

Test driven development (TDD) in Erlang with EUnit

Module 8 - Erlang tutorial 4

Operating systems and process oriented programming 2018

1DT096

The Erlang "crypto"



Where does the Erlang syntax come from?

Mostly from Prolog. Erlang started life as a modified Prolog. **!** as the send-message operator comes from CSP. Eri Pascal was probably responsible for **,** and **;** being separators and not terminators.

Symbol	Description
=	Match operator
,	Separates expressions
•	Terminates module attributes and function declarations (a.k.a. 'forms')
;	Separates "choices"
->	Separate the head of a clause (for example a function) from the body
!	Sends a message from one process to another
%%	Starts a comment (until end of line)

TDD

Test Driven Development

Unit testing in Erlang with EUnit

Compiling modules

We start with an empty file named `max.erl`

We start the Erlang shell and uses the BIF `c/1` to compile.

```
Terminal — beam.smp — 55x9
beam.smp
karl > erl
Erlang R14B03 (erts-5.8.4) [source] [64-bit] [smp:2:2]
[rq:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.8.4 (abort with ^G)
1> c(max).
./max.erl:5: no module definition
error
2> █
```



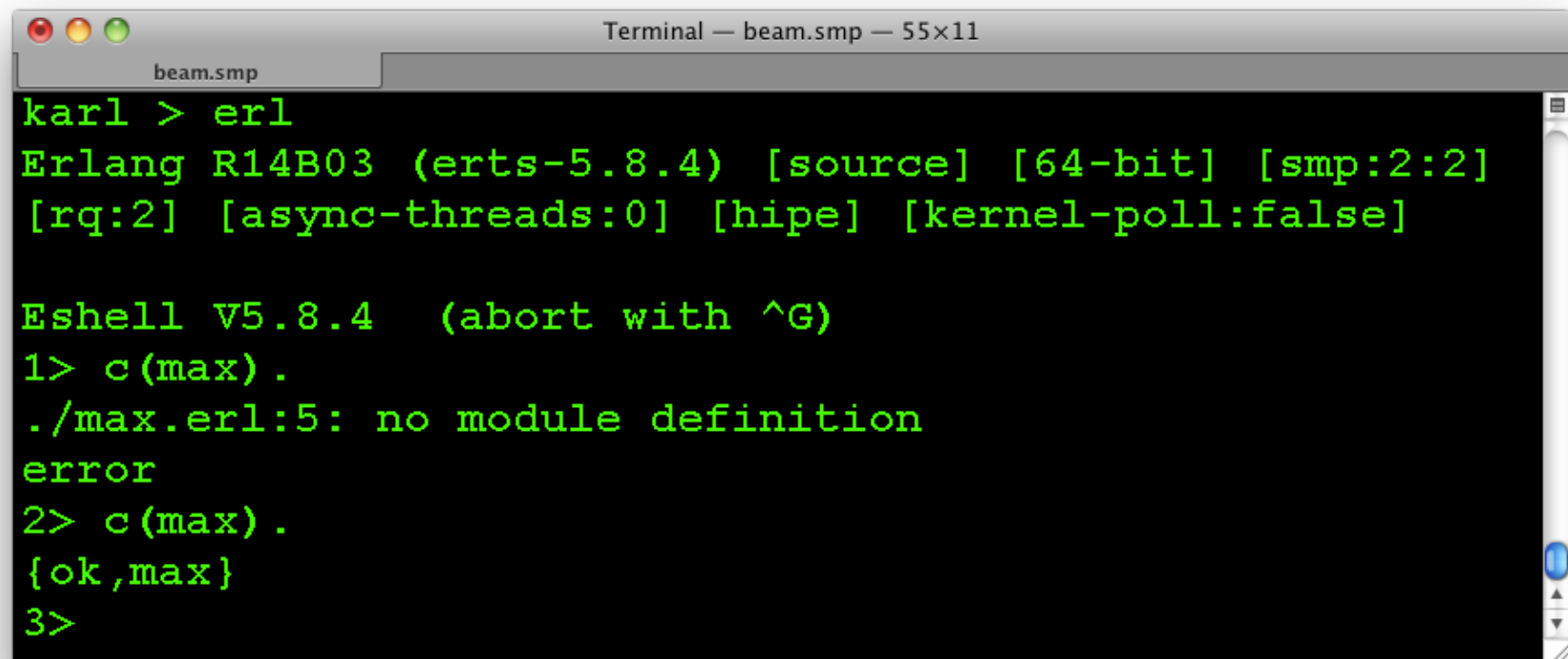
We got a compilation error :-)

Compiling modules

Every Erlang module must start with a **module declaration** defining the name of the module. The module name must be given as an atom and must be the same as the file name minus the extension **.erl**

```
-module(max).
```

Now we can compile the module without errors :-)

A terminal window titled "Terminal — beam.smp — 55x11" with a tab labeled "beam.smp". The terminal shows the following text:

```
karl > erl
Erlang R14B03 (erts-5.8.4) [source] [64-bit] [smp:2:2]
[rq:2] [async-threads:0] [hipe] [kernel-poll:false]

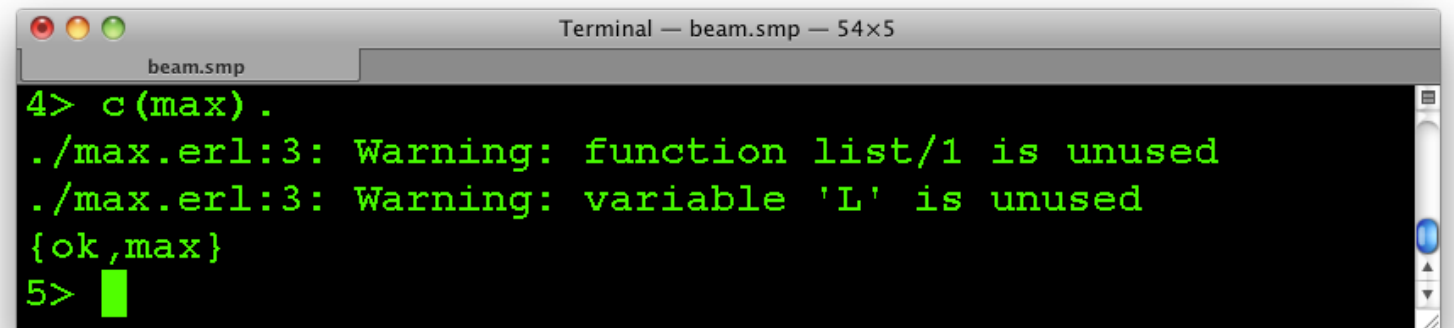
Eshell V5.8.4 (abort with ^G)
1> c(max) .
./max.erl:5: no module definition
error
2> c(max) .
{ok,max}
3>
```

Compiler warnings

We start by declaring the `max:list/1` function. The intention is for this function to return the max value in a list. Every function must return a value, for now we use the atom `tbi`.

```
-module(max).  
  
list(L) ->  
    tbi. %% To Be Implemented
```

Let's compile!



```
Terminal — beam.smp — 54x5  
beam.smp  
4> c(max).  
./max.erl:3: Warning: function list/1 is unused  
./max.erl:3: Warning: variable 'L' is unused  
{ok,max}  
5> █
```



The Erlang compiler gives two warnings.

Unused functions and variables are often an indication of a mistake and you should get into habit to always get rid of all warnings.

Unused variables and exported functions

To get rid of the warning of list/1 being unused we can export this function. Exported functions can be called from the Erlang shell and from other modules.

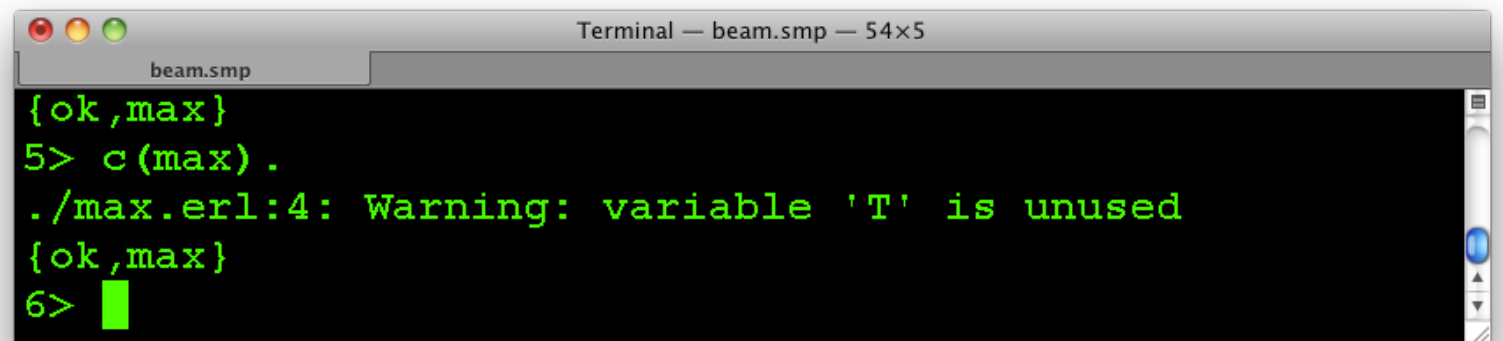
To get rid of the warning about the variable L being unused, we simply return the first element of the list.

```
-export([Name1/Arity1, ..., NameN/ArityN]).
```

The export module attribute takes a **list** as argument.

```
-module(max).  
-export([list/1]).  
  
list([H|T]) ->  
    H.
```

Let's compile!



```
Terminal — beam.smp — 54x5  
beam.smp  
{ok,max}  
5> c(max).  
./max.erl:4: Warning: variable 'T' is unused  
{ok,max}  
6> █
```



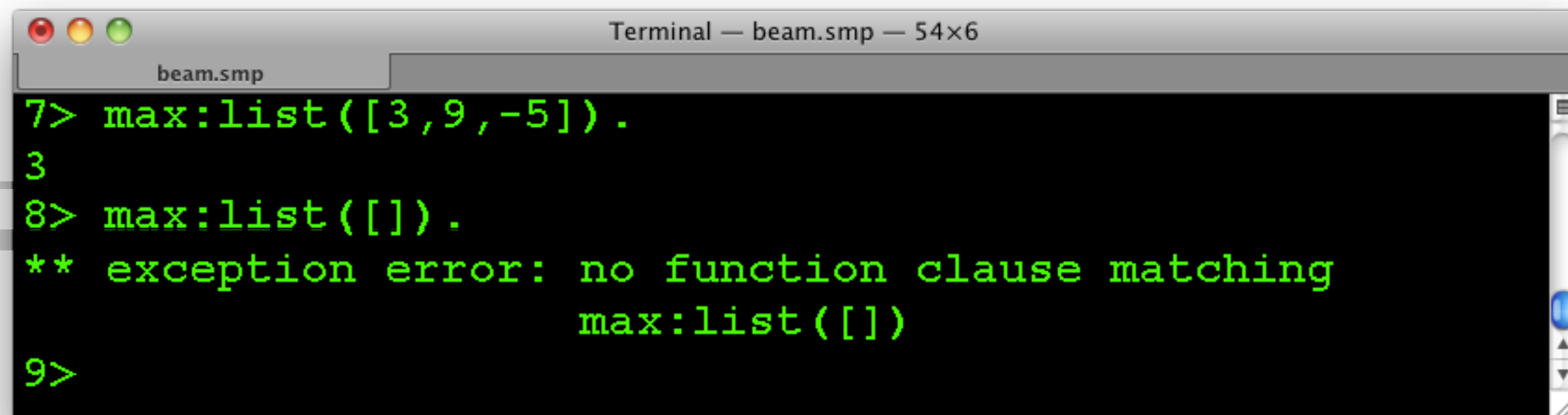
Oops - the variable T is unused...

Intentionally unused variables

By prefixing the T variable with an underscore we get rid of the compiler warning.

```
-module(max).  
-export([list/1]).  
  
list([H|_T]) ->  
    H.
```

Let's compile and test.

A terminal window titled "Terminal — beam.smp — 54x6" with a tab labeled "beam.smp". The terminal shows the following interaction:

```
7> max:list([3,9,-5]).  
3  
8> max:list([]).  
** exception error: no function clause matching  
    max:list([])  
9>
```

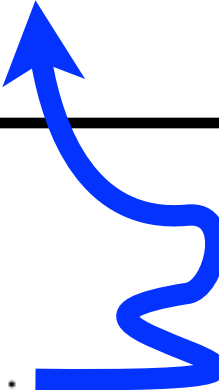


The compiler deduces that our function takes a single list as argument. Since a function always must return a value, we are obliged to handle the case of an empty list. Note - that there is no head element in an empty list.

Test Driven Development (TDD) using EUnit

We include the **EUnit unit test framework** and add a simple test case for the empty list case.

```
-module(max).  
-export([list/1]).  
  
%% To use EUnit we must include this:  
-include_lib("eunit/include/eunit.hrl").  
  
list([H|_T]) ->  
    H.  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%%                               EUnit Test Cases                               %%  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
%% All functions with names ending wiht _test() or _test_() will be  
%% called automatically by max:test()  
  
empty_list_test() ->  
    ?assertEqual({undefined, empty_list}, max:list([])).
```



EUnit provides a number of **?assert** macros.

We use **?assertEqual** to test if **max:list([])** returns the tuple **{undefined, empty_list}**.

If you use multiple **?assert** macros within a **test_()** function, EUnit will stop at the first failure.

Test Driven Development (TDD) using EUnit

Compile and test.

```
Terminal — beam.smp — 60x14
beam.smp
11> c(max) .
{ok,max}
12> max:test().
max: empty_list_test (module 'max')...*failed*
::error:function_clause
  in function max:list/1
    called as list([])
  in call from max:'-empty_list_test/0-fun-0-' /1

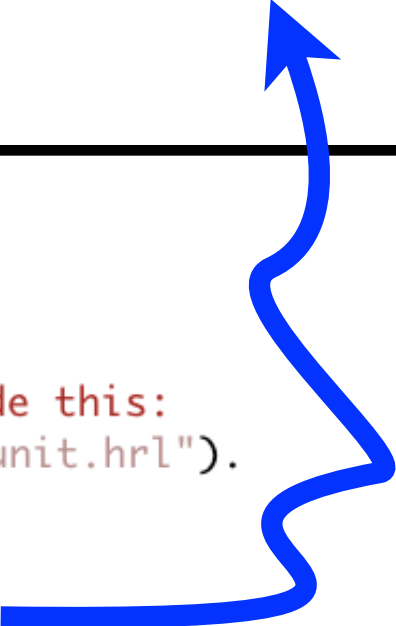
=====
Failed: 1.  Skipped: 0.  Passed: 0.
error
13> █
```

The **function_clause** error means there's no function clause matching **max:list([])**



Code a little ...

We add a function clause for the empty list case.



```
-module(max).
-export([list/1]).

%% To use EUnit we must include this:
-include_lib("eunit/include/eunit.hrl").

list([]) ->
    {undefined, empty_list}.
list([H|_T]) ->
    H.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                EUnit Test Cases                                %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% All functions with names ending wiht _test() or _test_() will be
%% called automatically by fifo:test()

empty_list_test() ->
    ?assertEqual({undefined, empty_list}, max:list([])).
```

Code a little ... test a little

```
Terminal — beam.smp — 60x14
beam.smp
max: empty_list_test (module 'max')...*failed*
::error:function_clause
  in function max:list/1
    called as list([])
  in call from max:'-empty_list_test/0-fun-0-' /1

=====
Failed: 1.  Skipped: 0.  Passed:
error
13> c(max) .
./max.erl:9: function list/1 already defined
error
14> █
```

```
list([]) ->
    {undefined, empty_list}.
list([H|_T]) ->
    H.
```



Oops - when a function has several clauses, semi-colon (;) must be used to separate clauses. The last clause must be ended with a period (.).

Multiple function clauses

We separate the two function clauses with a semi-colon.

```
-module(max).
-export([list/1]).

%% To use EUnit we must include this:
-include_lib("eunit/include/eunit.hrl").

list([]) ->
    {undefined, empty_list};
list([H|_T]) ->
    H.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                               EUnit Test Cases                               %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

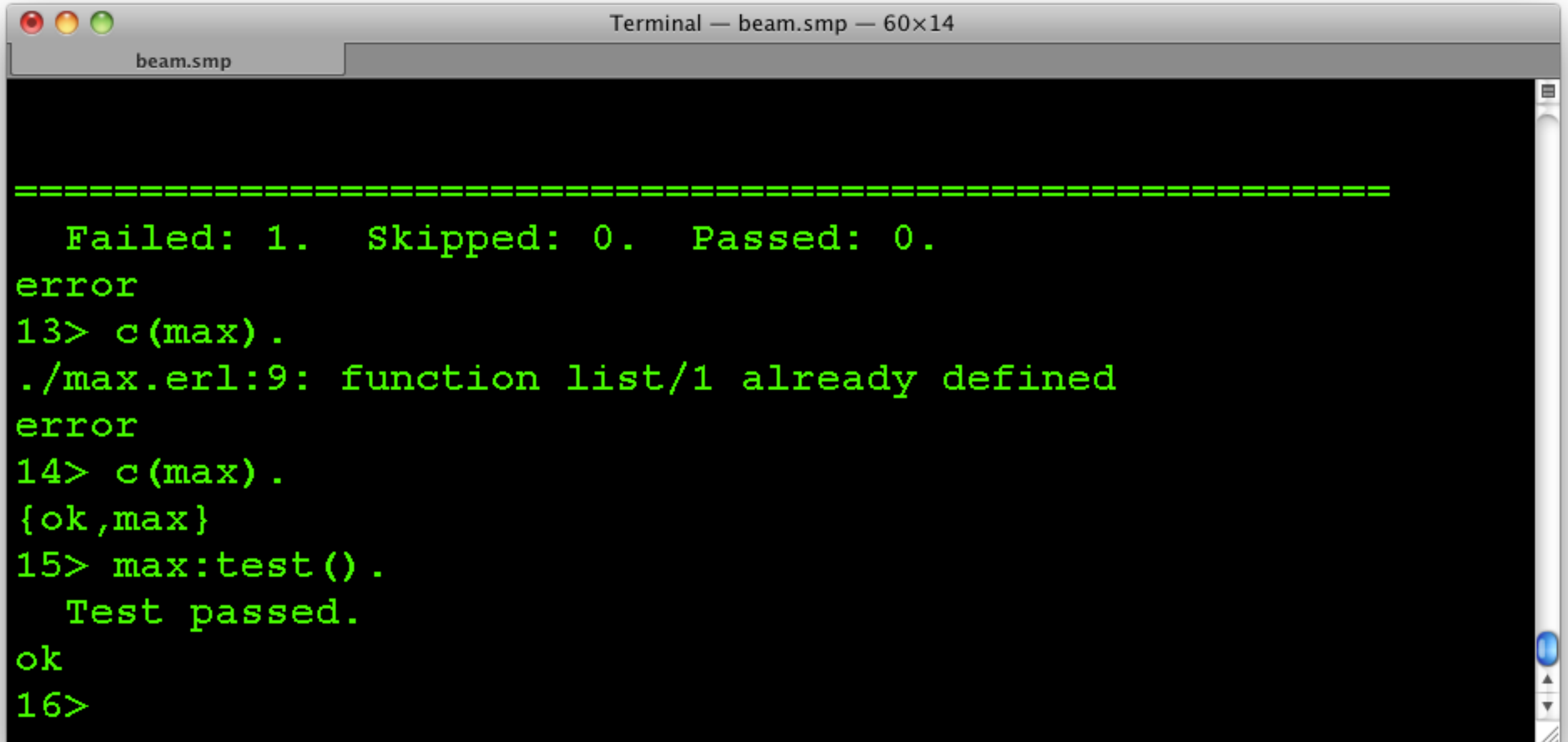
%% All functions with names ending wiht _test() or _test_() will be
%% called automatically by fifo:test()

empty_list_test() ->
    ?assertEqual({undefined, empty_list}, max:list([])).
```

empty_list};

Test Driven Development (TDD) using EUnit

Now we can successfully compile and run the EUnit test case.

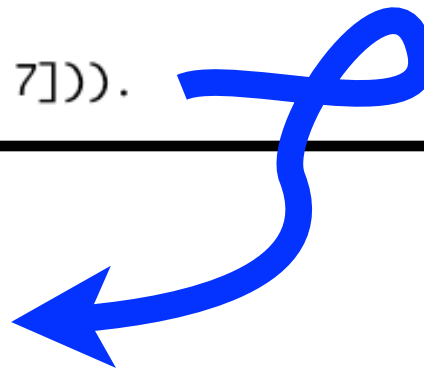
A screenshot of a macOS Terminal window titled "Terminal — beam.smp — 60x14". The window has a tab labeled "beam.smp". The terminal output is in green text on a black background. It shows a series of test results: a separator line of equals signs, a summary "Failed: 1. Skipped: 0. Passed: 0.", an "error" message, a shell prompt "13>" followed by "c(max) ." and an error from the Erlang shell: "./max.erl:9: function list/1 already defined". This is followed by another "error" message, a shell prompt "14>" followed by "c(max) ." and the output "{ok,max}", a shell prompt "15>" followed by "max:test() ." and the output "Test passed.", and finally "ok" and a shell prompt "16>".

```
=====  
Failed: 1.  Skipped: 0.  Passed: 0.  
error  
13> c(max) .  
./max.erl:9: function list/1 already defined  
error  
14> c(max) .  
{ok,max}  
15> max:test() .  
Test passed.  
ok  
16>
```

One more test

```
-module(max).  
-export([list/1]).  
  
%% To use EUnit we must include this:  
-include_lib("eunit/include/eunit.hrl").  
  
list([]) ->  
    {undefined, empty_list};  
list([H|_T]) ->  
    H.  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%%                               EUnit Test Cases                               %%  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
%% All functions with names ending wiht _test() or _test_() will be  
%% called automatically by fifo:test()  
  
empty_list_test() ->  
    ?assertEqual({undefined, empty_list}, max:list([])).  
  
list_test() ->  
    ?assertEqual(42, max:list([3, 7,-9, 42, 11, 7])).
```

Add a test case for a non empty list.



Compile and test again ...

```
Terminal — beam.smp — 60x18
beam.smp
19> c(max) .
{ok,max}
20> max:test().
max: list_test...*failed*
::error:{assertEqual_failed,
        [{module,max},
         {line,23},
         {expression,"max : list ( [ 3 , 7 , - 9 , 42 , 11
, 7 ] )"},
         {expected,42},
         {value,3}]]}
in function max:'-list_test/0-fun-0-' /1

=====
Failed: 1.  Skipped:
error
21> █
```

```
list([]) ->
    {undefined, empty_list};
list([H|_T]) ->
    H.
```

The new test obviously fails.

Write tests first - then implement the desired functionality

Now we implement a working `max:list/1` function. A common pattern is to use an auxiliary recursive function with an accumulator, here we use the `list/2` function.

```
list([]) ->
    {undefined, empty_list};
list([H|T]) ->
    list(T, H).

list([H|T], Max) when H > Max ->
    list(T, H);
list([_H|T], Max) ->
    list(T, Max).
```

A function head with a **guard**.

Guards are constructs that we can use to **increase the power of pattern matching**.

Besides boolean evaluation and math operations such (for example $A*B/C \geq 0$), only a small number of BIFs are allowed in guards:

`is_alive/0`
`is_boolean/1`
`is_builtin/3`
`is_constant/1`
`is_float/1`
`is_function/2`
`is_function/1`
`is_integer/1`
`is_list/1`

`is_number/1`
`is_pid/1`
`is_port/1`
`is_record/3`
`is_record/2`
`is_reference/1`
`is_tuple/1`
`tuple_size/1`
`is_binary/1`

`is_bitstring/1`
`bit_size/1`
`byte_size/1`
`length/1`



```
Terminal — beam.smp — 60x17
beam.smp
21> c(max).
./max.erl:14: Warning: variable 'H' is unused
{ok,max}
22> c(max).
{ok,max}
23> max:test().
max: list_test...*failed*
::error:function_clause
  in function max:list/2
    called as list([],42)
  in call from max:'-list_test/0-fun-0-' /1

=====
Failed: 1. Skipped: 0. Passed: 1.
error
24>
```

Oops - we did it again!

The **function_clause** error means there's is no function clause matching **`list([], 42)`**



Final version

We add a function clause for the case with an empty list.

```
list([]) ->
    {undefined, empty_list};
list([_H|_T]) ->
    list(T, H).
list([], Max) ->
    Max; %% Recursive base case
list([_H|_T], Max) when H > Max ->
    list(T, H);
list([_H|_T], Max) ->
    list(T, Max).
```



Add recursive base case!

Patterns are scanned from
top to bottom.


```
Terminal — beam.smp — 24x6
beam.smp
24> c(max) .
{ok,max}
25> max:test() .
    All 2 tests passed.
ok
26>
```

Divide and conqueror

What if we could split the list and find the max in each the sublists using separate processes.

Let's start with the splitting:

```
%%  
%% Split the list L into lists of length N  
%%  
  
%% Can we stop splitting?  
split(L, N) when length(L) < N ->  
    L;  
%% Do the splitting  
split(L, N) ->  
    split(L, N, []).  
  
%% An auxiliary recursive split function  
split(L, N, Lists) ->  
    {L1, L2} = lists:split(N, L),  
    if length(L2) > N ->  
        split(L2, N, [L1|Lists]);  
    true ->  
        [L1, L2|Lists]  
end.
```



Use the `lists:split/2` function which splits a lists into two lists `L1` and `L2` such that `length(L1) = N`.

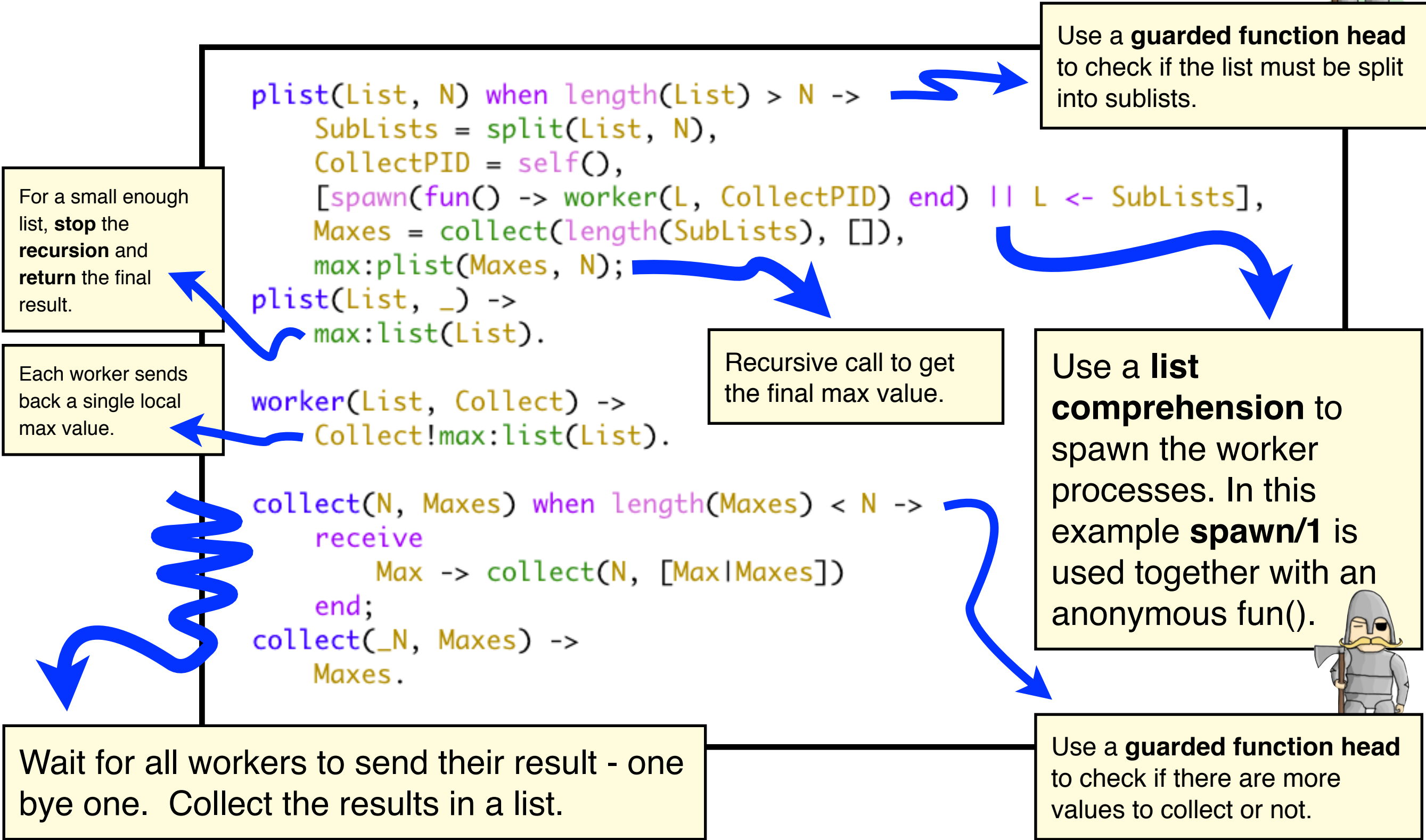
```
if  
    GuardSeq1 ->  
        Body1;  
    ...;  
    GuardSeqN ->  
        BodyN  
end
```

The branches of an **if-expression** are scanned sequentially until a guard sequence `GuardSeq` which evaluates to true is found. Then the corresponding `Body` (sequence of expressions separated by ',') is evaluated.

The return value of `Body` is the **return value** of the if expression.

Spawn workers - collect results

Use the split function to divide the original list into sublists. Spawn worker processes for each sublist and collect the results.



Add some more unit tests

Let's use random lists to test `max:list/1` and `max:plist/2`

Once again, **list comprehensions** makes the code dense but still clear and easy to understand.

```
random_list(N) ->
  [random:uniform() || _ <- lists:seq(1, N)].

random_lists_test() ->
  %% A list [1, 10, 100, ....]
  Lengths = [trunc(math:pow(10, N)) || N <- lists:seq(0, 5)],

  %% A list of random lists of increasing lengths
  RandomLists = [random_list(Length) || Length <- Lengths],

  [?_assertEqual(lists:max(L), max:list(L)) || L <- RandomLists].

random_plist_test() ->
  N = 10000,
  L = random_list(N),
  ?assertEqual(lists:max(L), max:plist(L, trunc(N/4))).
```

`_test_()` functions are used when running multiple tests.

`pow/2` returns a float, `trunc/1` returns an integer.

A `_test_()` function should return a list of tests. Use the `?_assert` EUnit macros to define the tests in the list.

EUnit will continue to run all tests within a `_test_()` function regardless if some of the tests fails.

```
Terminal — beam.smp — 24x6
beam.smp
44> c(max) .
{ok,max}
45> max:test() .
All 9 tests passed.
ok
46>
```

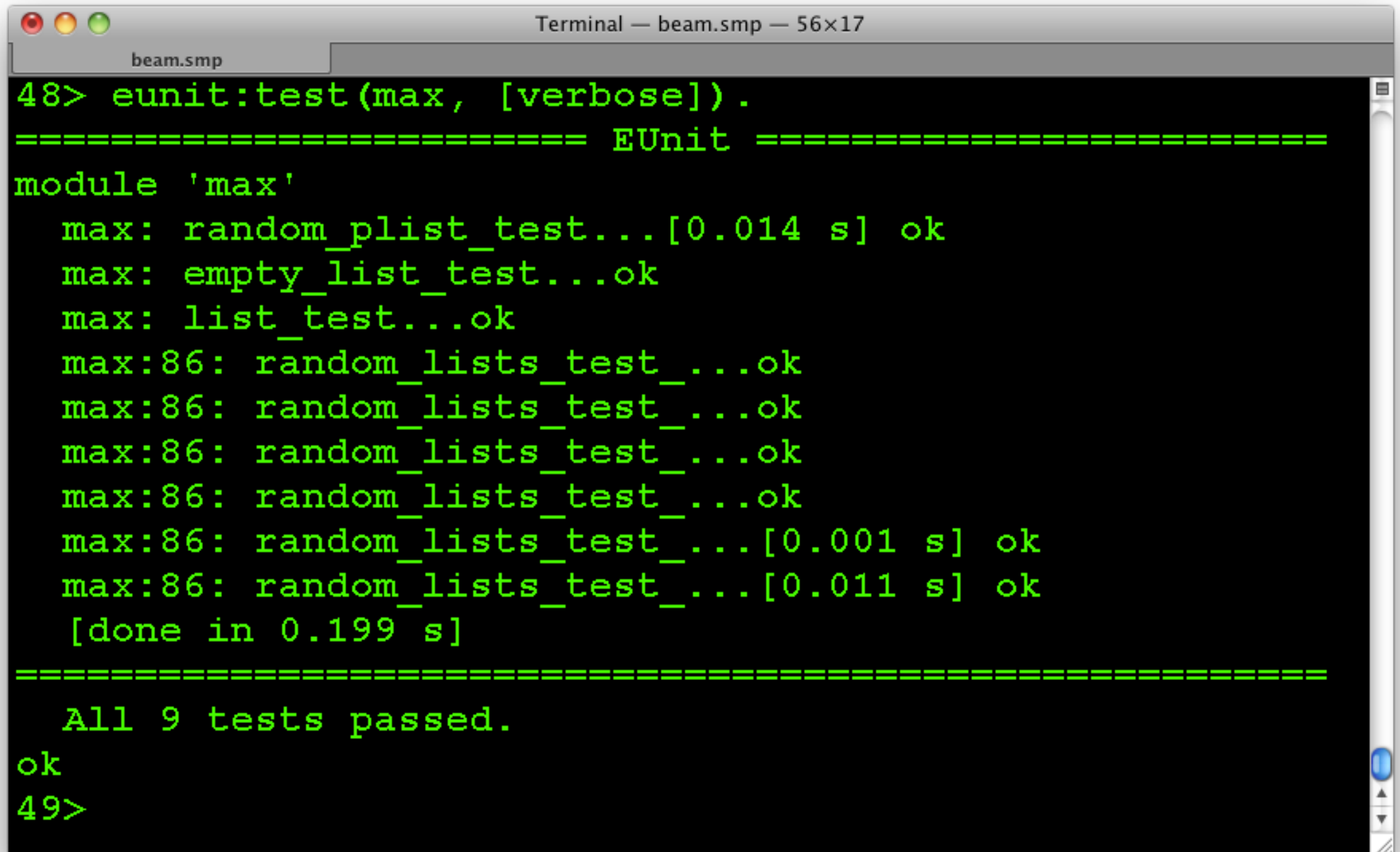


Adding unit tests early is a very good investment.

Future modifications can be made without worrying about messing up .

Verbose mode

Using `eunit:test/2` we can make EUnit display more details about the running tests.

A terminal window titled "Terminal — beam.smp — 56x17" with a tab labeled "beam.smp". The terminal shows the execution of the command `48> eunit:test(max, [verbose]).`. The output is a detailed report of test results for the 'max' module, including individual test names, durations, and pass/fail status. The tests are: `max: random_plist_test...` (0.014 s, ok), `max: empty_list_test...` (ok), `max: list_test...` (ok), `max:86: random_lists_test...` (ok), `max:86: random_lists_test...` (ok), `max:86: random_lists_test...` (ok), `max:86: random_lists_test...` (ok), `max:86: random_lists_test...` (0.001 s, ok), and `max:86: random_lists_test...` (0.011 s, ok). The total time is 0.199 s. The output concludes with "All 9 tests passed." and "ok". The prompt `49>` is visible at the bottom.

```
Terminal — beam.smp — 56x17
beam.smp
48> eunit:test(max, [verbose]).
===== EUnit =====
module 'max'
  max: random_plist_test...[0.014 s] ok
  max: empty_list_test...ok
  max: list_test...ok
  max:86: random_lists_test...ok
  max:86: random_lists_test...ok
  max:86: random_lists_test...ok
  max:86: random_lists_test...ok
  max:86: random_lists_test...[0.001 s] ok
  max:86: random_lists_test...[0.011 s] ok
  [done in 0.199 s]
=====
  All 9 tests passed.
ok
49>
```

Expressive density

```
-module(max).
-export([list/1, plist/2]).

%% To use EUnit we must include this:
-include_lib("eunit/include/eunit.hrl").

list([]) ->
    {undefined, empty_list};
list([_H|_T]) ->
    list(T, H).
list([], Max) ->
    Max; %% Recursive base case
list([H|_T], Max) when H > Max ->
    list(T, H);
list([_H|_T], Max) ->
    list(T, Max).

%%
%% Split the list L into lists of length N
%%

%% Can we stop splitting?
split(L, N) when length(L) < N ->
    L;
%% Do the splitting
split(L, N) ->
    split(L, N, []).

%% An auxiliary recursive split function
split(L, N, Lists) ->
    {L1, L2} = lists:split(N, L),
    if length(L2) > N ->
        split(L2, N, [L1|Lists]);
    true ->
        [L1, L2|Lists]
    end.
```

```
plist(List, N) when length(List) > N ->
    SubLists = split(List, N),
    CollectPID = self(),
    [spawn(fun() -> worker(L, CollectPID) end) || L <- SubLists],
    Maxes = collect(length(SubLists), []),
    max:plist(Maxes, N);
plist(List, _) ->
    max:list(List).

worker(List, Collect) ->
    Collect!max:list(List).

collect(N, Maxes) when length(Maxes) < N ->
    receive
        Max -> collect(N, [Max|Maxes])
    end;
collect(_N, Maxes) ->
    Maxes.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                               EUnit Test Cases
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% All functions with names ending with _test() or _test_() will be
%% called automatically by max:test()

empty_list_test() ->
    ?assertEqual({undefined, empty_list}, max:list([])).

list_test() ->
    ?assertEqual(42, max:list([3, 7, -9, 42, 11, 7])).

random_list(N) ->
    [random:uniform() || _ <- lists:seq(1, N)].

random_lists_test_() ->
    %% A list [1, 10, 100, ....]
    Lengths = [trunc(math:pow(10, N)) || N <- lists:seq(0, 5)],

    %% A list of random lists of increasing lengths
    RandomLists = [random_list(Length) || Length <- Lengths],

    [?_assertEqual(lists:max(L), max:list(L)) || L <- RandomLists].

random_plist_test() ->
    N = 10000,
    L = random_list(N),
    ?assertEqual(lists:max(L), max:plist(L, trunc(N/4))).
```

The Erlang language is very dense.

Imagine the number of lines of code needed to implement this in any other language you know.

Adding unit tests doesn't require many lines of code.

Tip 1: list comprehensions

Tip 2: invest early in unit tests.

Tip 3: there are many useful functions in the lists library.