Introduction

Erlang was developed at Ericsson. Since 1998, it is distributed independently with an open source license.

The main web site distributing Erlang is

`http://www.erlang.org`

The implementation is called **Erlang/OTP** (Open Telecommunication Platform).

Introduction (cont)

Sequential programming in Erlang consists in not using the concurrence features. This allows to focus on the kernel of Erlang, which consists in basic functional features.

Erlang is **dynamically typed**, that is, there is no need for the programmer to annotate the code with type informations.

The difference with scripting languages, like Bash, Perl, Python or Ruby, is that in these languages, in case of a type error in an operation, there is always a default value which can be implicitly the result.

In Erlang, a type error at run-time ends the execution of the programs, if not handled.

Introduction/Modules

Following a top-down approach, an Erlang program is first structured in **modules**.

At first glance, a module is a file containing Erlang functions.

The name of the module is given in the file by a special directive (i.e., it is not part of the language itself, strictly speaking) and must be the same as the file name on the underlying operating system.

For example, a file `my_prog.erl` must include, as a first statement, the directive

```
-module(my_prog).
```

Note the starting dash and the final period.

Introduction/Shell and virtual machine

Since Erlang modules only contain functions, they serve as a *database of functions* and something else is needed to use them, just like in a relational database one needs a query language.

This role is devoted to the **Erlang shell**, similar to the command line shells on Unix and Windows operating systems, like, respectively, Bash and DOS.

From this shell, the user can open modules and call functions exported by them.

Introduction/Shell and virtual machine (cont)

In order to run the functions, these must be compiled first to **byte-code**, just like Java for instance.

Second, this byte-code is loaded into the **Erlang virtual machine** (called **BEAM**), a task which is hidden from the user and taken care of by the Erlang shell.

Introduction/First module

Consider the file `math2.erl`:

```
-module(math2).
-export([double/1]).

double(X) -> mult(X,2).
mult(X,Y) -> X * Y.
```

This module exports a list of functions consisting of the lone `double` function, which takes only one argument. This function is defined in the body of the module, together with `mult`, which is not visible from the other modules or the Erlang shell.

Note that the order of definition is not meaningful.

Introduction/First compilation

There are two ways of compiling Erlang modules: from the OS shell or from the Erlang shell.
From the OS shell (whose prompt is here represented by the symbol $), type

```
$ erl
Erlang (BEAM) emulator version 5.5.2 [source] [async-threads:0] [hipe]

Eshell V5.5.2  (abort with ^G)
1> _
```

The Erlang shell prompt is here "1>", where "1" is the command line number (i.e., the first command is expected).

Introduction/From the Erlang shell

Then, the compilation function `c/1` takes the name of the file to compile, e.g. `math2.erl`, launches the compiler and, if no error is found, the resulting byte-code is output in a file `math2.beam`:

```
1> c(math2).
{ok,math2}
2> _
```

This means that the compilation was successful (`ok`) and the result is the file `math2.beam`. The `/1` in `c/1` is the number of arguments (called **arity**).

Introduction/From the OS shell

The other way is just to call the Erlang compiler from the OS command-line shell:

```
$ erlc math2.erl
$ _
```

Introduction/Running the program

After the module has been compiled to byte-code, we need an Erlang shell to use it.

From the Erlang shell, a function is called by giving the name of the module first, then typing in a colon, the function name and finally its arguments, between parentheses and separated by commas.

For instance

```
1> math2:double(5).
10
2> _
```

If there are no arguments, just type the opening and closing parentheses.

Introduction/A classic

By default, all functions can be recursive. Take a look at the classic factorial mathematical function in Erlang:

```
-module(math1).
-export([fact/1]).

fact(0) -> 1;
fact(N) -> N * fact(N-1).
```

Notice how a function can be defined in a manner similar to the mathematical notation:

$$0! = 1$$
$$n! = n \times (n-1)!$$

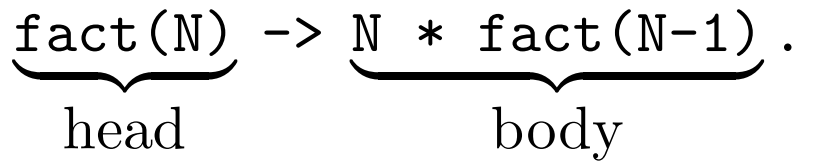
Note that the function names in Erlang (e.g. `fact`) must start in lowercase and the variables (e.g. `N`) in uppercase.

Introduction/A classic (cont)

The factorial function is defined by two cases, called **clauses** in Erlang. Clauses of the same function are separated by semicolons and the last one is terminated by a period:

```
...
fact(0) -> ... ; % First clause
fact(N) -> ... . % Second clause
```

The part before the arrow is called the **head** and the part after it until the semicolon or the period is called the **body** of the function:



Introduction/Clause selection

The order of the clauses matters.

Given a function call and the clauses of the corresponding function, the Erlang run-time

1. computes the arguments in the call;
2. examines the heads of the clauses, in the order of the module, until the head of a clause **matches** the function call;
3. that matching may bind variables to values in the call (in other words: "the parameters are bound to the arguments") and these variables are replaced by their value in the body of the selected clause;
4. the body is evaluated.

Introduction/Clause selection (cont)

For example, consider the function call `math1:fact(0)`.

We do not need here to evaluate the arguments, since `0` is already a value.

The first clause in module `math1` is

```
fact(0) -> 1
```

Its head is exactly the function call: it is a trivial case of matching. There are no variable in the head, so there is no binding as the result of the matching.

Hence we do not need to replace any variable in the body. Finally, the body, `1`, is the result — there is no need to evaluate it, since it is already a value.

Introduction/Clause selection (cont)

Consider now the function call `math1:fact(1)`.

Matching rule #1: *Two values match if and only if they are equal.*

The head of the first clause is `fact(0)` and `0` does not match `1`, so the first clause is skipped.

The head of the second clause is `fact(N)`.

Matching rule #2: *A variable matches any value.*

Therefore `N` matches `1` and the body of the second clause, `N * fact(N-1)`, is selected. The matching leads to bind `N` to `1`.

Finally, `N` is replaced by `1` in the body, which becomes `1 * fact(1-1)`, and is further evaluated as the result (i.e., the value of the function call).

Introduction/Tracing computations

It is possible to summarise the evaluation of a function call by a series of rewrite steps. For instance

$$
\begin{aligned}
\texttt{fact(2)} &\xrightarrow{2} \texttt{2 * fact(2-1)} \\
&\longrightarrow \texttt{2 * fact(1)} \\
&\xrightarrow{2} \texttt{2 * (1 * fact(1-1))} \\
&\longrightarrow \texttt{2 * (1 * fact(0))} \\
&\xrightarrow{1} \texttt{2 * (1 * (1))} \\
&\longrightarrow \texttt{2 * (1)} \\
&\longrightarrow \texttt{2}
\end{aligned}
$$

Note: The number on the arrow is the clause number. An arrow with no number is an elementary arithmetic computation.

Introduction/Why order matters

Let us try the following mistake:

```
-module(math1bis).
-export([fact/1]).

fact(N) -> N * fact(N-1);
fact(0) -> 1.
```

Introduction/Why order matters (cont)

Then let us evaluate

$$
\begin{aligned}
\texttt{fact(2)} &\xrightarrow{1} \texttt{2 * fact(2-1)} \\
&\longrightarrow \texttt{2 * fact(1)} \\
&\xrightarrow{1} \texttt{2 * (1 * fact(1-1))} \\
&\longrightarrow \texttt{2 * (1 * fact(0))} \\
&\xrightarrow{1} \texttt{2 * (1 * (0 * fact(0-1)))} \\
&\longrightarrow \texttt{2 * (1 * (0 * fact(-1)))} \\
&\xrightarrow{1} \texttt{2 * (1 * (0 * (-1 * fact(-1-1))))} \\
&\longrightarrow \ldots
\end{aligned}
$$

This results in an infinite loop because the head of the second clause can never match — and thus terminate the computation.

Introduction/Classic mistakes

Let us try to call an undefined function:

```
1> math:double(5).
** exited: {undef,[{math,double,[5]},
                   {erl_eval,do_apply,5},
                   {shell,exprs,6},
                   {shell,eval_loop,3}]} **

=ERROR REPORT==== 12-Jan-2007::01:46:49 ===
Error in process <0.29.0> with exit value:
{undef,[{math,double,[5]},
{erl_eval,do_apply,5},{shell,exprs,6},{shell,eval_loop,3}]}
```

Similar problem with the call `math2:triple(5)` and `math2:double()` and `math2:mult(5,5)`.

Introduction/Function composition and stop

As in mathematics, function calls can be composed (i.e., nested), as in

```
1> math2:double(math2:double(5)).
20
2> _
```

To stop the Erlang shell, type

```
2> init:stop().
ok
3> $ _
```