Erlang Concepts

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**:

```
-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 V5.7.5  (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 convenience, the Erlang shell provides a function **f**() that removes ("forgets") all the current variable bindings, leaving them available for re-assignment. **f**(**Var**) "forgets" the variable **Var**. *Note that this is a convenience feature provided by the Erlang shell; it is not part of the Erlang language, and you cannot use* **f**() *in your programs.*

Tuples

Related pieces of data can be grouped using a *tuple*. Tuples are 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 sometimes useful.

Pattern matching

The = operator in Erlang is able to select parts of a tuple by *pattern matching*, and can bind several variables at the same time.

```
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.

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:

```
case Y of
    {circle,_}   -> simple;
    {rect,_,_}   -> simple;
    {stack,_,_} -> complex;
    _            -> unknown
end
```

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

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

Race conditions

A *race condition* occurs when the speed at which two separate computations proceed can affect the result. Consider two ticket booths selling tickets for a concert. When a customer phones up, the ticket seller selects the best available 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 the best available 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 the best available seat" and "reserve this seat". There are no "global" variables in Erlang, and the only way computations can communicate is by sending messages to each other.

Message passing

Every Erlang program (or *process*) is able to send and receive messages with any other process it knows about. 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, or 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 until you use receive to fetch it.

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

If there are no messages, receive will wait until one arrives. If you don't want to wait, you can 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 spawn function. spawn takes the name of a module (here we use ?**MODULE**, which stands for the current module name), a function name, and a list of arguments for the function.

```
1> spawn( ?MODULE, area, [Y] ).
<0.110.0>
```

The `<0.110.0>` is the identifier for the newly created process.

This **area** process did nothing useful. It calculated the area of **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 **area_server** as follows:

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

This new function waits for a message, which is expected to include a "return address" (the **Pid**) and a **Shape**. The **area_server** then it computes the area of the shape, sends the result back, and then continues to listen for further messages. Sending **quit** will stop the server. Any other messages are silently ignored.

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

You would normally only go to the trouble of spawning a process to compute a value if 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. To start, each Erlang node needs to be given a distinct name, using the `-sname` ("short name") flag:

```
vranx> erl -sname loki
```

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

```
vranx> ssh trillian.it.uu.se
Password:
...
trillian> erl -sname sigyn
```

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, using the register function. This function needs an existing process, and we can use self() to keep things simple:

```
(loki@vranx)1> register( loki_shell, self() ).
true
```

Now Sigyn has everything she need to communicate: the name of the machine (`loki@vranx`) and the process on that machine (**loki_shell**).

```
(sigyn@trillian)1> {loki_shell, loki@vranx} ! hi.
{<0.59.0>, hi}
```

Loki can now check his mail:

```
(loki@vranx)2> receive Msg -> Msg after 0 -> nope end.
hi
```

(If you wish to receive another message using the same statement, first use the special **f**(**Msg**). command to "forget" the current value of **Msg**. If you don't, then the receive pattern will only match if exactly the same message is sent again).

— John Hamer, 27 June 2011