Erlang

```erlang
% Percent sign starts a one-line comment.

%% Two percent characters shall be used to comment functions.

%%% Three percent characters shall be used to comment modules.

% We use three types of punctuation in Erlang.
% Commas (`,`) separate arguments in function calls, data constructors, and
% patterns.
% Periods (`.`) (followed by whitespace) separate entire functions and
% expressions in the shell.
% Semicolons (`;`) separate clauses. We find clauses in several contexts:
% function definitions and in `case`, `if`, `try..catch`, and `receive`
% expressions.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 1. Variables and pattern matching.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% In Erlang new variables are bound with an `=` statement.
Num = 42.  % All variable names must start with an uppercase letter.

% Erlang has single-assignment variables; if you try to assign a different
% value to the variable `Num`, you'll get an error.
Num = 43. % ** exception error: no match of right hand side value 43

% In most languages, `=` denotes an assignment statement. In Erlang, however,
% `=` denotes a pattern-matching operation. When an empty variable is used on the
% left hand side of the `=` operator to is bound (assigned), but when a bound
% variable is used on the left hand side the following behaviour is observed.
% `Lhs = Rhs` really means this: evaluate the right side (`Rhs`), and then
% match the result against the pattern on the left side (`Lhs`).
Num = 7 * 6.

% Floating-point number.
Pi = 3.14159.

% Atoms are used to represent different non-numerical constant values. Atoms
% start with lowercase letters, followed by a sequence of alphanumeric
% characters or the underscore (`_`) or at (`@`) sign.
Hello = hello.
OtherNode = example@node.

% Atoms with non alphanumeric values can be written by enclosing the atoms
% with apostrophes.
AtomWithSpace = 'some atom with space'.

% Tuples are similar to structs in C.
Point = {point, 10, 45}.

% If we want to extract some values from a tuple, we use the pattern-matching
% operator `=`.
```

```
{point, X, Y} = Point.  % X = 10, Y = 45

% We can use `_` as a placeholder for variables that we're not interested in.
% The symbol `_` is called an anonymous variable. Unlike regular variables,
% several occurrences of `_` in the same pattern don't have to bind to the
% same value.
Person = {person, {name, {first, joe}, {last, armstrong}}, {footsize, 42}}.
{_, {_, {_, Who}, _}, _} = Person.  % Who = joe

% We create a list by enclosing the list elements in square brackets and
% separating them with commas.
% The individual elements of a list can be of any type.
% The first element of a list is the head of the list. If you imagine removing
% the head from the list, what's left is called the tail of the list.
ThingsToBuy = [{apples, 10}, {pears, 6}, {milk, 3}].

% If `T` is a list, then `[H|T]` is also a list, with head `H` and tail `T`.
% The vertical bar (`|`) separates the head of a list from its tail.
% `[]` is the empty list.
% We can extract elements from a list with a pattern-matching operation. If we
% have a nonempty list `L`, then the expression `[X|Y] = L`, where `X` and `Y`
% are unbound variables, will extract the head of the list into `X` and the tail
% of the list into `Y`.
[FirstThing|OtherThingsToBuy] = ThingsToBuy.
% FirstThing = {apples, 10}
% OtherThingsToBuy = [{pears, 6}, {milk, 3}]

% There are no strings in Erlang. Strings are really just lists of integers.
% Strings are enclosed in double quotation marks (`"`).
Name = "Hello".
[72, 101, 108, 108, 111] = "Hello".


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 2. Sequential programming.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Modules are the basic unit of code in Erlang. All the functions we write are
% stored in modules. Modules are stored in files with `.erl` extensions.
% Modules must be compiled before the code can be run. A compiled module has the
% extension `.beam`.
-module(geometry).
-export([area/1]). % the list of functions exported from the module.

% The function `area` consists of two clauses. The clauses are separated by a
% semicolon, and the final clause is terminated by dot-whitespace.
% Each clause has a head and a body; the head consists of a function name
% followed by a pattern (in parentheses), and the body consists of a sequence of
% expressions, which are evaluated if the pattern in the head is successfully
% matched against the calling arguments. The patterns are matched in the order
% they appear in the function definition.
area({rectangle, Width, Ht}) -> Width * Ht;
area({circle, R})            -> 3.14159 * R * R.
```

```erlang
% Compile the code in the file geometry.erl.
c(geometry).  % {ok,geometry}

% We need to include the module name together with the function name in order to
% identify exactly which function we want to call.
geometry:area({rectangle, 10, 5}).  % 50
geometry:area({circle, 1.4}).  % 6.15752

% In Erlang, two functions with the same name and different arity (number of
% arguments) in the same module represent entirely different functions.
-module(lib_misc).
-export([sum/1]). % export function `sum` of arity 1
                  % accepting one argument: list of integers.
sum(L) -> sum(L, 0).
sum([], N)    -> N;
sum([H|T], N) -> sum(T, H+N).

% Funs are "anonymous" functions. They are called this way because they have
% no name. However, they can be assigned to variables.
Double = fun(X) -> 2 * X end. % `Double` points to an anonymous function
                              % with handle: #Fun<erl_eval.6.17052888>
Double(2).  % 4

% Functions accept funs as their arguments and can return funs.
Mult = fun(Times) -> ( fun(X) -> X * Times end ) end.
Triple = Mult(3).
Triple(5).  % 15

% List comprehensions are expressions that create lists without having to use
% funs, maps, or filters.
% The notation `[F(X) || X <- L]` means "the list of `F(X)` where `X` is taken
% from the list `L`."
L = [1,2,3,4,5].
[2 * X || X <- L].  % [2,4,6,8,10]
% A list comprehension can have generators and filters, which select subset of
% the generated values.
EvenNumbers = [N || N <- [1, 2, 3, 4], N rem 2 == 0]. % [2, 4]

% Guards are constructs that we can use to increase the power of pattern
% matching. Using guards, we can perform simple tests and comparisons on the
% variables in a pattern.
% You can use guards in the heads of function definitions where they are
% introduced by the `when` keyword, or you can use them at any place in the
% language where an expression is allowed.
max(X, Y) when X > Y -> X;
max(X, Y) -> Y.

% A guard is a series of guard expressions, separated by commas (`,`).
% The guard `GuardExpr1, GuardExpr2, ..., GuardExprN` is true if all the guard
% expressions `GuardExpr1`, `GuardExpr2`, ..., `GuardExprN` evaluate to `true`.
is_cat(A) when is_atom(A), A =:= cat -> true;
is_cat(A) -> false.
is_dog(A) when is_atom(A), A =:= dog -> true;
is_dog(A) -> false.
```

```erlang
% We won't dwell on the `=:=` operator here; just be aware that it is used to
% check whether two Erlang expressions have the same value *and* the same type.
% Contrast this behaviour to that of the `==` operator:
1 + 2 =:= 3.   % true
1 + 2 =:= 3.0. % false
1 + 2 ==  3.0. % true

% A guard sequence is either a single guard or a series of guards, separated
% by semicolons (`;`). The guard sequence `G1; G2; ...; Gn` is true if at
% least one of the guards `G1`, `G2`, ..., `Gn` evaluates to `true`.
is_pet(A) when is_atom(A), (A =:= dog);(A =:= cat) -> true;
is_pet(A)                                          -> false.

% Warning: not all valid Erlang expressions can be used as guard expressions;
% in particular, our `is_cat` and `is_dog` functions cannot be used within the
% guard sequence in `is_pet`'s definition. For a description of the
% expressions allowed in guard sequences, refer to this
% [section](http://erlang.org/doc/reference_manual/expressions.html#id81912)
% of the Erlang reference manual.

% Records provide a method for associating a name with a particular element in a
% tuple.
% Record definitions can be included in Erlang source code files or put in files
% with the extension `.hrl`, which are then included by Erlang source code
% files.
-record(todo, {
  status = reminder,  % Default value
  who = joe,
  text
}).

% We have to read the record definitions into the shell before we can define a
% record. We use the shell function `rr` (short for read records) to do this.
rr("records.hrl").  % [todo]

% Creating and updating records:
X = #todo{}.
% #todo{status = reminder, who = joe, text = undefined}
X1 = #todo{status = urgent, text = "Fix errata in book"}.
% #todo{status = urgent, who = joe, text = "Fix errata in book"}
X2 = X1#todo{status = done}.
% #todo{status = done, who = joe, text = "Fix errata in book"}

% `case` expressions.
% `filter` returns a list of all elements `X` in a list `L` for which `P(X)` is
% true.
filter(P, [H|T]) ->
  case P(H) of
    true -> [H|filter(P, T)];
    false -> filter(P, T)
  end;
filter(P, []) -> [].
filter(fun(X) -> X rem 2 == 0 end, [1, 2, 3, 4]). % [2, 4]
```

```erlang
% `if` expressions.
max(X, Y) ->
  if
    X > Y -> X;
    X < Y -> Y;
    true -> nil
  end.

% Warning: at least one of the guards in the `if` expression must evaluate to
% `true`; otherwise, an exception will be raised.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 3. Exceptions.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Exceptions are raised by the system when internal errors are encountered or
% explicitly in code by calling `throw(Exception)`, `exit(Exception)`, or
% `erlang:error(Exception)`.
generate_exception(1) -> a;
generate_exception(2) -> throw(a);
generate_exception(3) -> exit(a);
generate_exception(4) -> {'EXIT', a};
generate_exception(5) -> erlang:error(a).

% Erlang has two methods of catching an exception. One is to enclose the call to
% the function that raises the exception within a `try...catch` expression.
catcher(N) ->
  try generate_exception(N) of
    Val -> {N, normal, Val}
  catch
    throw:X -> {N, caught, thrown, X};
    exit:X -> {N, caught, exited, X};
    error:X -> {N, caught, error, X}
  end.

% The other is to enclose the call in a `catch` expression. When you catch an
% exception, it is converted into a tuple that describes the error.
catcher(N) -> catch generate_exception(N).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 4. Concurrency
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Erlang relies on the actor model for concurrency. All we need to write
% concurrent programs in Erlang are three primitives: spawning processes,
% sending messages and receiving messages.

% To start a new process, we use the `spawn` function, which takes a function
% as argument.

F = fun() -> 2 + 2 end. % #Fun<erl_eval.20.67289768>
```

```erlang
spawn(F). % <0.44.0>

% `spawn` returns a pid (process identifier); you can use this pid to send
% messages to the process. To do message passing, we use the `!` operator.
% For all of this to be useful, we need to be able to receive messages. This is
% achieved with the `receive` mechanism:

-module(calculateGeometry).
-compile(export_all).
calculateArea() ->
    receive
      {rectangle, W, H} ->
        W * H;
      {circle, R} ->
        3.14 * R * R;
      _ ->
        io:format("We can only calculate area of rectangles or circles.")
    end.

% Compile the module and create a process that evaluates `calculateArea` in the
% shell.
c(calculateGeometry).
CalculateArea = spawn(calculateGeometry, calculateArea, []).
CalculateArea ! {circle, 2}. % 12.56000000000000049738

% The shell is also a process; you can use `self` to get the current pid.
self(). % <0.41.0>

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 5. Testing with EUnit
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Unit tests can be written using EUnits's test generators and assert macros
-module(fib).
-export([fib/1]).
-include_lib("eunit/include/eunit.hrl").

fib(0) -> 1;
fib(1) -> 1;
fib(N) when N > 1 -> fib(N-1) + fib(N-2).

fib_test_() ->
    [?_assert(fib(0) =:= 1),
     ?_assert(fib(1) =:= 1),
     ?_assert(fib(2) =:= 2),
     ?_assert(fib(3) =:= 3),
     ?_assert(fib(4) =:= 5),
     ?_assert(fib(5) =:= 8),
     ?_assertException(error, function_clause, fib(-1)),
     ?_assert(fib(31) =:= 2178309)
    ].

% EUnit will automatically export to a test() function to allow running the tests
% in the erlang shell
```

```
fib:test()

% The popular erlang build tool Rebar is also compatible with EUnit
% ```
% rebar eunit
% ```
```

References

- "Learn You Some Erlang for great good!"
- "Programming Erlang: Software for a Concurrent World" by Joe Armstrong
- Erlang/OTP Reference Documentation
- Erlang - Programming Rules and Conventions