Erlang is a *functional programming language* that supports *concurrent programming*. Computations in Erlang can proceed in parallel on a network of computers, and the language is designed to ensure that no unwanted interactions occur.

The key concepts in Erlang are: *functions*, *single-assignment variables*, *tuples*, *pattern matching*, *race conditions*, *message-passing* and *recursion*.

---

### Functions

Erlang programs are functions that compute a value from some given inputs. Here, we define three functions, `pi`, `circle_area`, and `rect_area`:

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

pi() ->
    3.14159.

circle_area(Radius) ->
    pi() * Radius * Radius.

rect_area(Height, Width) ->
    Height * Width.
```

Erlang programs are organised into modules. The name of the module (which must match the name of the file) appears at the top of the file, followed by a statement of which functions can be called from other modules or from the Erlang command shell. We will usually export all the functions, but it is possible to be more selective.

The `c()` function loads a module into the Erlang shell:

```
Eshell V5.6.5 (abort with ^G)
1> c(areas).
% load areas.erl
{ok,areas}
2> areas:circle_area(10).
314.159
3> areas:circle_area(10) > areas:rect_area(15,15).
true
```

---

### Single-assignment variables

In Erlang, variables start with an Upper Case letter. Names starting with a lower-case letter are constants.

Erlang variables cannot change once they have been assigned a value. Erlang reports a “no match” error if an attempt is made to change the value of a variable.
1> X = 1.
2> X = 2.
** exception error: no match of right hand side value 1
3> 1 = 2.
** exception error: no match of right hand side value 1
4> X = 1.
1
5> 1 = 1.
1

For your convenience (and this is not a feature you can use in your programs!), the Erlang shell (where you type in commands) provides a function f() that removes (“forgets”) all the current variable bindings, leaving their names available for re-assignment. f(Var) “forgets” the variable Var.

---

**Tuples**

Related pieces of data can be grouped using a tuple. Tuples are written as comma-separated values enclosed in curly braces.

1> X = {circle, 10}.
{circle,10}
2> Y = {stack, X, {rect,5,2}}.
{stack,{circle,10},{rect,5,2}}

The empty tuple is written {}, and is actually sometimes useful.

---

**Pattern matching**

The = operator in Erlang is able to select parts of a tuple by pattern matching. A pattern looks like a tuple, but appears on the left-hand side of the =. If a variable appears in a pattern, it is matched against the corresponding part of the value on the right-hand side of the =. Several variables can be matched at a time. For example:

3> {_, _, {rect,Height,Width}} = Y.
{stack,{circle,10},{rect,5,2}}
4> Height.
5
5> Width.
2
The underscore (_) is an *anonymous* variable: Erlang treats each underscore as a fresh, unbound variable — so you don't have to make up names for variables you don't care about.

If the left and right-hand sides of the = don't match, Erlang gives an error.

```
6> \{\text{circle},W\} = X.
** exception error: no match of right hand side value \{\text{circle},10\}
```

It's important to remember that if a bound variable appears in a pattern, it is a value. Hence the error in line 8 below.

```
7> \{_,X\} = \{\text{draw},\{\text{circle},10\}\}.
\{\text{draw},\{\text{circle},10\}\}
8> \{_,X\} = \{\text{draw},\{\text{circle},5\}\}.
** exception error: no match of right hand side value \{\text{draw},\{\text{circle},5\}\}
```

The `case` statement can be used to try several patterns in turn, and selects the first pattern to successfully match. For example, we can match a tuple \(Y\) against tuples used to represent shapes like this:

```erlang
case \(Y\) of
  \{\text{circle},..\} -> \text{simple};
  \{\text{rect},..\} -> \text{simple};
  \{\text{stack},..\} -> \text{complex};
  _ -> \text{unknown}
end
```

We can use pattern matching to write a function that computes the area of various shapes:

```erlang
area( \text{Shape} ) ->
case \text{Shape} of
  \{\text{circle}, R\} -> \pi() * R * R;
  \{\text{rect}, H, W\} -> H * W;
  \{\text{stack}, A, B\} -> area( A ) + area( B )
end.
```

---

**Race conditions**

A *race condition* occurs when the speed at which two separate operations proceed can affect the result. Consider two ticket booths selling tickets for a concert. When a customer phones up, the ticket seller selects a seat, takes the payment, and then updates the seat to 'sold'. If the second ticket seller receives a call while the first is processing the payment, the same seat may be sold twice.
Marking the seat as “unavailable” as soon as a customer phones may still not solve the problem. If the available seats are stored in a central database, the interaction with the database may involve two queries: “find a free seat” and “reserve this seat”. Any delay between finding and reserving a seat will allow a race condition to arise. Since they only occur sometimes, race condition errors can be difficult to find and fix.

Erlang is designed to make it easy to specify composite actions, such as “find a free seat” and “reserve this seat”. It does this by making sure separate operations can only communicate by passing messages to each other. It is up to each operation to decide when to look at it’s messages.

---

**Message passing**

Every Erlang program (or process) is able to send and receive messages with any other process. Each process has its own private mailbox, and the receiving process gets to decide when to fetch its messages and which messages to fetch. Processes can ignore messages, look only for particular messages, and limit how long they wait for a message to arrive.

Messages are sent using the “bang” operator, !. You can (somewhat pointlessly) send a message to the currently running process like this:

```
1> self() ! hi.
hi
```

The message “hi” will remain in the mailbox for the current process (this process identifier is returned by `self()`) until you use `receive` to fetch it.

```
2> receive Msg -> Msg end.
hi
```

If there are no messages, `receive` will wait until one arrives. Since you don’t want to wait indefinitely, you should set a timeout. The timeout value is in milli-seconds (\(\frac{1}{1000}\) second).

```
3> receive NextMsg -> NextMsg after 0 -> no_message end.
no_message
```

```
4> NextMsg.
* 1: variable ’NextMsg’ is unbound
```

Sending messages to yourself illustrates the principles involved, but is otherwise not especially interesting. Sending a message to another process requires knowing the identity of another process, and the simplest means of finding a process identity is to create one. Any function
can be turned into a process, using the built-in \texttt{spawn} function. \texttt{spawn} takes the name of a module, a function name (without the brackets), and a list of arguments for the function (the argument values can’t be passed to the function as they normally are, as \texttt{spawn} needs to set up a new process for the function first).

\begin{verbatim}
1> \texttt{spawn( area, area, [Y] ).
<0.110.0>}
\end{verbatim}

The \texttt{<0.110.0>} is the identifier for the newly created process. You’ll probably see a different identifier.

The \texttt{area} process we just spawned did nothing useful. It calculated the area of \texttt{Y}, and then finished. No result was returned, because processes can only communicate by mail, and this one neither sent nor received any messages.

We can make a talkative \texttt{area_server} like this:

\begin{verbatim}
area_server() ->
    receive
        \{Pid, Shape\} ->
            Area = area(Shape),
            Pid ! Area,
            area_server();
        quit -> ok;
    end.
\end{verbatim}

This new function waits for a message, which is expected to include a “return address” (the \texttt{Pid}) and a \texttt{Shape} (line 3). The \texttt{area_server} then it computes the area of the shape (line 4), sends the result back (line 5), and then continues to listen for further messages (line 6). Sending \texttt{quit} will stop the server (line 7). Any other messages are silently ignored (line 8).

\begin{verbatim}
1> c(areas).
\{ok,areas\}.
2> \texttt{AreaPid = spawn( area, area_server, []).}
<0.126.0>
3> \texttt{AreaPid ! \{self(), \{rect, 10, 10\}\}.}
\{<0.70.0>,\{rect,10,10\}\}
4> receive Ans -> Ans after 2000 -> timeout end.
100
\end{verbatim}

You would not normally go to the trouble of spawning a process to compute a value unless the computation takes a long time and the originating process has other work to do. However, Erlang processes have another purpose: communicating across a network.
Distributed Erlang

You can run Erlang programs on different computers (or “nodes”) and have them talk to each other. The “nodes” can be separate shells running on the same physical computer, different computers in the local network, or computers in entirely different places. The Erlang mechanism remains the same in all these situations, but if the computers are in different places you need to have a suitable authorisation arrangement before the nodes will have permission to communicate.

We will use an example with two computers on the same local network. Provided you login with the same account, Erlang will have permission to communicate.

To start, each Erlang node needs to be given a distinct name, using the \texttt{-sname} (“short name”) flag:

\begin{verbatim}
  vranx> \texttt{erl -sname loki}
\end{verbatim}

Then, open a terminal window and (optionally) log on to another computer:

\begin{verbatim}
  vranx> \texttt{ssh trillian.it.uu.se}
Password: ...
trillian> \texttt{erl -sname sigyn}
\end{verbatim}

In order for Sigyn to send a message to Loki, she needs to know which process running on the Loki machine to talk to. Loki must provide a name for the process that will receive her message, and this is done using the \texttt{register} function. This function needs an existing process; we can use \texttt{self()} to keep things simple:

\begin{verbatim}
  (loki@vranx)1> \texttt{register( loki\_shell, self() ).}
true
\end{verbatim}

Now Sigyn has everything she need to communicate: the name of the machine (\texttt{loki@vranx}) and the named process on that machine (\texttt{loki\_shell}).

\begin{verbatim}
  (sigyn@trillian)1> \{loki\_shell, loki@vranx\} ! hi.
  \{<0.59.0>, hi\}
\end{verbatim}

Loki can now check his mail:

\begin{verbatim}
  (loki@vranx)2> \texttt{receive Msg -> Msg after 0 -> nope end.}
hi
\end{verbatim}
(If you wish to receive another message using the same statement, first use the special \texttt{f(Msg)}. command to “forget” the current value of \texttt{Msg}. If you don’t, then the \texttt{receive} pattern will only match if exactly the same message is sent again).

— John Hamer, 26 June 2012