Test driven development (TDD) in Erlang with EUnit

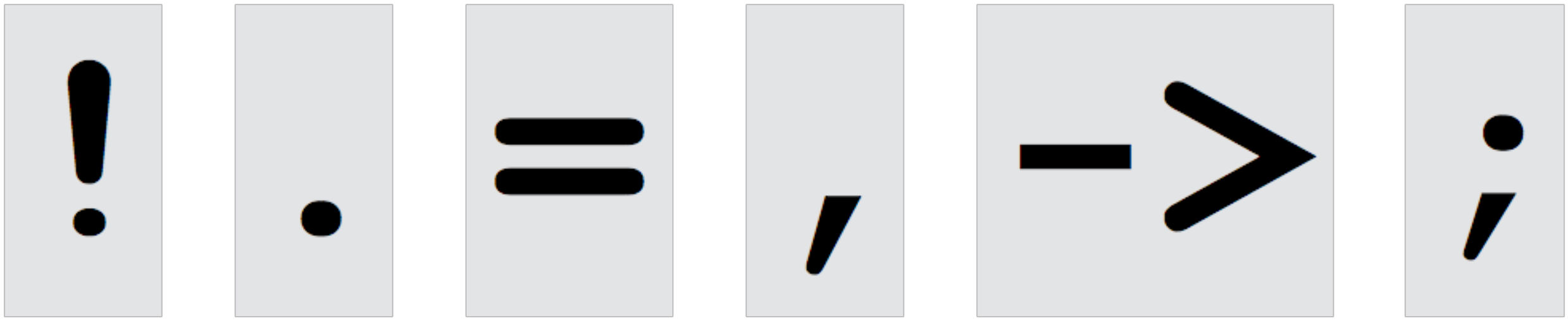
Module 8 - Erlang tutorial 5

Operating systems and process oriented programming 2020

1DT096

Spring 2020 karl.marklund@it.uu.se Uppsala university

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. Eripascal 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

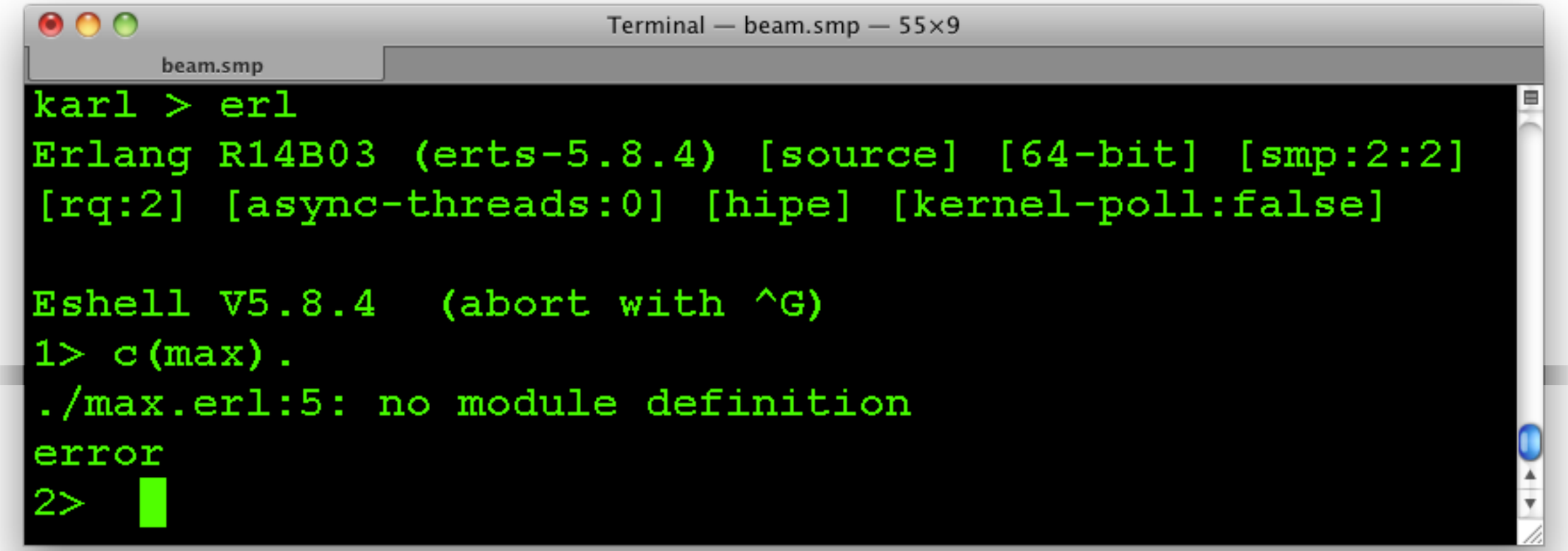
Test Driven Development

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.



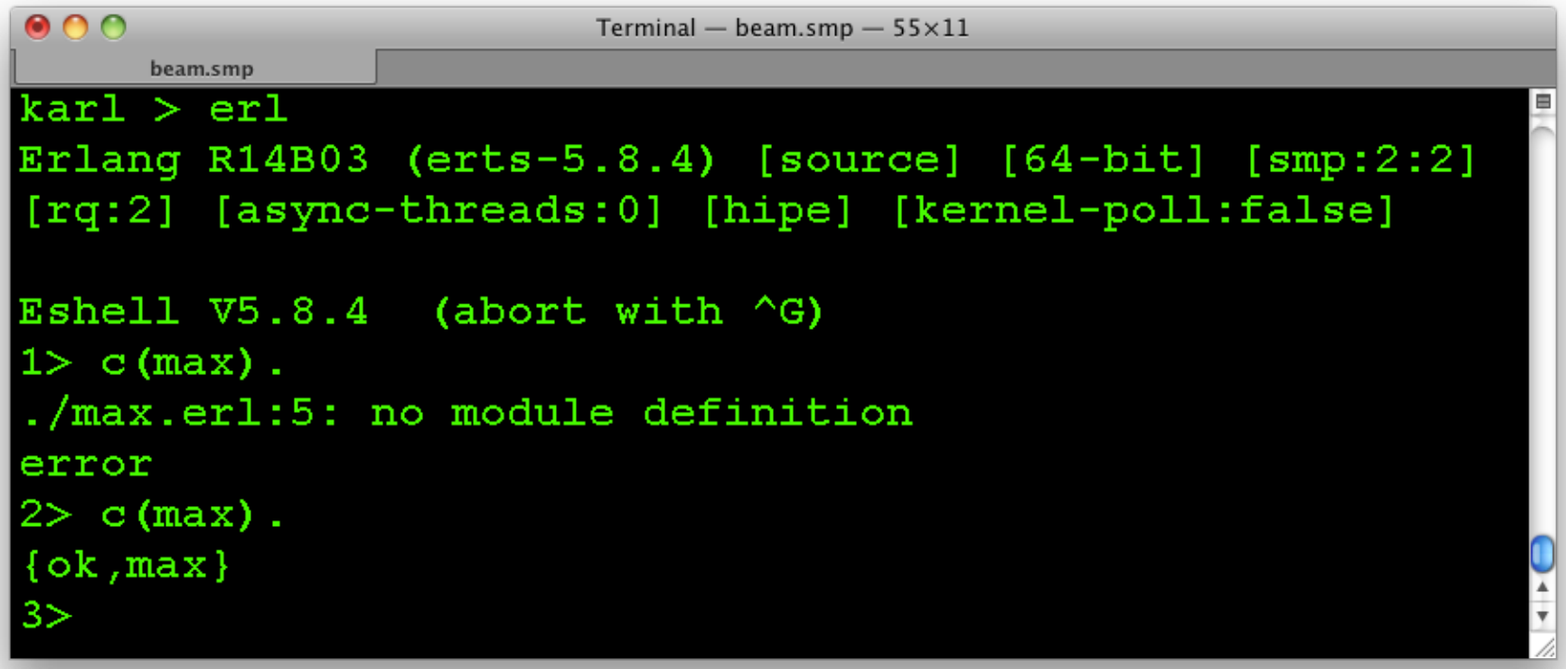
We got a compilation error :-(

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 :-)

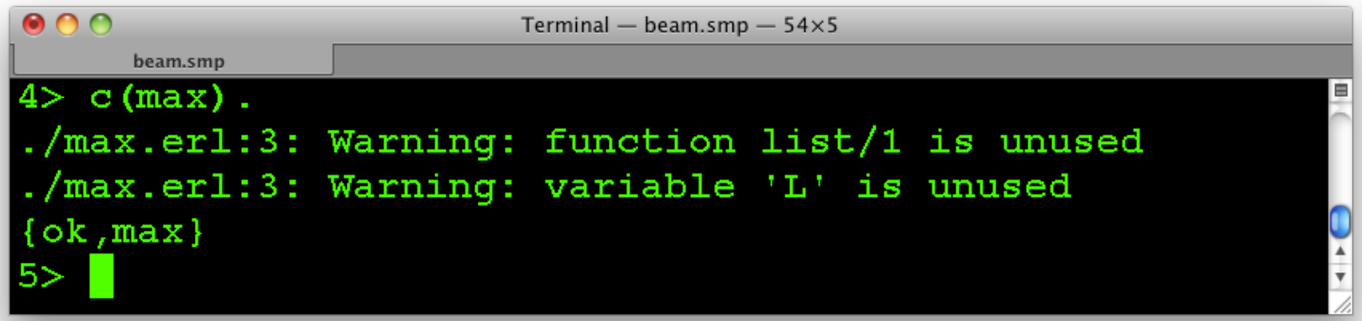


Compiler warnings

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!



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.

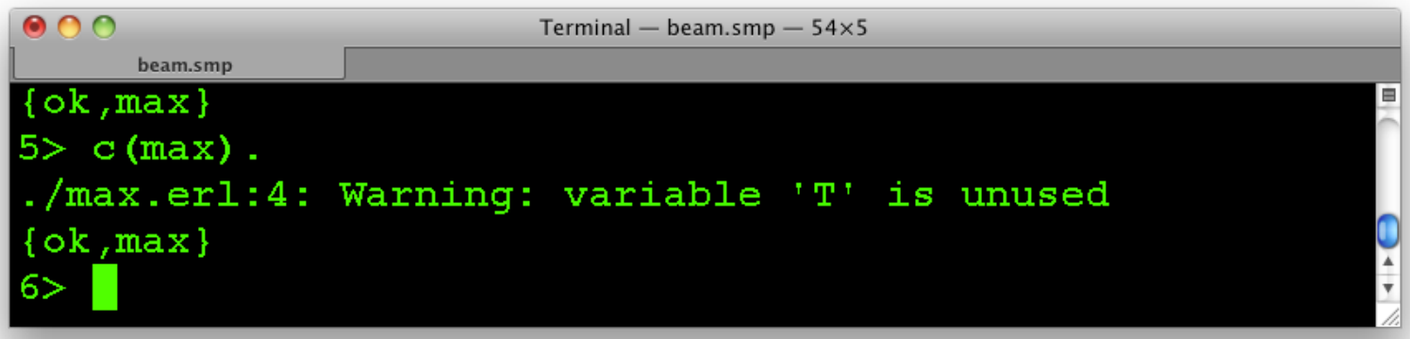
To get rid of the warning about the variable L being unused, we simply returns the first element of the list.

```
-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!



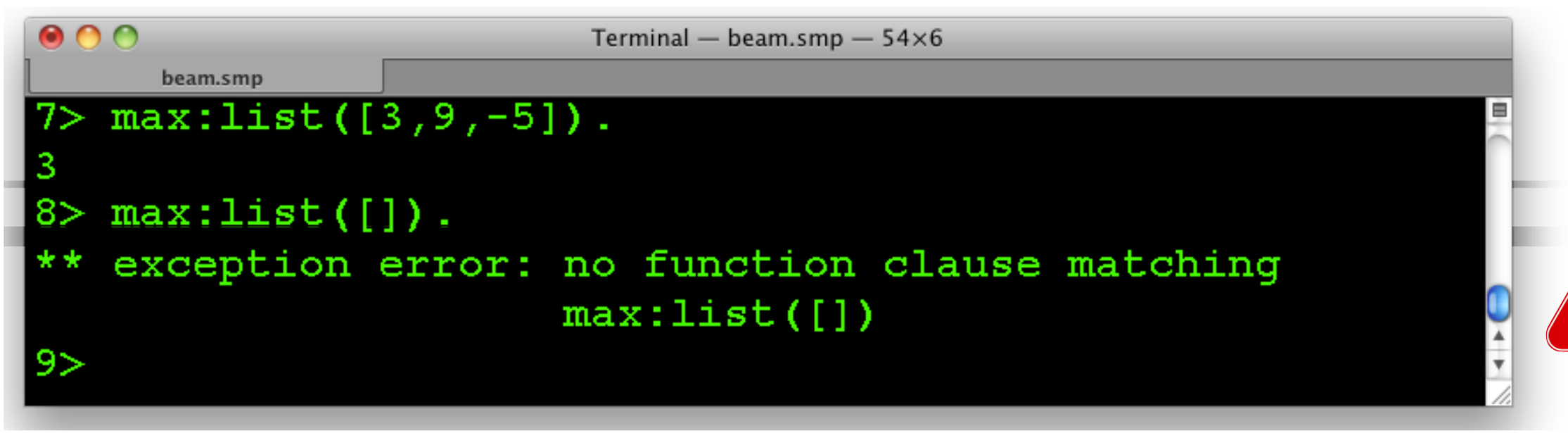
Oops - the variable T is unused...

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.



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([])).
```

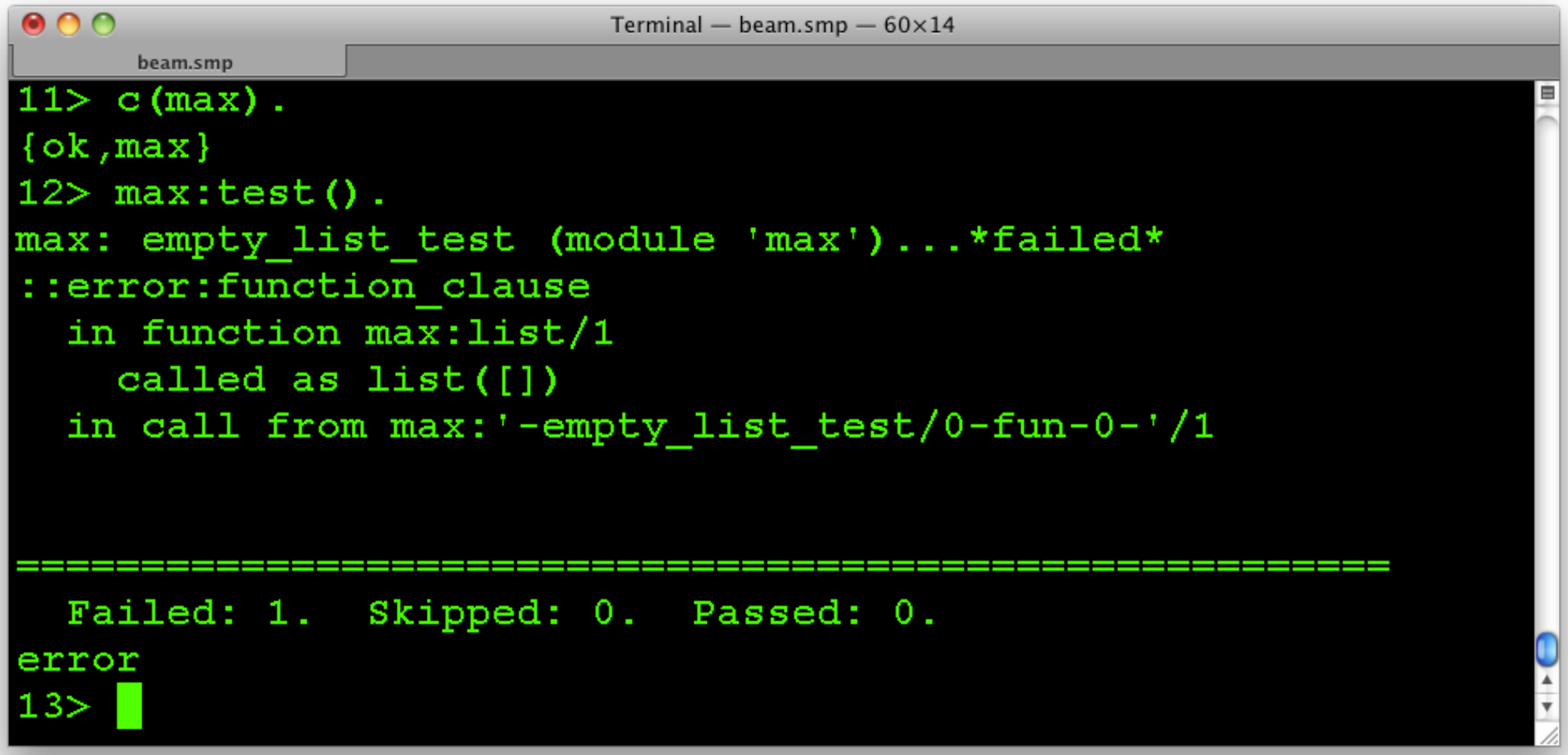
EUnit provides a number of **`?assert`** macros.

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.



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 — 60×14

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

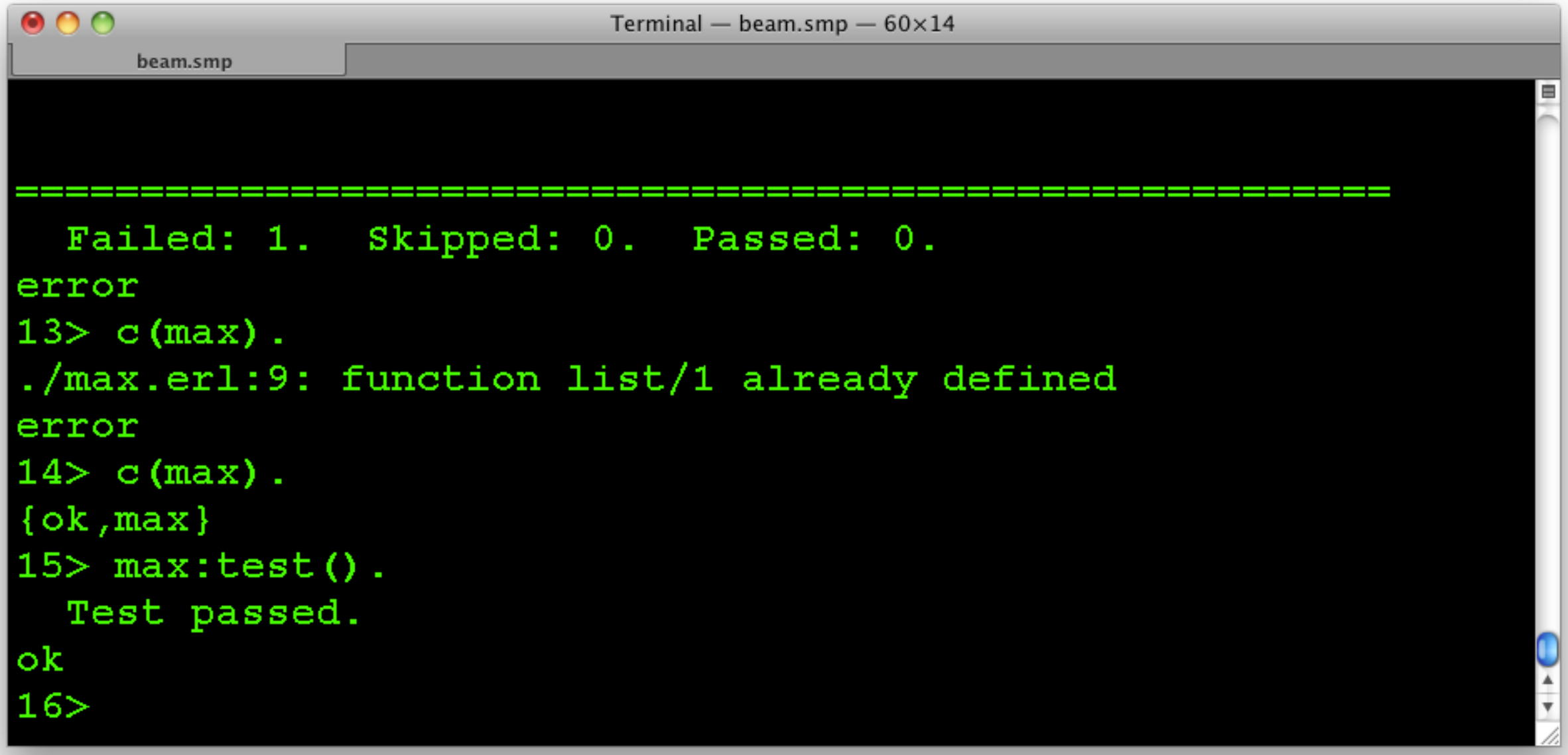
We separate the two function clauses with a semi-colon.

```
-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.



One more test

```erlang
-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 — 60×18

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 >= 0`), only a small number of BIFs are allowed in guards:

`is_alive/0`	**`is_number/1`**	**`is_bitstring/1`**
`is_boolean/1`	**`is_pid/1`**	**`bit_size/1`**
`is_builtin/3`	**`is_port/1`**	**`byte_size/1`**
`is_constant/1`	**`is_record/3`**	**`length/1`**
`is_float/1`	**`is_record/2`**	
`is_function/2`	**`is_reference/1`**	
`is_function/1`	**`is_tuple/1`**	
`is_integer/1`	**`tuple_size/1`**	
`is_list/1`	**`is_binary/1`**	

```
Terminal — beam.smp — 60×17
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)`**

Image: http://learnyousomeerlang.com

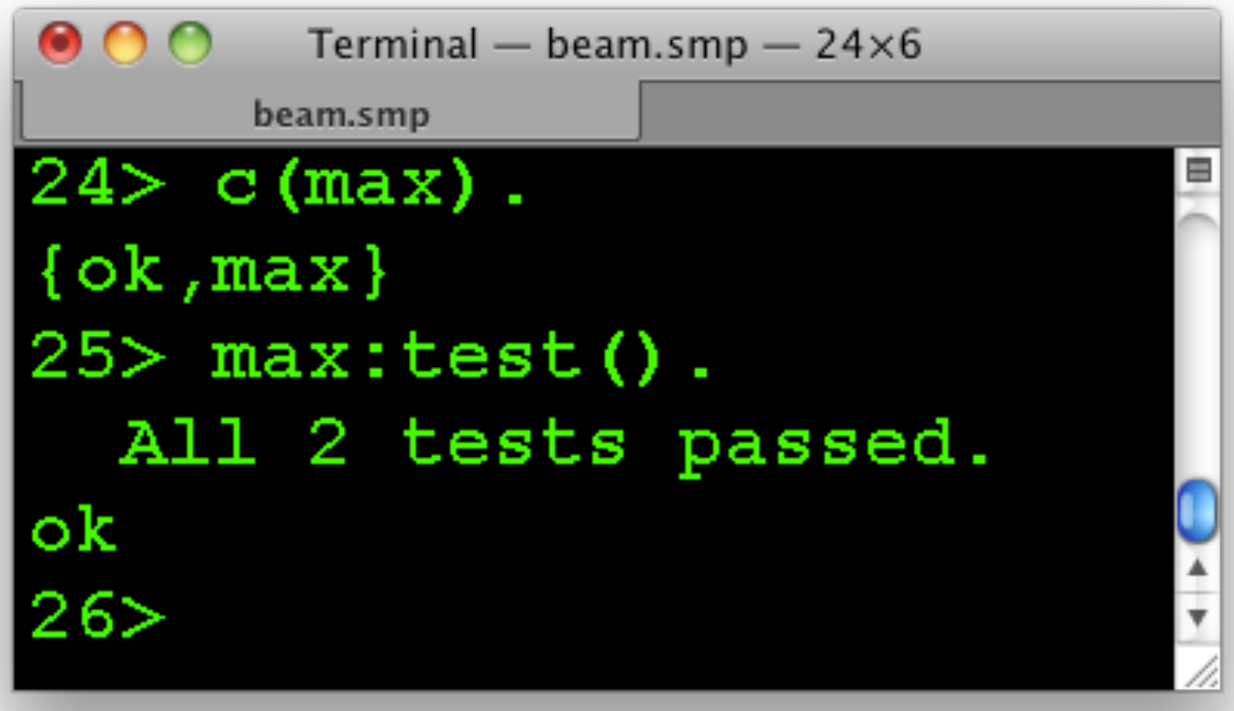
Final version

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.



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 correspondingBody (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.

```
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.
```

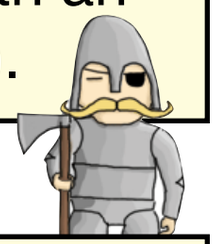
Use a **guarded function head** to check if the list must be split into sublists.

For a small enough list, **stop** the **recursion** and **return** the final result.

Each worker sends back a single local max value.

Recursive call to get the final max value.

Use a **list comprehension** to spawn the worker processes. In this example **spawn/1** is used together with an anonymous fun().



Wait for all workers to send their result - one bye one. Collect the results in a list.

Use a **guarded function head** to check if there are more values to collect or not.

Add some more unit tests

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

```erlang
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))).
```

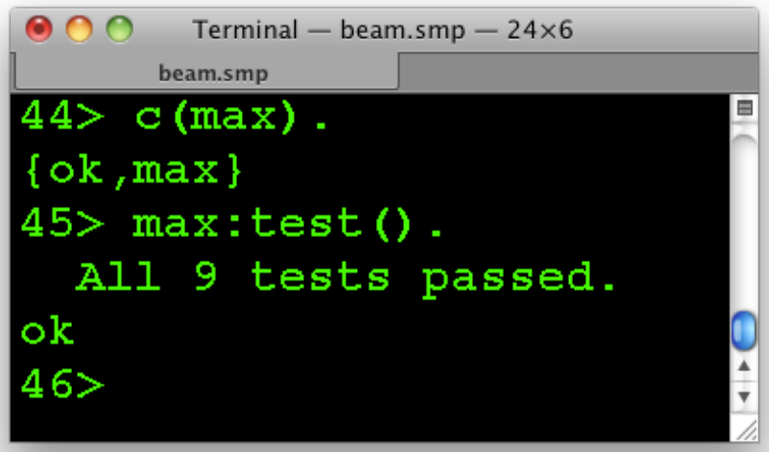
Once again, **list comprehensions** makes the code dense but still clear and easy to understand.

`_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.

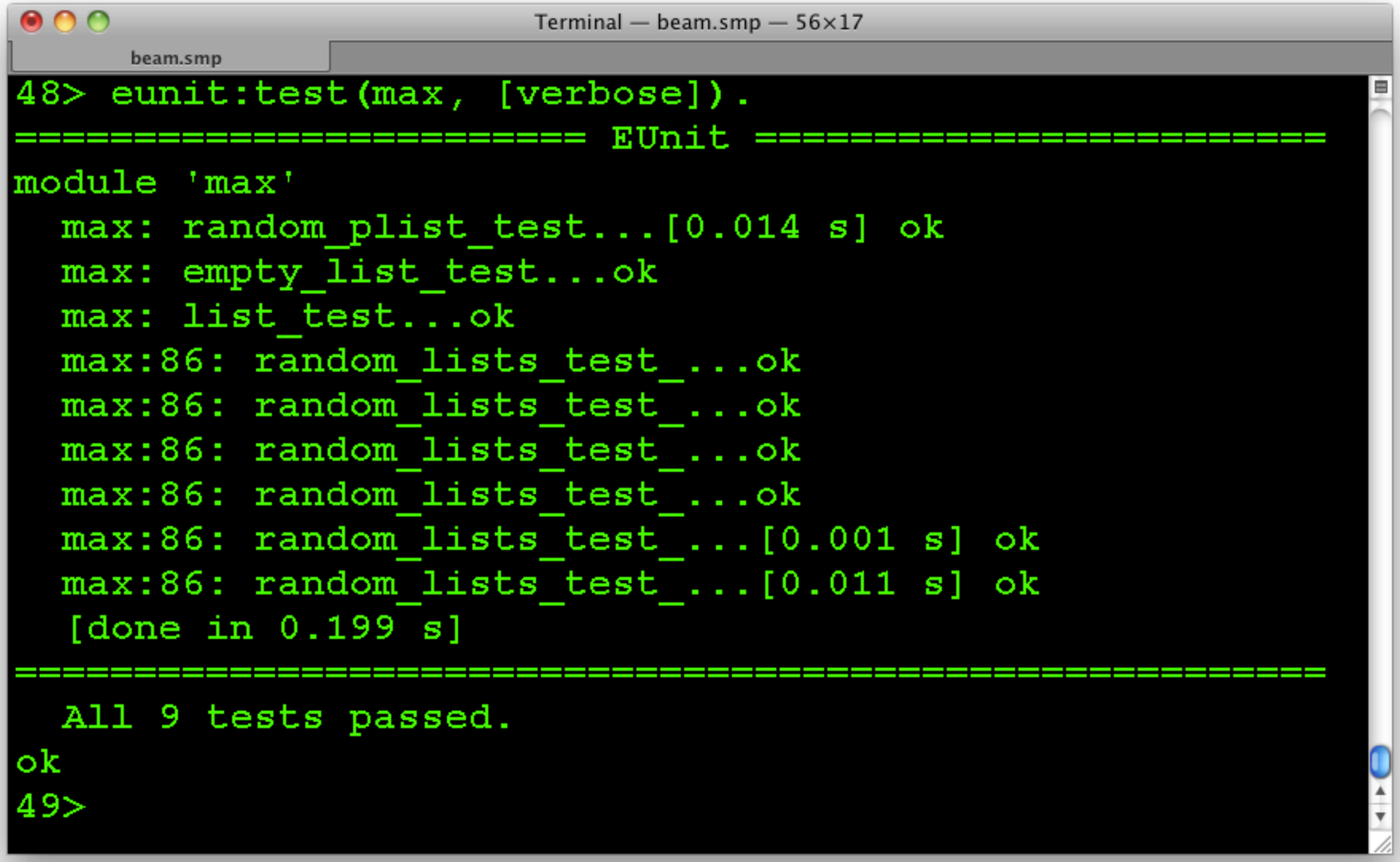


Adding unit tests early is a very good investment.

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.



Expressive density

```erlang
-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.
```

```erlang
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.
```

```erlang
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                          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([])).

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.**