```
c(module.erl).
```

Now that module's functions can be accessed with:

```
module:func
```

```
-module(filename).
```

... for a program defined in filename.erl.

```
-import(filename, [func1/arity1, ..., funcN/arityN]).
```

```
-export(filename, [func1/arity1, ..., funcN/arityN]).
```

or

```
-compile(export_all).
```

Similar to Prolog, and Haskell, functions and atoms begin with a lower case, variables begin with a capital letter or an underscore.

An atom is a word that stands for itself.
It begins with a lowercase letter or is enclosed in single quotes.

Integers, floats, strings, atoms, lists, tuples, binaries.

Lists are enclosed in square brackets, tuples in curly brackets, and binaries in double angle brackets.

They don't short-circuit by default. For short-circuiting you have `andalso` and `orelse`.

>= and =<.

not <=.

`==` equal to
`/=` not equal to
`=:=` exactly equal to
`=/=` not exactly equal to

The non-exact versions will coerce its into floats. Otherwise use exact versions to give better hints to the compiler.

Pattern matching, not assignment. "Assignment" is just a simple case of pattern matching.

```
case Expression of
    Pattern1 [when Guard1] -> Expression_sequence1;
    ...
    PatternN [when GuardN] -> Expression_sequenceN
end
```

A sequence of expressions separated by commas.

The value of the last expression evaluated.

```
if
    Guard1 -> Expression_sequence1;
    ...
    Guard2 -> Expression_sequence2
end
```

An error will be returned.

One can use the `true` atom as the final guard, although it is usually better to use something more explicit than `true` when possible.

To enforce no-side-effects, they cannot be user-defined.

You can use type tests, many operators, and a number of built-in functions.

```
length(List)

size(Tuple)
```

```
name(Patterns1) -> Expression_sequence1;
...
name(PatternsN) -> Expression_sequenceN.
```

```
fun(Patterns1) -> Body1;
    (Patterns2) -> Body2;
    ...
    (PatternsN) -> BodyN
end
```

They are often used as parameters to other functions.

```
[Expression || Generator, GuardOrGenerator, ..., GuardOrGenerator]
```

The expression typically makes use of variables defined by a generator.
Guards are simply boolean expressions.
Generators are of the form `El <- List`.

```
[X * 2 || X <- [1, 2, 3, 4]]
```

`hd` (head), `tl` (tail), `length`

```
lists:seq(From, To)
lists:seq(From, To, Step)
```

Note: `To` is inclusive, unlike in Scala/Python/etc

```
Line = io:get_line(Prompt).

io:format(FormatString, ListOfData).
```

```
{ok, Stream} = file:open(FileName, write),
io:format(Stream, FormatString, ListOfData),
file:close(Stream).
```

`~s` a string
`~w` a value in its standard syntax (e.g., strings as lists of integers)
`~p` a value, pretty printed (e.g., strings with quotes around them)
`~n` or `\n` newline

% causes the rest of the line to be discarded.

First the function to apply to the members of the list, and then the list.

```
0> 2 rem 3.
2
```

Create an anonymous function in-line using the function's full name, *funcName/arity*.
E.g.,

```
get_red() -> filter(fun is_red/1, fruit()).
```

Lists of ASCII integer values.

The same as Scala's `forall` and `exists`.

The two lists must be of the same length.

```
lists:append([1, 2], [3, 4])
```

or

```
[1, 2] ++ [3, 4]
```

result in

```
[1, 2, 3, 4]
```

`takewhile` scans through a list until the predicate fails for the first time, at which point it discards the failed element and everything after.

`dropwhile` scans through the list, discarding everything until the predicate holds for the first time.

It creates a tuple of two lists. The first list holds the elements of the input list for which the predicate held, and the second holds the elements for which the predicate failed.

```
Pid = spawn(Function)
```

`self()` results in the Pid of the executing process.

```
receive
    Pattern1 [when Guard1] -> Expression_sequence1;
    ...
    PatternN [when GuardN] -> Expression_sequenceN
after Timeout ->
    TimeoutExpressionSequence
end
```

`loop` in Scala is a control structure in the actors library that does the looping for you.
In Erlang `loop` is just the name of a function you're calling so you must make the recursive call at the end of each `receive` case.

Provide a function that takes as an argument the Pid of the instance of the module to send the message to.
Send the message there, wait for a response, and then reply.

Ultimately a user may start several processes, but after that point will interact with the blocking rpc function.

Registering a Pid makes it globally available.

`register(AnAtom, Pid)` gives `Pid` a name

`unregister(AnAtom)`

`registered() -> [AnAtom :: atom()]`

returns a list of all registered processes

`whereis(AnAtom) -> Pid | undefined`

gets Pid of a registered process, or undefined if no such process

`spawn` creates a processes independent from the current one.

`spawn_link` creates a new process linked to the current one, such that if the new process exits non-normally, so will the current one.

`link(Pid)` and `unlink(Pid)`

`exit(Reason)` exits and sends signals to linked processes.

`exit(Pid, Reason` sends the exit signal to the given process, but doesn't terminate the current process.

```
process_flag(trap_exit, true)
spawn_link(Function)
```

Now exit signals are converted to regular `{'EXIT', From, Reason}` messages.
However, if the reason is `kill`, exit will still be forced.

Crashed processes are normally handled through linking, not in `catch` sequences.

Create a helper function with an additional argument that will serve as an accumulator.