% io:format alike printf
% In the string ~ denotes a token
% ~~ is ~
% ~n is line break
% ~s is a string or a bitstring
% ~p prints an erlang term in a nice way (with indentation)
```

erl session

```
19  greet(male, Name) ->
20      io:format("Hello, Mr. ~s!~n", [Name]);
21  greet(female, Name) ->
22      io:format("Hello, Mrs. ~s!~n", [Name]);
23  greet(_, Name) ->
24      io:format("Hello, ~s!~n", [Name]).
```

```
86> functions:greet(male,andrea).
Hello, Mr. andrea!
ok
87> functions:greet(female,andrea).
Hello, Mrs. andrea!
ok
88> functions:greet(child,andrea).
Hello, andrea!
ok
```

erl session

```
36  % define second element of a list
37  second([_,X|_]) -> X.
```

```
89> functions:second([1,2]).
2
90> functions:second([2]).
** exception error: no function clause matching
functions:second([2]) (functions.erl, line 37)
```

erl session

```
137  %%% Higher Order
138  one() -> 1.
139  two() -> 2.
140  % functions:one() .
141  % functions:two() .
142
143  add(X,Y) -> X() + Y().
144  % functions:add(fun functions:one/0 , fun functions:two/0).
```

```
95> functions:add(fun functions:one/0 , fun functions:two/0).
3
```

erl session

```
146  apply(F,X) -> F(X).
```

```
96> Succ = fun(X) -> X+1 end.
#Fun<erl_eval.42.125776118>
97> functions:apply(Succ,3).
4
98> lists:map(Succ,[1,2,3]).
[2,3,4]
```

erl session

recursion.erl

recursion.erl

```
1  -module(recursion).
2  -compile(export_all).
3
4  %% EXAMPLE: Factorial with different syntax
5
6  % with GUARDS and PATTERN MATCHING
7  fac(0) -> 1;
8  fac(N) when N > 0  -> N*fac(N-1);
9  fac(_) -> 0.
10
11 % with IFs
12 fact(N) -> if N == 0 -> 1
13               ; N > 0  -> N*fact(N-1)
14               ; true -> 0
15            end .
16
17 % TAIL RECURSION
18 tail_fac(N) -> tail_fac(N,1).
19
20 tail_fac(0,Acc) -> Acc;
21 tail_fac(N,Acc) when N > 0 -> tail_fac(N-1,N*Acc);
22 tail_fac(_,_) -> 0.
```

erl session

```
26  %% EXAMPLE: permutations
27
28  perms([]) -> [[]];
29  perms(L)  -> [[H|T] || H <- L, T <- perms(L--[H])].
```

```
99> c(recursion).
recursion.erl:2:2: Warning: export_all flag enabled - all functions
will be exported
{ok,recursion}
100> recursion:perms("abc").
["abc","acb","bac","bca","cab","cba"]
101> recursion:perms("abcdef").
["abcdef","abcdfe","abcedf","abcefd","abcfde","abcfed",
 "abdcef","abdcfe","abdecf","abdefc","abdfce","abdfec",
 "abecdf","abecfd","abedcf","abedfc","abefcd","abefdc",
 "abfcde","abfced","abfdce","abfdec","abfecd","abfedc",
 "acbdef","acbdfe","acbedf","acbefd",
 [...]|...]
```

erl session

```
32  %% EXAMPLE: length of a list
33
34  len([]) -> 0;
35  len([_|T]) -> 1 + len(T).
36
37  tail_len(L) -> tail_len(L,0).
38
39  tail_len([],Acc) -> Acc;
40  tail_len([_|T],Acc) -> tail_len(T,Acc+1).
```

```
42  %% EXAMPLE: replicate
43
44  replicate(0,_) -> [];
45  replicate(N,Term) when N > 0 -> [Term|replicate(N-1,Term)].
46
47  tail_replicate(N,Term) -> tail_replicate(N,Term,[]).
48
49  tail_replicate(0,_,List) -> List;
50  tail_replicate(N,Term,List) when N > 0 -> tail_replicate(N-1, Term, [Term|List]).
```

```
52  %% EXAMPLE: reverse
53
54  reverse([]) -> [];
55  reverse([H|T]) -> reverse(T)++[H].
56  % costs too much!!
57
58  tail_reverse(L) -> tail_reverse(L,[]).
59
60  tail_reverse([],Acc) -> Acc;
61  tail_reverse([H|T],Acc) -> tail_reverse(T, [H|Acc]).
```

erl session

```
% In Erlang, processes belong to the programming language
% and NOT the operating system.

% In Erlang:
% - Creating and destroying processes is very fast.
% - Sending messages between processes is very fast.
% - Processes behave the same way on all operating systems.
% - We can have very large numbers of processes.
% - Processes share no memory and are completely independent.
% - The only way for processes to interact is via message passing.
% For these reasons Erlang is a pure message passing language.
%
% Erlang uses the actor model:
% each actor is a separate process in the virtual machine.
% In Erlang everyone communicates by writing letters and that's it.
% If you were an actor in Erlang's world,
% you would be a lonely person,
% - sitting in a dark room with no window,
% - waiting by your mailbox to get a message.
% Once you get a message,
% - you react to it in a specific way:
% you pay the bills when receiving them,
% you respond to Birthday cards with a "Thank you" letter
% and you ignore the letters you can't understand.
```

erl session

```
% If the shell freezes: ^G
%
% - type in i then c
%   Erlang should stop the currently running code
%   and bring you back to a responsive shell
%
% - j will give you a list of processes running
%   current job marked by a star
%
% - k kills the shell as it is instead of just interrupting it.
%
% - Press s to start a new one.
```

erl session

```
% To start a new process
% - spawn/1, takes a single function and runs it
%   once it's done, it disappears.

102> F = fun() -> 2 + 2 end.
#Fun<erl_eval.43.125776118>
103> spawn(F).
<0.210.0>

% we get a process id
% <0.210.0>
% we can't see the result of the function F.
% We only get its pid.
% That's because processes do not return anything.
% to get the pid of the current process

104> self().
<0.192.0>
```

erl session

```
105> G = fun (X) ->
  ..       timer:sleep(200-X),
  ..       io:format("~p says ~p~n",[self(),X])
  ..       end.
#Fun<erl_eval.42.125776118>
106> G(5).
<0.192.0> says 5
ok
107> [spawn(fun() -> G(X) end) || X <- lists:seq(1,30)].
[<0.215.0>,<0.216.0>,<0.217.0>,<0.218.0>,<0.219.0>,
 <0.220.0>,<0.221.0>,<0.222.0>,<0.223.0>,<0.224.0>,<0.225.0>,
 <0.226.0>,<0.227.0>,<0.228.0>,<0.229.0>,<0.230.0>,<0.231.0>,
 <0.232.0>,<0.233.0>,<0.234.0>,<0.235.0>,<0.236.0>,<0.237.0>,
 <0.238.0>,<0.239.0>,<0.240.0>,<0.241.0>,<0.242.0>,<0.243.0>|...]
<0.244.0> says 30
<0.243.0> says 29
<0.240.0> says 26
<0.242.0> says 28
<0.241.0> says 27
...

% The order doesn't make sense. Welcome to concurrency
```

erl session

```
% to send a message use the bang symbol !
% Pid ! Msg

109> self() ! memo.
memo

% you can send the same message to multiple recipients

110> self() ! self() ! self() ! memo.
memo

% to receive messages (but X was already bound to 10 earlier...)

111> receive X -> io:format("got ~p~n",[X]) end.
^G
User switch command (type h for help)
 --> i

 --> c

** exception exit: killed
```

erl session

```
% Let us retry

112> f().
ok
113> self() ! self() ! self() ! hello.
hello
114> receive X -> io:format("got ~p~n",[X]) end.
got hello
ok
115> receive X -> io:format("got ~p~n",[X]) end.
got hello
ok
116> receive X -> io:format("got ~p~n",[X]) end.
got hello
ok
117> receive X -> io:format("got ~p~n",[X]) end.
^G
User switch command (type h for help)
 --> i

 --> c

** exception exit: killed
```

erl session

```
% to see and empty the mailbox: flush()

119> self() ! self() ! self() ! self() ! self() ! bye.
bye
120> flush().
Shell got bye
Shell got bye
Shell got bye
Shell got bye
Shell got bye
ok
```

erl session

counter.erl

counter.erl

```erlang
-module(counter).
-export([start/0,loop/1]).

start() ->
% Pid = spawn(Module,Function,Args)
    spawn(counter, loop, [0]).

loop(Val) ->
    receive
        inc -> loop(Val + 1); % tail recursive call
        reset -> loop(0);  % tail recursive call
        val -> io:format("Val is ~p~n",[Val]),
               loop(Val); % tail recursive call
        stop -> io:format("~p says bye!~n",[self()]);  % NO recursive call
        _ -> loop(Val) % other messages are discarded
    end.
```

```
121> c(counter).
{ok,counter}
122> counter:start().
<0.317.0>
123> Cnt = counter:start().
<0.319.0>
```

erl session

```
123> Cnt = counter:start().
<0.319.0>
124> Cnt ! val.
Val is 0
val
125> Cnt ! inc.
inc
126> Cnt ! Cnt ! Cnt ! Cnt ! inc.
inc
127> Cnt ! val.
Val is 5
val
128> Cnt ! reset.
reset
129> Cnt ! val.
Val is 0
val
130> Cnt ! stop.
<0.319.0> says bye!
stop
131> Cnt ! stop.
stop
132> Cnt ! stop.
stop
133> Cnt ! val.
val
```

counter.erl

```erlang
-module(counter).
-export([start/0,loop/1]).

start() ->
% Pid = spawn(Module,Function,Args)
    spawn(counter, loop, [0]).

loop(Val) ->
    receive
        inc -> loop(Val + 1); % tail recursive call
        reset -> loop(0);  % tail recursive call
        val -> io:format("Val is ~p~n",[Val]),
               loop(Val); % tail recursive call
        stop -> io:format("~p says bye!~n",[self()]);  % NO recursive call
        _ -> loop(Val) % other messages are discarded
    end.
```

erl session

```
134> c(icounter).
{ok,icounter}
135> Icnt = icounter:start().
<0.336.0>
136> icounter:inc(Icnt).
true
137> icounter:inc(Icnt).
true
138> icounter:inc(Icnt).
true
139> icounter:inc(Icnt).
true
140> icounter:inc(Icnt).
true
141> icounter:inc(Icnt).
true
142> icounter:val(Icnt).
6
143> icounter:reset(Icnt).
true
144> icounter:val(Icnt).
0
145> icounter:stop(Icnt).
<0.336.0> says bye!
true
146> icounter:inc(Icnt).
true
147> icounter:val(Icnt).
^G
User switch command (type h for help)
 --> i

 --> c

** exception exit: killed
```

icounter.erl

```erlang
-module(icounter).
-export([start/0,loop/1,inc/1,val/1,reset/1,stop/1]).

start() ->
    spawn(icounter, loop, [0]).

loop(Val) ->
    receive
        inc -> loop(Val + 1);
        reset -> loop(0);
        {From,val} -> From ! {self(),Val}, % RECEIVE AND REPLY
                      loop(Val);
        stop -> io:format("~p says bye!~n",[self()]);
        _ -> loop(Val)
    end.

% functions over counters
inc(Counter) ->
    Counter ! inc,
    true .

val(Counter) ->
    Counter ! {self(),val}, % SEND REQUEST
    receive
        {Counter,Value} -> Value; % WAIT FOR REPLY
        _ -> true
    end.

reset(Counter) ->
    Counter ! reset,
    true .

stop(Counter) ->
    Counter ! stop,
    true .
```

erl session

alarm.erl

alarm.erl

```
1  -module(alarm).
2  -export([start/2,cancel/1,alarm/3,sleep/1,flush/0,priority_rec/0]).
3
4  start(Time, Alarm) ->
5      spawn(alarm, alarm, [self(),Time,Alarm]).
6
7  cancel(Timer) ->
8      Timer ! {self(),cancel}.
9
10 alarm(Pid, Time, Alarm) ->
11     receive
12         {Pid,cancel} -> true
13     % DO SOMETHING IF NO MESSAGE ARRIVES IN TIME
14     after Time -> Pid ! Alarm
15     % special values:
16     % infinity: timeout will never occur
17     %           useful if time is computed at runtime
18     % 0: tries all messages currently in the mailbox first
19     end.
20
21 % c(alarm).
22 % alarm:start(10000,wake_up).
23 % receive A -> A end.
24 %
25 % Al = alarm:start(30000,wake_up).
26 % alarm:cancel(Al).
```

erl session

alarm.erl

alarm.erl

```erlang
-module(alarm).
-export([start/2,cancel/1,alarm/3,sleep/1,flush/0,priority_rec/0]).

start(Time, Alarm) ->
    spawn(alarm, alarm, [self(),Time,Alarm]).

cancel(Timer) ->
    Timer ! {self(),cancel}.

alarm(Pid, Time, Alarm) ->
    receive
        {Pid,cancel} -> true
    % DO SOMETHING IF NO MESSAGE ARRIVES IN TIME
    after Time -> Pid ! Alarm
    % special values:
    % infinity: timeout will never occur
    %           useful if time is computed at runtime
    % 0: tries all messages currently in the mailbox first
    end.

% c(alarm).
% alarm:start(10000,wake_up).
% receive A -> A end.
%
% Al = alarm:start(30000,wake_up).
% alarm:cancel(Al).
```

```
148> c(alarm).
{ok,alarm}
149> alarm:start(10000,wake_up).
<0.356.0>
150> receive A -> A end.
wake_up
```

erl session

```
28  %% EXAMPLES with timeouts
29
30  sleep(Time) ->
31      receive
32      after Time -> true
33      end.
34  % alarm:sleep(5000).
35
36
37
38  flush() ->
39      receive
40          Any -> io:format("flush ~p~n",[Any]),
41                 flush()
42      after 0 -> true
43      end.
44  % self() ! one.
45  % self() ! two.
46  % self() ! three.
47  % self() ! four.
48  % alarm:flush().
```

erl session

```
151> self() ! {good,goal}.
{good,goal}
152> self() ! {good,win}.
{good,win}
153> self() ! {bad,loose}.
{bad,loose}
154> self() ! {good,baby_born}.
{good,baby_born}
155> alarm:priority_rec().
loose
156> alarm:priority_rec().
goal
157> alarm:priority_rec().
win
158> alarm:priority_rec().
baby_born
159> alarm:priority_rec().
^G
User switch command (type h for help)
 --> i

 --> c
** exception exit: killed
160> halt().
```

```
50  priority_rec() ->
51      receive
52          {bad,News} -> News
53      after 0 ->
54          receive
55              {good,News} -> News
56          end
57      end.
```

Exercises

[**Ex. 1**] Write a server in erlang to convert temperatures from Celsius degrees to Fahrenheit degrees and vice versa, using the formula $F = 1.8C + 32$. The server receives requests of the form $(Pid, \texttt{cs}, C)$ or $(Pid, \texttt{ft}, F)$ and replies to Pid by sending messages in analogous format. The server can be stopped by sending the message `stop`. All the other messages are ignored. Spawn a copy of the server, send it some temperatures to convert, check out the results and stop the server.

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive

end.
```

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive
      {Pid,cs,C} ->
```

```
end.
```

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive
      {Pid,cs,C} -> Pid ! {self(),ft,(1.8 * C) + 32},
                    convert();

end.
```

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive
      {Pid,cs,C} -> Pid ! {self(),ft,(1.8 * C) + 32},
                    convert();
      {Pid,ft,F} ->


end.
```

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive
      {Pid,cs,C} -> Pid ! {self(),ft,(1.8 * C) + 32},
                    convert();
    {Pid,ft,F} -> Pid ! {self(),cs,(F - 32) / 1.8},
                    convert();


end.
```

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive
      {Pid,cs,C} -> Pid ! {self(),ft,(1.8 * C) + 32},
                    convert();
      {Pid,ft,F} -> Pid ! {self(),cs,(F - 32) / 1.8},
                    convert();
      stop -> true;

end.
```

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive
      {Pid,cs,C} -> Pid ! {self(),ft,(1.8 * C) + 32},
                    convert();
      {Pid,ft,F} -> Pid ! {self(),cs,(F - 32) / 1.8},
                    convert();
      stop -> true;
      _ ->
    end.
```

Ex. 1, temp converter

```
-module(ex1).
-export([convert/0]).

convert() ->
    receive
      {Pid,cs,C} -> Pid ! {self(),ft,(1.8 * C) + 32},
                    convert();
      {Pid,ft,F} -> Pid ! {self(),cs,(F - 32) / 1.8},
                    convert();
      stop -> true;
      _ -> convert()
    end.
```

Ex. 1, temp converter

```
Eshell V10.2.1  (abort with ^G)
1> c(ex1).
{ok,ex1}
2>
```

Ex. 1, temp converter

```
Eshell V10.2.1  (abort with ^G)
1> c(ex1).
{ok,ex1}
2> Conv = spawn(ex1,convert,[]).
<0.84.0>
3>
```

Ex. 1, temp converter

```
Eshell V10.2.1  (abort with ^G)
1> c(ex1).
{ok,ex1}
2> Conv = spawn(ex1,convert,[]).
<0.84.0>
3> Conv ! {self(),cs,23}.
{<0.77.0>,cs,23}
4>
```

Ex. 1, temp converter

```
Eshell V10.2.1  (abort with ^G)
1> c(ex1).
{ok,ex1}
2> Conv = spawn(ex1,convert,[]).
<0.84.0>
3> Conv ! {self(),cs,23}.
{<0.77.0>,cs,23}
4> receive
4>     {Conv,ft,F} -> io:format("23 celsius = ~p fahrenheit~n",[F])
4> end.
23 celsius = 73.4 fahrenheit
ok
5>
```

Ex. 1, temp converter

```
Eshell V10.2.1  (abort with ^G)
1> c(ex1).
{ok,ex1}
2> Conv = spawn(ex1,convert,[]).
<0.84.0>
3> Conv ! {self(),cs,23}.
{<0.77.0>,cs,23}
4> receive
4>     {Conv,ft,F} -> io:format("23 celsius = ~p fahrenheit~n",[F])
4> end.
23 celsius = 73.4 fahrenheit
ok
5> Conv ! {self(),ft,74}.
{<0.77.0>,ft,74}
6>
```

Ex. 1, temp converter

```
Eshell V10.2.1  (abort with ^G)
1> c(ex1).
{ok,ex1}
2> Conv = spawn(ex1,convert,[]).
<0.84.0>
3> Conv ! {self(),cs,23}.
{<0.77.0>,cs,23}
4> receive
4>     {Conv,ft,F} -> io:format("23 celsius = ~p fahrenheit~n",[F])
4> end.
23 celsius = 73.4 fahrenheit
ok
5> Conv ! {self(),ft,74}.
{<0.77.0>,ft,74}
6> receive
6>     {Conv,cs,C} -> io:format("74 fahrenheit = ~p celsius~n",[C])
6> end.
74 fahrenheit = 23.333333333333332 celsius
ok
7>
```

Ex. 1, temp converter

```
Eshell V10.2.1  (abort with ^G)
1> c(ex1).
{ok,ex1}
2> Conv = spawn(ex1,convert,[]).
<0.84.0>
3> Conv ! {self(),cs,23}.
{<0.77.0>,cs,23}
4> receive
4>     {Conv,ft,F} -> io:format("23 celsius = ~p fahrenheit~n",[F])
4> end.
23 celsius = 73.4 fahrenheit
ok
5> Conv ! {self(),ft,74}.
{<0.77.0>,ft,74}
6> receive
6>     {Conv,cs,C} -> io:format("74 fahrenheit = ~p celsius~n",[C])
6> end.
74 fahrenheit = 23.333333333333332 celsius
ok
7> Conv ! stop.
stop
8>
```

[**Ex. 2**] Write an erlang function `copy` that receives an integer n and if n is positive it prints n copies of n (one per line). Write an erlang function that receives a list of integers and spawn an instance of `copy` for each integer in the list.

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 ->
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) ->
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) -> true.
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) -> true.

copy(N,M) when N > 0 ->
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) -> true.

copy(N,M) when N > 0 -> io:format("~p~n",[M]),
                        copy(N-1,M);
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) -> true.

copy(N,M) when N > 0 -> io:format("~p~n",[M]),
                        copy(N-1,M);
copy(_,_) -> true.
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) -> true.

copy(N,M) when N > 0 -> io:format("~p~n",[M]),
                        copy(N-1,M);
copy(_,_) -> true.

listCopy(L) ->
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) -> true.

copy(N,M) when N > 0 -> io:format("~p~n",[M]),
                        copy(N-1,M);
copy(_,_) -> true.

listCopy(L) -> [                              || N <- L ].
```

Ex. 2, copy

```
-module(ex2).
-export([copy/1,listCopy/1]).

copy(N) when N > 0 -> copy(N,N);
copy(_) -> true.

copy(N,M) when N > 0 -> io:format("~p~n",[M]),
                        copy(N-1,M);
copy(_,_) -> true.

listCopy(L) -> [ spawn(ex2,copy,[N]) || N <- L ].
```

Ex. 2, copy

```
Eshell V10.2.1  (abort with ^G)
1> c(ex2).
{ok,ex2}
2>
```

Ex. 2, copy

```
Eshell V10.2.1  (abort with ^G)
1> c(ex2).
{ok,ex2}
2> ex2:listCopy(lists:seq(1,5)).
1
2
3
4
5
[<0.84.0>,<0.85.0>,<0.86.0>,<0.87.0>,<0.88.0>]
2
3
4
5
3
4
5
4
5
5
3>
```

[**Ex. 3**] Write an erlang function `view` that displays the content of the mailbox but makes all messages remain available in the mailbox afterwards.

Ex. 3, view

```
-module(ex3).
-export([view/0          ]).
```

```
view( ) ->
    receive
        Any -> io:format("view ~p~n",[Any]),
               view(         )
    after 0 -> true
    end.
```

Ex. 3, view

```
-module(ex3).
-export([view/0          ]).
```

```
view(L) ->
    receive
        Any -> io:format("view ~p~n",[Any]),
               view([Any|L])
    after 0 -> true
    end.
```

Ex. 3, view

```
-module(ex3).
-export([view/0          ]).
```

```
view() -> view([]);
view(L) ->
    receive
        Any -> io:format("view ~p~n",[Any]),
               view([Any|L])
    after 0 -> true
    end.
```

Ex. 3, view

```
-module(ex3).
-export([view/0          ]).
```

```
view() -> view([]);
view(L) ->
    receive
        Any -> io:format("view ~p~n",[Any]),
               view([Any|L])
    after 0 -> send(L)
    end.
```

Ex. 3, view

```
-module(ex3).
-export([view/0,send/1]).

send([]) -> true;
send([A|L]) -> send(L),
               self() ! A.

view() -> view([]);
view(L) ->
    receive
        Any -> io:format("view ~p~n",[Any]),
               view([Any|L])
    after 0 -> send(L)
    end.
```

Ex. 3, view

```
Eshell V10.2.1  (abort with ^G)
1> c(ex3).
{ok,exercise}
2>
```

Ex. 3, view

```
Eshell V10.2.1  (abort with ^G)
1> c(ex3).
{ok,exercise}
2> ex3:send([3,2,1]).
3
3>
```

Ex. 3, view

```
Eshell V10.2.1  (abort with ^G)
1> c(ex3).
{ok,exercise}
2> ex3:send([3,2,1]).
3
3> ex3:view().
view 1
view 2
view 3
3
4>
```

Ex. 3, view

```
Eshell V10.2.1  (abort with ^G)
1> c(ex3).
{ok,exercise}
2> ex3:send([3,2,1]).
3
3> ex3:view().
view 1
view 2
view 3
3
4> ex3:view().
view 1
view 2
view 3
3
5>
```

Ex. 3, view

```
Eshell V10.2.1  (abort with ^G)
1> c(ex3).
{ok,exercise}
2> ex3:send([3,2,1]).
3
3> ex3:view().
view 1
view 2
view 3
3
4> ex3:view().
view 1
view 2
view 3
3
5> flush().
Shell got 1
Shell got 2
Shell got 3
ok
6>
```

Ex. 3, view

```
Eshell V10.2.1  (abort with ^G)
1> c(ex3).
{ok,exercise}
2> ex3:send([3,2,1]).
3
3> ex3:view().
view 1
view 2
view 3
3
4> ex3:view().
view 1
view 2
view 3
3
5> flush().
Shell got 1
Shell got 2
Shell got 3
ok
6> ex3:view().
true
```