



Erlang and Functional Programming

Alessio Bechini Dept. of Information Engineering, Univ. of Pisa

alessio.bechini@unipi.it

© A. Bechini 2021



Outline

Functional programming
features, its use in the
message-passing model,
the Erlang language,
addressing distribution,
scalability, fault tolerance

- Functional Programming:
Principles and concurrency
- Introducing Erlang
- The Actor Model
- Going concurrent & distributed,
actually

© A. Bechini 2021



Functional Programming: Principles and Concurrency

© A. Bechini 2021



Programming Paradigms

A paradigm is a “pattern”, a “way” of *something*, usually of *thinking*.

A programming paradigm is a way of programming:
it influences structure, performance, etc. of a program
and *our ability to reason about it and the problems it solves*.

We focus on two broad programming paradigms:

imperative - sequences of commands drive the control flow

functional - no state mutation,
computation as *evaluation* of expressions/functions

© A. Bechini 2021

Imperative Programming

Computation is intended as a sequence of commands that operate on *state information*: the **effect** of the **execution** of a **statement** is a **modification of the program state**.

The adoption of certain program organization rules led to more specific paradigms: “structured”, “procedural”, “object oriented”...

```
if x>0: result = 15/3
else: result = 2*3
result = result -1
```

Information *updates* lead to the solution

Functional Programming

Typical trait: **lack of state** during computation.

A computation is a **sequence of expressions** resulting from the **evaluation** (and subsequent **substitution**) of sub-expressions.

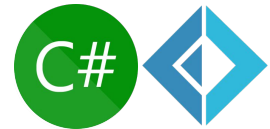
No side-effect in “ideal” computations: the only result is the computed value - even if, in practice, at least I/O is required!

```
(15/3 if x>0 else 2*3) - 1
⇒ (5 if 7>0 else 6) - 1
⇒ 5 - 1
⇒ 4
```

Evaluation + substitution

Functional Programming Adoption

Ideas used in functional programming have found application also in new languages, as well as languages grounded on other paradigms. Some examples:



Concepts in Functional Programming

Some popular concepts in functional programming are not present or are unusual in imperative languages - among them:

- Referential integrity / pure functions
- Lack of state
- Eager / lazy evaluation
- First-class functions & Higher-order functions
- Primary role of recursion
- Use of recursive data structures - lists

Expression & Referential Transparency

Computations are seen as *evaluation of expressions*;

an expression is said **referentially transparent** if it can be replaced by its value *with no change in the program behavior*.

So, such an expression has **no side effects**.

A **function** is said **pure** if its calls are referentially transparent, thus it has no side-effect - a potential target for *memo-ization*.

Whenever ref. transp. holds, it is possible to formally reason on the program as a *rewriting system* (objects + rules to modify them)

→ useful for automatic verification, optimizations, parallelization, etc.

Lack of State

In principle, state information is not kept → no *mutable* variables.

According to this approach, the classical *assignment operation* is not supported in functional programs.

A “variable” (symbolic name) is **immutable**:

once it is bound to a value, such a binding never changes.

We can create new variables, but we cannot modify existing ones.

Typically, *garbage collection* is used to get rid of what cannot be used any more in the computation

Eager vs. Lazy Evaluation

In computations, evaluation of expressions (mainly for function arguments) can be carried out, in different languages/situations, according to different strategies:

Eager evaluation - an expression is evaluated as soon as it is encountered; usually adopted in *call-by-value* and *call-by-reference* semantics of function argument passing.

Lazy evaluation - expression evaluation is postponed to the time its value will be really needed; for actual parameters in functions, this leads to the *call-by-need* argument passing semantics.

First-class & Higher-order Functions

Functions can be handled in the program *as any other value*: so, they can be ordinarily passed as arguments to functions, and can be returned by functions: they are “first-class citizens.”

Higher-order functions are those that can accept functions as arguments and can return functions as result.

As a returned function may depend on actual parameters of the function that generated it, such values have to be kept: this leads to the notion of *closures*.

Erlang uses high-order functions as a prime means of abstraction

Use of Recursion

As is known, recursion and iteration have the same expressive power.

Without relying on state variables, recursion represents the only way to support the repetition of specific computations.

Example,
in Python:
(factorial)

```
def it_fact(n):
    res = 1 #base case
    while n>1:
        res *= n
        n -= 1
    return res
```



```
def rc_fact(n, res=1):
    #res acts as an "accumulator"
    if n <= 1:
        return res #base case
    return rc_fact(n-1, res*n)
```

Tail recursion: for dealing with performance problems and stack limits.

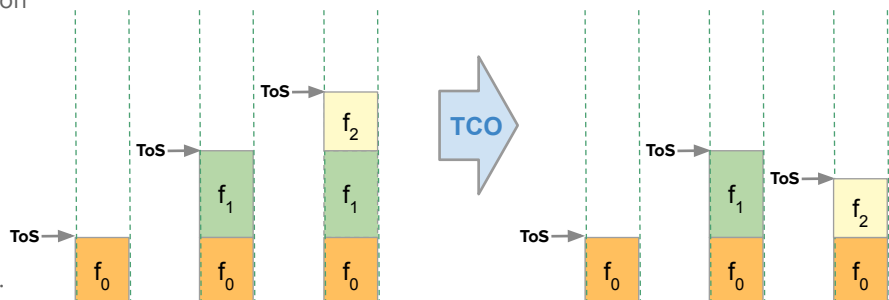
Aside: Tail Call Optimization

Tail recursion can be efficiently handled adopting the so-called "tail call optimization" to avoid excessive growth of the call stack.

Trick: whenever the *last executed* operation within a function call is another function call, we can **substitute** the caller's frame with the callee's frame - no need to keep both!

Example:

f_1 is called inside f_0 ;
 f_2 is called as last op in f_1 .



Use of Recursive Data Structures

Primary role of recursion → widespread use of recursive data structures.

No side effects → no arrays in the language: they keep state info!

Immutable data: inefficiency? → low-level tricks allow data reuse

Basic data structure in most functional languages → **single linked list**

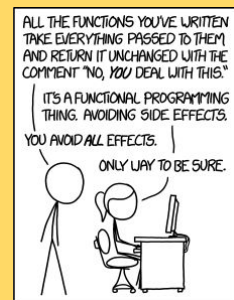


It can be managed *recursively* by telling apart:
the first *element* (head), and the rest (tail) as *a list itself*.

Pause for Thought

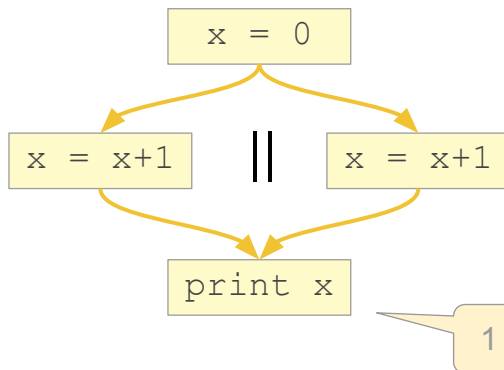


No Side-Effects: What Benefits?



Addressing Concurrency (I)

Main problem in concurrent computations: *data races*



The problem is related to *data mutability*.

Similar considerations apply to issues related to *visibility of updates* (memory consistency models)

© A. Bechini 2021

A. Bechini - UniPi

Addressing Concurrency (II)

Dealing with mutable state requires **synchronization**, which limits resource utilization. In imperative languages, concurrent computations must be explicitly specified.

In pure functional programming:
no state, no side-effects!

No shared mutable state → No problem in concurrent access

→ Thread-safe computations (no critical races)

Simple approach to parallelization:
sub-expressions can be evaluated *in parallel*

Synchronization code
- locking, etc. -
is often error-prone
(e.g. deadlock)
and hampers
parallel executions

© A. Bechini 2021

A. Bechini - UniPi

Addressing Concurrency (III)

But, in practice:

“The trouble is that essentially all the interesting applications of concurrency involve the deliberate and controlled mutation of shared state [...]. The right solution, therefore, is to provide mechanisms which allow the safe mutation of shared state.”

Cit. from paper “Concurrent Haskell”, in POPL 1996

Side effects cannot be avoided in practical programming, and encapsulation can be used also in concurrent settings: A “process” can be considered as a basic program component, and a way to make processes interact has to be provided.

What about Message Passing?

Message Passing can be used also in a functional landscape, with “processes” used as basic components.

Notably, here “processes” refer to *abstract entities*, which could be mapped onto actual computational units, possibly distributed ones.

The Erlang language, built according to the functional programming vision, adopts this vision of concurrency, with “processes” exchanging messages and reacting to receptions of messages.

In principle, concurrent programs written according to this paradigm could be (mostly transparently) run over parallel/distributed platforms.

Introducing Erlang

Erlang: Simple Language Suited to...

- Concurrent apps with fine-grained parallelism
- Network servers, Distributed systems
- Middleware machinery:
 - Distributed databases
 - Message queue servers
- Soft real-time apps; Monitoring, control, and testing tools



The need to address large-scale apps led to →



We'll present only the basic features of the language

System Overview (I)

Erlang code is executed by **ERTS** (Erlang Run-Time System), via an intermediate language (so compilation is needed); bytecode is run **over a dedicated virtual machine (BEAM)**.

Also an implementation over JVM exists (Erjang)

Each Erlang code file (.erl) contains a basic application block (*module*).

Modules are *compiled* (→ .beam files), and *loaded* by BEAM

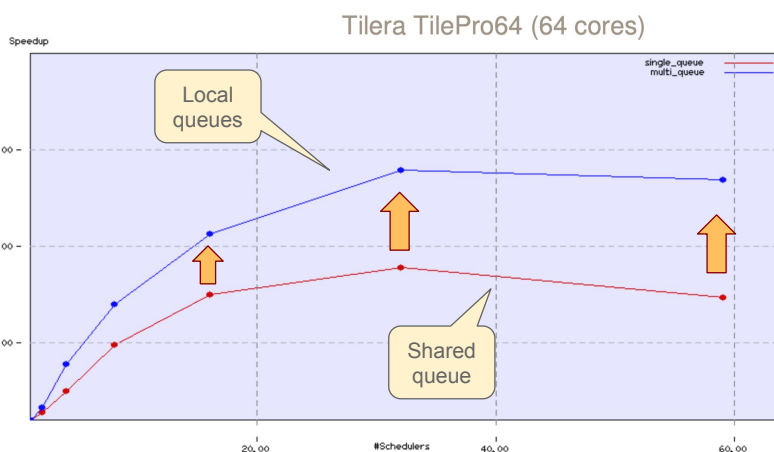
BEAM: one single OS process → concurrency aspects managed **internally**. It uses its own 1+ schedulers (one per CPU core) to distribute the execution of Erlang processes over the “supporting” threads (one per core).

ATTENTION: Erlang processes ≠ OS processes/threads !!!

Interlude 😊 - BEAM Exploits SMP

The BEAM process scheduling has been improved by using one separate process queue per scheduler.

Work stealing is used to improve load balancing.



System Overview (II)

ERTS has garbage collection facilities to handle dynamic memory.

Users can interact with the system by means of the Erlang shell (`erl`).

Compilation+loading can be issued from the shell (`c (Mod)`).

Modules can be *replaced* also as the app is running!

Shell (“repl”): Expressions are evaluated, but functions cannot be defined. Functions exported by modules can be accessed after module loading.

```

alessio@altissimo:~$ erl -root /opt/local/lib/erlang -progname erl -- -ho...
Last login: Fri Mar 20 15:46:10 on ttys004
[(base) Altissimo-MacBook-Pro:~$ erl
Erlang/OTP 21 [erts-10.1] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1] [hipe]

Eshell V10.1 (abort with ^G)
1> 2+3.
5
2>

```

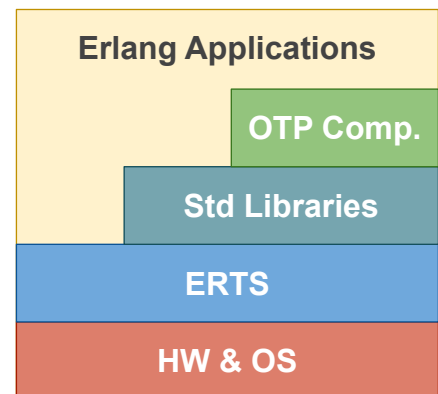
System Overview (III)

Erlang apps are independent of HW/OS, so distributed portions can naturally interact.

Processes can be spawn on different nodes, and inter-process communication is transparent w.r.t placement on nodes.

OTP (Open Telecom Platform) is a set of tools/components/libraries/patterns to boost the development of Erlang apps.

OTP acts as middleware for Erlang apps.



Predictable Prelude: Expressions etc.

An Erlang program is basically made of **expressions**.

The program is run by **evaluating** such expressions, one after the other.

The simplest expression: **term**, i.e. a piece of data of any data type, which can be written by its literal. It evaluates to (“returns”) the term itself.

Expressions can be made of **sub-expressions** combined by **operators**:

All subexpressions are evaluated *before* an expression itself is evaluated, unless explicitly stated otherwise.

A **variable** is an expression. If a variable is bound to a value, the evaluation of the variable is such a value.

Starting Up: Expressions, Numbers

In the shell, an expression must be terminated by “.” + a whitespace char! Multiple subsequent expressions must be separated by “,”

Two types of numeric literals:

- Integers
- Floats

Two Erlang-specific notations:

- `base#value` for ints in any base
- `$char` ASCII/Unicode codepoint for char

```
1> 1+2.  
3  
2> 1+2, 2+3.  
5  
3> 7 rem 3.  
1  
4> 2.3*1.1.  
2.53  
5> 16#A5F2.  
42482  
6> $A.  
65
```

Variables (so to speak)...

Starting with a capital letter, or underscore (_) - mandatory!

Erlang is **dynamically typed** - no static type indication for variables.

A variable can be **bound to a value only once** (a.k.a. “single assignment”), so it actually “doesn’t vary.”

Variables are bound to values by means of **pattern matching** (see later).

A *pattern* is structured like a term, but includes *unbounded variables*.

The anonymous variable (only _) can be used when a variable is required but its value can be ignored.

Pivotal Concept: Pattern Matching

Idea: attempt to make a (right-side) pattern and a (left-side) term *identical*, by possibly binding (to proper values) the unbound variables in the pattern.

Trivial case: use of **match operator** “= ”

Other cases of occurrence of pattern matching:

- Evaluation of a function call
- case- receive- try-expressions

If the matching fails, a run-time error occurs.

```
1> A.  
** 1: variable 'A' is  
unbound **  
2> A = 2.  
2  
3> {A, B} = {0, 1}.  
** exception error: no match  
of right hand side value  
{0,1}  
5> {A, B} = {2, 1}.  
{2,1}  
6> B.  
1
```

Basic Data Types

- **Number** (int, float)
- **Atom**
- no Boolean: instead, atoms `true` and `false`
- Bit string
- Reference (unique term in ERTS)
- **Fun** (a.k.a. “lambda”)
- Port Identifier
- **Pid** (to identify a process)

Compound data types:

- **Tuple** - with fixed number of terms
- **Record** - syntactic sugar for tuple (with named fields), not available in the shell
- **List** - with variable number of terms
- **String** - actually, a list of int
- **Map** - with a variable number of key-value associations



atoms

Atoms are used to represent **constant values**, so that an atom value is unique within the program.

They resemble enumerated types in C/Java.

The value of an atom is just the atom.

Atoms start with **lowercase letters**, followed by alphanumeric chars plus `_` and `@`.

Otherwise, they can be delimited by single quotes `' '`, with any char inside.

```
1> lion.
lion
2> Animal = lion.
lion
3> Animal = tiger.
** exception error: no match of
right hand side value tiger
4> {zoo, Animal1, Animal2} = {zoo,
'zebra joe', 'monkey bob'}.
{zoo,'zebra joe','monkey bob'}
5> Animal1.
'zebra joe'
6> Animal2.
'monkey bob'
```


{ Tuples }

A *tuple* is a compound data type with a *fixed* number of terms (*elements*).

Tuples are used to group up items; similar to structs in C, without named fields.

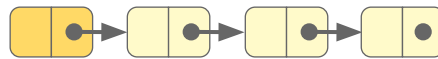
Delimiters: curly brackets - braces; elements separated by commas.

Values can be extracted from tuples by using pattern matching.

“Named” variant of tuples: *records*.

```
1> {alan,turing}.
{alan,turing}
2> Logician = {alonso, church,
{birthyear, 1903}}.
{alonso,church,{birthyear,1903}}
3> {Name,church,{birthyear,1903}}
= Logician.
{alonso,church,{birthyear,1903}}
4> Name.
alonso
5> {_, Lastname, _} = Logician.
{alonso,church,{birthyear,1903}}
6> Lastname.
church
```

[Lists]



A *list* is a compound data type to store an *arbitrary* number of terms (*elements*).

Delimiters: square brackets; comma-separated elements.

Empty list: `[]`

Not-empty list: head (elem) and tail (list): `[H|T]`

“cons”

Note: `[E1, ..., EN]` is thus equivalent to `[E1 | ... | [EN | []]]`

Values can be extracted from tuples by using pattern matching, also in the form `[H|T]`.

```
1> L1 = [bob, 42, {4,2}].
[bob,42,{4,2}]
2> L2 = [3 | L1].
[3,bob,42,{4,2}]
3> [X|Y] = L1.
[bob,42,{4,2}]
4> X.
bob
5> Y.
[42,{4,2}]
6> [X,Z,W] = L1.
[bob,42,{4,2}]
7> W.
{4,2}
8> length(L1).
3
```

Built-in Functions (BIFs)

```
1> date().
{2021,10,6}
2> time().
{14,48,47}
3> Tup = {lion,rhino,ostrich}.
{lion,rhino,ostrich}
4> element(2,Tup).
rhino
5> Lis = tuple_to_list(Tup).
[lion,rhino,ostrich]
6> length(Lis).
3
7> io:format("hello~n").
hello
ok
```

Some operations cannot be developed using the basic Erlang constructs, or at least not in a very efficient way.

For this reason, several “built-in” functions (within BEAM) have been made available; some are “auto-imported.”

They are typically used for system access, data conversion, efficient handling of compound data, I/O, etc.

List Comprehensions

Compact notation for generating elements in a list according to specified rules.

[Expr || Qualif1, ... QualifN]

Idea: qualifiers specify what values to consider, and such values will be used in Expr to construct list elements.

Qualifiers:

- Generators - Pattern <- ListExpr
- Filters - expr. that evaluate to true/false
- Bit string generators - not discussed here

```
1> L1 = [1,2,3,4,5].
[1,2,3,4,5]
2> [ X*X || X <- L1 ].
[1,4,9,16,25]
3> [ X || X <- L1, X rem 2 /= 0 ].
[1,3,5]
4> L2 = [0,1].
[0,1]
5> [ {X,Y} || X<-L2, Y<-L2 ].
[{0,0},{0,1},{1,0},{1,1}]
6> L3 = [a,{3,1},0,{2,2},L1].
[a,{3,1},0,{2,2},[1,2,3,4,5]]
7> [ X+Y || {X,Y}<-L3 ].
[4,4]
```

What about Strings?

No special data type for strings;
A string can be represented
as *list of integers*, each element
corresponding to a Unicode codepoint.

(Strings can be represented also as *binaries*,
but this is not discussed here.)

Special syntax for strings handling:
“double quote” delimiters are used.

```
1> S1 = "Hello".
"Hello"
2> S2 = S1 ++ " World!". %str-list concat
"Hello World!"
3> S2 ++ [10]. %10: newline
"Hello World!\n"
4> S2 ++ [-3].
[72,101,108,108,111,32,87,111,114,108,100,
33,-3]
5> S3 = "\x{2200} e \x{2208} A \x{2a01}
B".
[8704,32,101,32,8712,32,65,32,10753,32,66]
6> io:format("~ts~n",[S3]).
∀ e ∈ A ⊕ B
ok
```

Modules

A module, contained in a .erl file,
is the **basic unit of code**.

It contains *metadata* (for the module itself),
plus *functions*.

Most important metadata:

- module name (same as file)
- what functions can be called from
outside the module
(i.e. what functions are *exported* - “APIs”).

Modules **have to be compiled**.

```
-module(myfirstmod).
%% API
-export([sayhello/0,addinc/2]).

add(X,Y) ->
    X+Y.

%% not exported!
addinc(X) ->
    X+1.
addinc(X,Y) ->
    add(addinc(X), Y).

sayhello() ->
    io:format("Hello World!~n").
```

myfirstmod.erl

Compiling & Using Modules

From the command line →

```
erlc [flags] myfirstmod.erl
```

From inside a module or the repl →

```
compile:file("myfirstmod.erl")
```

From the repl → `c(myfirstmod)`

```
1> ls().
myfirstmod.erl
ok
2> compile:file("myfirstmod.erl").
{ok,myfirstmod}
3> c(myfirstmod).
{ok,myfirstmod}
4> ls().
myfirstmod.beam      myfirstmod.erl
ok
```

Functions in a module

can be used *externally* with the syntax: `mymodule:myfunction(...)`

Importing specific functions: `-import(mymodule, [myfunct/1 ...])`

Functions - Basics

Function declaration (only in a module):

sequence of **function clauses**, separated by “,” and terminated by “;”

J-th function clause:

Name (**PattJ1**, ..., **PattJN**) [when GuardSeqJ] -> **BodyJ**

Function **name**: atom

Function **arguments**: patterns

Within a module, a function is identified by the couple name/arity.

Clause **body**: sequence of expressions separated by “;”;
a clause evaluates to the value of the last expression in its body.

Functions - Example Def/Call

shapes.erl

```
-module(shapes).
-export([area/1]).

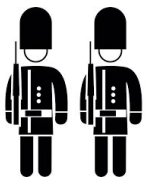
area({square,Side}) ->
    Side*Side;
area({circle,Radius}) ->
    Radius*Radius*3.1415;
area({triangle,B,H}) ->
    B*H/2.

...
```

shell

```
1> c(shapes). %compile module
{ok,shapes}
2> shapes:area({square,3.2}).
10.240000000000002
3> shapes:area({circle,3.2}).
32.168960000000006
4> shapes:area({triangle,3.2, 2}).
3.2
5> shapes:area({rectangle,3.2, 2}).
** exception error: no function
clause matching
shapes:area({rectangle,3.2,2}) (...)
```

“.” specifies
the *module* containing
the *exported function*
to use



Guards

Generally speaking, a guard is a boolean expression that affects the way a program is executed.

In Erlang, guards can be used *to increase the power of pattern matching*, in particular in heads of function clauses.

Formally: a *guard* is a sequence of *guard expressions*; the set of guard expressions is a subset of valid Erlang expressions (see docs).

Attention: No user-defined function is allowed in guards (to be sure to avoid side-effects.)

```
sign(X) (when is_number(X), X>0) ->
    1;
sign(X) when is_number(X), X==0 ->
    0;
sign(X) when is_number(X) ->
    -1;
sign(_) -> argerror.
```

“Case” Expressions...

Sometimes, instead of relying on pattern matching on many clauses, it may be convenient using *case expressions*.

```
case Expr of
  P1 [when C1] -> E1;
  :
  Pn [when Cn] -> En
end
```

Semantics: expression `Expr` is evaluated to `T`, and patterns `P1 ... Pn` are sequentially matched against `T`. As soon as a match occurs and the (optional) relative guard is true, the corresponding `E` is evaluated, and the return value of `E` becomes the return value of the case expression.

!!! - No matching pattern with a true guard sequence →
→ `case_clause` run-time error (to avoid this, catch-all clauses are often used.)

... and “If” Expressions

Another conditional expression is provided: “if”
- to be used in case the resulting term must depend only on guards.

```
if
  Guard1 -> E1;
  :
  Guardn -> En
end
```

Semantics: Guards `Guard1 ... Guardn` are sequentially checked; as soon as a guard succeeds, the value of the whole if expression is the return value of the relative `E` expression sequence.

No guard succeeds → `if_clause` run-time error.

Note: “if” clauses are usually called “branches.”

!!! - “**if**” in Erlang is an expression; to avoid a possible exception, often a final `true` guard is inserted.

Looping? What?

Repeating calculations in functional languages is usually performed **by means of recursion**: *no basic looping construct is provided!*

Recursion is particularly suited to work on lists.

Iteration: seen as a particular type of recursion

This recursive function mimics the “for” behavior:
It implements LINEAR ITERATIVE execution

The use of **tail recursion** improves the execution performance, and makes extensive looping actually feasible.

```
-module(helloworld) .
-export([forhw/1,start/0]) .

forhw(0) -> done;
forhw(N) ->
    io:fwrite("Hello~n"),
    forhw(N-1) .

start() ->
    forhw(5) .
```

Example: Looping over a List

```
-module(showlist) .
-export([scanl/1,start/0]) .

scanl(L) -> scanl(L,0) .

scanl([], Index) -> Index;
scanl([H|T], Index) ->
    io:fwrite("~w: ~w~n", [Index,H]),
    scanl(T, Index+1) .

start() ->
    X = [10,2,7,4],
    scanl(X) .
```

The “Head-Tail” structure of a list helps us work *recursively* on the data structure.

Just in case, additional “state variables” can be added in *helper* functions.

Usually, helper functions are not exported, to promote module-level encapsulation.

scanl/2 is a helper function, with the additional argument used to indicate the position in the list

Having Fun with Funs

Higher-order functions work with functions as parameters/returned values. A data type for functions is needed: in Erlang, it is called “**fun**”.

General syntax of an *anonymous* function →

Multi-clause `fun`s can be defined as well, e.g.:

```
fun(Arg1, ... ArgN) ->
  FunBody
end
```

Like “lambda” in Python and other languages

Syntax to refer to values of NAMED functions

```
fun Module:Function/Arity
```

```
1> TempConv = fun({cel,C}) -> {far, 32 + C*9/5};
1>           ({far,F}) -> {cel, (F-32)*5/9}
1>           end.
#Fun<erl_eval.6.128620087>
2> TempConv({cel,22}).
{far,71.6}
3> TempConv({far,0}).
{cel,-17.77777777777778}
```

© A. Bechini 2021

A. Bechini - UniPi

Funs for Building Control Abstractions

A control abstraction has to refer to an operation to be executed according to some given rules.

The way to apply the rules can be specified *by a function*, and the generic operation *by a Fun as parameter*. E.g.:

```
-module(mycontrols).
-export([myfor/3, test/0]).

myfor(Max,Max,Oper) -> [Oper(Max)];
myfor(I,Max,Oper) ->
  [Oper(I) | myfor(I+1, Max, Oper)].

test() -> myfor(1,5, fun(X) -> 2*X end).
```

```
1> mycontrols:test().
[2,4,6,8,10]
2> mycontrols:myfor(1,3, fun(Z) -> Z*Z/2 end).
[0.5,2.0,4.5]
3>
```

© A. Bechini 2021

A. Bechini - UniPi

Example: Filtering a List

```
...
myfilter( _ , [] ) -> [];
myfilter(Pred,[H|T]) -> case Pred(H) of
  true  -> [H|myfilter(Pred,T)];
  false -> myfilter(Pred,T)
end.

is_even(X)-> case (X rem 2) of
  0 -> true;
  _ -> false
end.

test() -> myfilter(fun is_even/1,
  [1,2,3,4,5]).
```

The anonymous variable “_” is used to match anything, when we don’t care about the match

In this case, a *predicate* is used as the first argument for “myfilter/2”.

Inside “test/0,” function “is_even/1” (actually, a predicate) is passed as the actual parameter for “myfilter.”

The standard module **lists** contains several functions to operate on lists (and *lists:filter* as well)

Example: Returning a Function

Functions can be returned by functions as well.

In the example, **fcomp** returns the composition of two functions (with arity 1) passed as input.

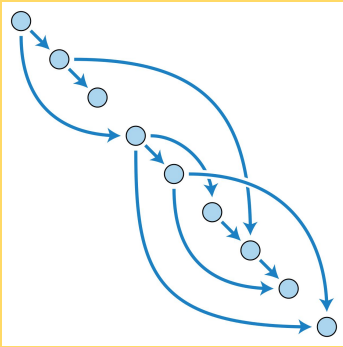
Function values as params

```
-module(mymod).
...
fcomp(F,G) ->
  fun(X) -> F(G(X)) end.
...
```



```
1> Myf1 = mymod:fcomp(fun(X)->X*X end,
fun(X)->X+1 end).
#Fun<mymod.1.70048917>
2> Myf2 = mymod:fcomp(fun(X)->X+1 end,
fun(X)->X*X end).
#Fun<mymod.1.70048917>
3> io:format("~p~n", [Myf1(5)]) .
36
ok
4> io:format("~p~n", [Myf2(5)]) .
26
ok
```

Pause for Thought

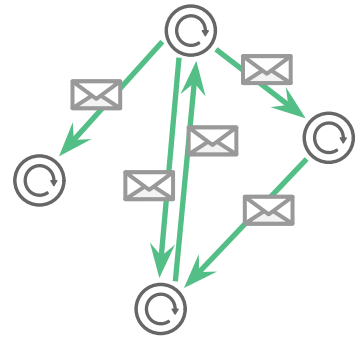


Moving
from Sequential
to Concurrent

Erlang Concurrency & Distribution

Concurrent, but How? The Actor Model

- The fundamental computational unit is an **actor**.
- Any computation involves 1+ actors.
- An actor **shares nothing** with other actors.
- Actors can communicate by **asynchronous message passing**.
- Message addressing: via Actor IDs.
- Actor creation: by another actor
→ variable topology.
- “Reactive” behavior upon message receiving.



Re-acting Actors

Actor's *behavior*: what an actor does in processing a received message.

Possibilities: upon receiving a message, an actor can concurrently:

- send a finite number of messages to other actors;
- create a finite number of new actors;
- designate the behavior to be used for the next message it receives.

In practice, we have to better specify how the overall receive operation has to be structured.

Erlang Ingredients for Concurrency

Erlang follows a version of the actor model; actors are named “processes”. In the language, only three *concurrent* constructs are required, namely for:

1. Creating processes → **spawn** (. . .) BIF, with arguments that identify a function (+ actual argument values) that corresponds to the process body; **spawn** (. . .) returns the process **PID**, used as its address;
2. Sending messages → ! “bang” operator. e.g. **self()** ! **emptymessage**.
3. Receiving messages → **receive** ... **end**, which applies pattern matching (as for **case** ... **end**) to select the message to be received/processed.

self() returns the PID of the current process

Spawning of Processes

```
-module(test) .
-export([body/2]) .
body(X,Y) ->
  io:format("~w+~w=~w~n", [X,Y,X+Y]) .
```

```
1> F=fun(X,Y)->io:format("~w+~w=~w~n",
[X,Y,X+Y]) end.
#Fun<erl_eval.12.128620087>
2> [spawn(fun()->F(A,A+1)end) || A<-[3,4]] .
3+4=7
4+5=9
[<0.134.0>,<0.135.0>]
3> spawn(test,body,[1,2]) .
1+2=3
<0.137.0>
```

A process body can be specified by

- *Fun*: **spawn/1**. Arg: a **fun()**.
- *Function* (“MFA”): **spawn/3**. Args: Module, Function (atoms), and the list of function Args. Used for dynamic code loading.

EXPORTED function

Returned Pid: what node (0: local), and a process counter - in two parts.

Process end: at its body function end.

Each process has its own “dictionary” (BIFs: get, put, get_keys, erase).

Possibly, “write once”

Messages, and Sending thereof

In Erlang, communication is performed by asynchronous signalling (most common signals: **messages**).

A message has a *recipient* (by Pid) and a *content*.

Content: *any* Erlang term.

What *ordering guarantee*?

Only point-to-point FIFO, for all signals.



Non-blocking send: The expression

Pid ! Msg

recipient

content

evaluates Msg to term T, *sends* it to process Pid, and *returns* T.

Send is right-associative: e.g.

Pid2 ! Pid1 ! Pid0 ! `examplemsg`

`atom examplemsg` is sent to Pid0, Pid1, Pid2

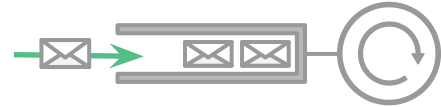
...and Other Signals...

- Exit
- Link/Unlink
- Monitor/Demonitor

Signals to a terminated receiver: possibly trigger other signals (typically, are silently ignored).

Receive Machinery

Every process has a mailbox (queue) to hold incoming messages.



A mailbox can be inspected by the BIF `process_info(Pid, messages)`

To extract messages from the mailbox: *receive expression*

```
receive
  P1 [when C1] -> E1;
  :
  Pn [when Cn] -> En
end
```

By sequentially going through the clauses, it is selected the oldest term T (message) in the mailbox that:

- matches a pattern `Pk`, and
- satisfies condition `Ck`.

...and the message is removed from the mailbox!

If such a term T exists, `Ek` (evaluated to `Tk`) is returned;

otherwise, evaluation **blocks** until a suitable message arrives.

Example: Ping Pong (code)

```
-module(pp) .
-export([start/0, alice/2, bob/0]).

alice(0, Other_PID) ->
  Other_PID ! finished, %terminates bob
  io:format("Alice finished~n");
alice(N, Other_PID) ->
  Other_PID ! {ping_msg, self()},
  receive
    pong_msg ->
      io:format("Alice received pong~n")
  end,
  alice(N - 1, Other_PID). %last call opt.

...
```

```
...

bob() ->
  receive
    finished ->
      io:format(" Bob finished~n");
      {ping_msg, Other_PID} -> %pttrn with Pid
      io:format(" Bob received ping~n"),
      Other_PID ! pong_msg,
      bob() %last call optimization!
  end.

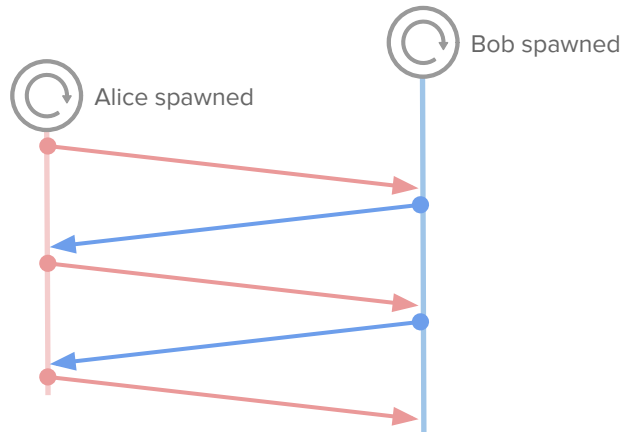
start() ->
  Bob_PID = spawn(?MODULE, bob, []),
  spawn(?MODULE, alice, [2, Bob_PID]).
```

MACRO: current module

Example: Ping Pong (run)

```
1> pp:start().
    Bob received ping
    Alice received pong
    Bob received ping
    Alice received pong
    Alice finished
    Bob finished
<0.79.0>
2>
```

...what Pid is this???



© A. Bechini 2021

A. Bechini - UniPi

Basic Client-Server (I)

A *server process* is intended to perform (repeatedly) the following operations:

1. Wait for a request;
2. As it comes, compute the answer, and
3. Reply back with the answer.

The basic code can be arranged as indicated aside:

The used msg structuring:
just for the sake of clarity → { PID, { <payload> } }

For dynamic code loading:
`server:loop()`

```
-module(server).
-export([start/0, loop/0]).

start() -> spawn(?MODULE, loop, []).

loop() ->
    receive
        {From, {plus, X, Y} } ->
            From ! {self(), X+Y},
            loop();
        {From, {minus, X, Y} } ->
            From ! {self(), X-Y},
            loop();
    end.
```

Msg body with {Sender, opID, params}

Different "services"

Reply back, with sender ID as well

© A. Bechini 2021

A. Bechini - UniPi

Basic Client-Server (II)

```
-module(client).
-export([start/4, body/4]).

start(ServID, OpID, X, Y) ->
  spawn(?MODULE, body, [ServID, OpID, X, Y]).

body(ServID, OpID, X, Y) ->
  ServID ! {self(), {OpID, X, Y}},
  receive
    {ServID, Result} ->
      io:format("Result: ~p~n", [Result])
  end.
```

Only replies from the server are received!

Basic operations for a client:

1. Send out a request to a server;
2. Receives back the reply.

← Basic code indicated aside.

```
1> S = server:start().
<0.77.0>
2> C = client:start(S, plus, 2, 2).
<0.79.0>
Result: 4
3> exit(S, kill).
true
```

Requested service: sum

Command to kill a running process

shell

Basic Client-Server (III)

```
-module(client).
-export([start/4, body/4]).

start(ServID, OpID, X, Y) ->
  spawn(?MODULE, body, [ServID, OpID, X, Y]).

rpc(ServID, Msg) ->
  ServID ! {self(), Msg},
  receive
    {ServID, Result} -> Result
  end.

body(ServID, OpID, X, Y) ->
  Result = rpc(ServID, {OpID, X, Y}),
  io:format("Result: ~p~n", [Result]).
```

Often, communication details **can be abstracted away** within functions, e.g. on client side:

```
1> S = server:start().
<0.77.0>
2> C = client:start(S, plus, 2, 2).
<0.79.0>
Result: 4
3> exit(S, kill).
true
```

shell

Same as before

Basic Client-Server (IV)

At server side, we should deal also with incoming messages with unexpected format which, otherwise, would clutter the server mailbox!

Trick to terminate a running process (a.k.a. "poison pill")

"Catch-all" clause:
Just discard messages with wrong format

```
...
start() -> spawn(?MODULE, loop, []).

loop() ->
  receive
    {From, {plus, X, Y} } ->
      From ! {self(), X+Y},
      loop();
    ...
    stop ->
      ok;
    _Unexpected ->
      loop()
  end.
```



Receive Glitches & Timeouts

In some unexpected situations, a blocking receive operation may hamper the program progression. This may happen, e.g., in presence of server crashes.

```
1> S = server:start().
<0.77.0>
2> exit(S, kill).
true
3> client:rpc(S, {plus, 1, 2}).
```

**SHELL
BLOCKED!**

```
receive
  P1 when C1 -> E1;
  :
  Pn when Cn -> En
after Delay -> ExprDel
end
```

Possible solution:
a TIMED version of receive;
in case no message is received by a given timeout (in milliseconds), a dedicated expression is evaluated instead.

Example with Timeout = 0

Also a timeout of 0 milliseconds may be useful.

Example:

a function to flush all the messages in the mailbox.

Recursively, it tries to match (and “consume”) any message; if no message is present, the 0 timeout terminates the recursive calls.

```
flush_messages() ->
  receive
    _Any ->
      flush_messages()
  after 0 ->
    true
  end.
```

The value `infinity` can be used for a timeout as well; it makes sense when the delay value has to be calculated/decided, and one of the possible options is having no timeout at all.

Semantics of Receive, Detailed (I)

1. Entering a receive statement, we start a timer (only if “after” is present).
2. Take the first message in the mailbox, and try to match it against the first pattern, then the second pattern, and so on. As soon as a match occurs, the message is removed from the mailbox, and the relative expressions are evaluated.
3. In case of no match for the first message in the mailbox, it is removed from the mailbox and put into a **“save queue.”** Then, the second message in the mailbox is tried. This procedure is repeated until:
 - a. a matching message is found, or
 - b. all the messages have been examined.

Semantics of Receive, Detailed (II)

4. If none of the messages matches, then the process is suspended, and will be rescheduled for execution at the time another message is put in the mailbox. When a new message arrives, the messages in the save queue are not rematched; only the new message is matched.
5. As a message match occurs, then all messages previously stored in the save queue are moved back to the mailbox, keeping their order of arrival. The timer (if any) is cleared.
6. If the timer elapses when we are waiting for a message, then evaluate the relative expressions; move all the saved messages back to the mailbox, keeping their order of arrival.

© A. Bechini 2021

A. Bechini - UniPi

Publishing of Processes

In Erlang, a Pid can be *published* (in a system repository), so to make any process in the system able to communicate with the relative process.



Such a process is called a *registered process*.

Only special-purpose processes should be registered!

Registration operations: by four BIFs

- **register**(AnAtom, Pid) - Register the process Pid with the name AnAtom.
- **unregister**(AnAtom) - Remove any registrations associated with AnAtom.
- **whereis**(AnAtom) -> Pid | undefined-
Lookup for the Pid of the AnAtom process
- **registered**() -> [AnAtom::atom()] Return a list of all registered processes.

A registered process is automatically unregistered as it dies

© A. Bechini 2021

A. Bechini - UniPi

Example: Registered Server

We can show how to deal with registered processes using the code developed before for basic server & client:

"send" can use the server name (an atom) as destination

Reply message (as flushed out of the mailbox)

In rpc, the PID is required for the selective receive

```
1> S = server:start().
<0.77.0>
2> register(calc_server, S).
true
3> calc_server ! { self(),
{plus,1,1} }.
{<0.75.0>,{plus,1,1}}
4> flush().
Shell got {<0.77.0>,2}
ok
5> client:rpc(whereis(calc_server),
{minus, 10,7} ).
3
```

Server registration

Keeping State at Server, Functionally

Often a service offered by a server has to exploit *state information*.

How to keep state in a server loop?
Functionally speaking, in 1+ parameter(s).

Example: a server that keeps the count of the number of executed loops.

The required state is initially obtained as a parameter (N), and the next recursive call makes use, for such a parameter, of a new calculated value (N+1).

```
-module(stserver).
-export([start/0, loop/1]).

start() -> spawn(?MODULE, loop, [0]).

loop(N) ->
  io:format("Loop nr.~p ~n", [N]),
  receive
    {From, {plus, X, Y} } ->
      From ! {self(), {X+Y, N} },
      loop(N+1);
    ... <other receive clauses>
  end.
```

Going Distributed, Actually

Erlang Distributed Applications

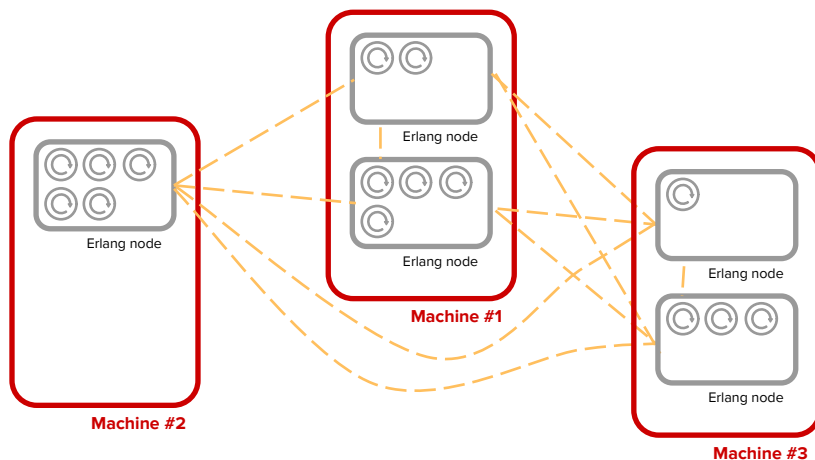
In Erlang, distributed programming is enabled by the possibility to spawn processes on remote nodes and machines.

There exist two main ways to develop distributed programs:

- **Distributed Erlang** - programs run on *Erlang nodes*, i.e. separate full-featured Erlang systems, with their own set of processes. We can spawn a process on any node, and processes can interact across nodes using the usual message-passing constructs.
- **Socket-based distribution** - more secure, to get more control on what a remote “client” can do in a node. We’ll skip this part in our description.

Nodes and Machines

1+ Erlang nodes can be hosted on a machine.



Node Naming and Addressing

Nodes must be given **unique** names, to be located and contacted.

Form of node names: **Name@Host**

Host: available DNS entries, + /etc/hosts

Optional

Two name types:

- Long names: with fully qualified domain names
- Short names: host w/o a “.”

```
erl -sname ale@localhost
(ale@localhost)1>
```

Only one type can be used in a single cluster.

EPMD (Erlang Port Mapper Daemon, on port 4369) runs on each of the machines in the Erlang cluster, acting as a name server, enabling contacts across nodes.

Connecting Nodes

Interaction between two nodes is possible only after a connection has been setup.

```
erl -sname tom@localhost
(tom@localhost)1>
```

Nodes get connected the 1st time they “try” to communicate: e.g.

Another way:
`net_adm:ping(Node)`

```
erl -sname ale@localhost
(ale@localhost)1> net_kernel:connect_node(tom@localhost).
true
(ale@localhost)1> nodes().
[tom@localhost]
```

“Transitive connections”: If node X connects to node Y, by default it obtains also the connection with all the nodes Y is connected to.

Going Distributed, Gradually

According to Joe Armstrong’s suggestion, it is recommendable refining a distributed application in three steps:

Write and test the program in a single, ordinary Erlang system

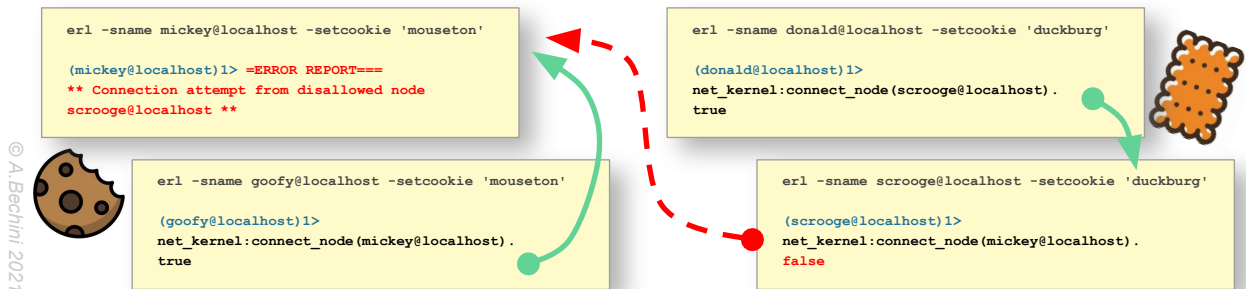
Test the program on two Erlang nodes on the same machine

Test the program on two Erlang nodes on two distinct machines

Keep your own Cookie!

Nodes of different clusters on the same HW must be kept separated.

Simple solution: one special value (cookie) is shared by all the nodes of the same cluster. Nodes with different cookies are not allowed to connect.



© A. Bechini 2021

A. Bechini - UniPi

Spawning, Remotely

Processes can be spawned on remote nodes by means of `spawn/2` and `spawn/4`: they behaves like classical `spawn/1` and `3`, but the node name must be added as 1st argument.

This is useful to:

- spawn remote servers
- execute remote commands
- ...

Returned Remote Pid

```

erl -sname mickey@localhost -setcookie 'mouseton'
(mickey@localhost)1> net_adm:ping(goofy@localhost) .
Pong
(mickey@localhost)1> spawn(goofy@localhost,
fun() -> io:format("~w\n", [node()]) end) .
goofy@localhost
<8502.88.0>

```

```

erl -sname goofy@localhost -setcookie 'mouseton'
(goofy@localhost)1>

```

© A. Bechini 2021

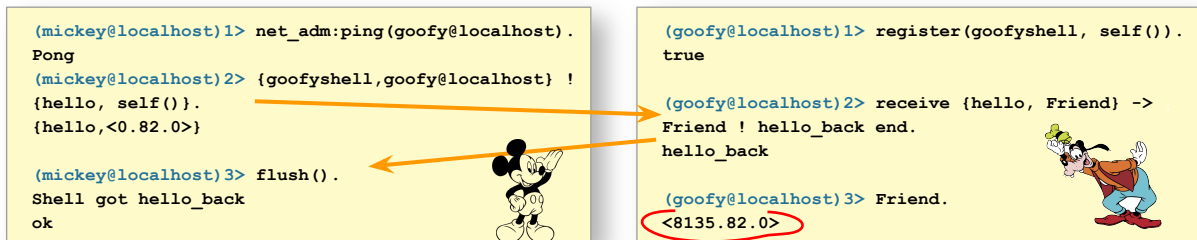
A. Bechini - UniPi

Communicating, Remotely

Send: Expr1 must evaluate to a pid (remote...),
a registered name,
or a tuple {RegisteredProcName, NodeName}.

Expr1 ! Expr2

Receive: as usual.



© A. Bechini 2021

A. Bechini - UniPi

Modules to Support Distribution

Modules provided to support distributed Erlang programs:

net_kernel - to connect/disconnect nodes, to switch a node
to distributed/non-distributed, to control heartbeat..

global - a *global* process registry; → use `global:send(Name, Msg)`;
handles name conflicts

rpc - functions to execute commands on remote nodes →
`rpc:call(Node, Mod, Funct, Args)` returns *locally* whatever is
returned by the function executed remotely; `rpc:call/5` w/ timeout;
support to *promises* for asynchronous computing

© A. Bechini 2021

A. Bechini - UniPi

OTP - Open Telecom Platform

OTP is a collection of supporting tools and libraries for boosting development of Erlang programs.

Central concept for application development: OTP *behavior*.

A behavior encapsulates common behavioral patterns, i.e. it is an *application framework* to be parameterized by a *callback* module.

Idea:

- Non-functional issues
- Specific Functional issues



Behavior

Callback (custom)

Aside: What's a “Callback” ???

Sometimes, the term “callback function” is used in a confusing way.

Regardless of the specific programming language:

a **callback function** is just a function f_c passed (somehow...) by a *caller* to a *called* function, so that in turn f_c could be called/executed *later*.

Important: the actual callback function must be identified via actual arguments, and sometimes also according to **naming conventions**.

- *Synchronous* callbacks - executed before the called function will return
- *Deferred* callbacks - executed after; *used for event handling*, GUIs, etc.

Parameterizing a Server - Behavior

Specific functionality must be specified apart in module `Mod`, which must contain `init()` and `handle(Req, State)`.

Standard API to get a service

To be computed: Response & updated state

```
-module(server_0).
-export([start/2, rpc/2]).

start(Name, Mod) ->
    register(Name, spawn(fun() -> loop(Name, Mod, Mod:init()) end)).

rpc(Name, Req) ->
    Name ! {self(), Req},
    receive {Name, Response} -> Response
    end.

loop(Name, Mod, State) ->
    receive
        {From, Req} ->
            {Response, NewState} = Mod:handle(Req, State),
            From ! {Name, Response},
            loop(Name, Mod, NewState)
    end.
```

Module with required request-handling function

Keep: Name, Mod, and (evolving) state

© A. Bechini 2021

A. Bechini - UniPi

Parameterizing a Server - Callback

Here we specify a server that receives one number per request, and returns the average of all the values received so far.

Standard API to get a service

Returns a tuple with response and updated state

```
-module(avg_server).
-export([init/0, handle/2, get_avg/1]).
-import(server_0, [rpc/2]).

% client routines
get_avg(X) -> rpc(calc_avg, X).

% callback routines
init() -> {0, 0}. % total: 0; # of reqs: 0

handle(X, {Total, Times}) ->
    NewTotal = Total + X,
    NewTimes = Times + 1,
    {NewTotal/NewTimes, {NewTotal, NewTimes}}.
```

Server name to be registered

Initial state

State info: Total, and # of calls

© A. Bechini 2021

A. Bechini - UniPi

Setup & Use the Custom Server

Name to be registered for the server

```
1> server_0:start(calc_avg, avg_server).
true

2> avg_server:get_avg(4).
4

3> avg_server:get_avg(6).
5

4> avg_server:get_avg(11).
7
```

Callback module

Requests by clients

Server setup: call `start/2` from the behavior, providing:

- standard name for server
- reference to the callback module.

The server is used via client routines, defined in the callback module.

Focus on: *callback module*.

OTP gen_server

The OTP behavior `gen_server` adopts the previously adopted approach.

To build up a server, `gen_server` must be paired to a callback module with standard API, to deal with any possible piece of required functionality (initialization, handling of rpcs, handling of “casts”, termination, etc.).

The callback module must specify `-behaviour(gen_server)`.

gen_server module	callback module
<code>gen_server:start, start_link, start_monitor</code>	<code>mod:init/1</code>
<code>gen_server:stop</code>	<code>mod:terminate/2</code>
<code>gen_server:call, send_request, multi_call</code>	<code>mod:handle_call/3</code>
<code>gen_server:cast, abcast</code>	<code>mod:handle_cast/3</code>
others	



OTP gen_server Example

A practical example is present in the lab exercises.



Data Storing: ETS & DETS

ets and **dets** are system modules to store Erlang terms.

Both provide *{key-value}* lookup tables: ETS on memory, DETS on disk.

Tables are collection of tuples; the first element acts as key.

- Sets - no duplicate keys; also “ordered sets” are available
- Bags - duplicate keys allowed, but not tuples; in “duplicate bags” also duplicate tuples can be present.

Operations: creation/opening, insertion, lookup, disposal.

Further details on documentation pages.



Integrated Data Management: Mnesia

ets and **dets** are system modules to store Erlang terms.

Erlang programs that need efficient and more complex data management can rely on an integrated Data Base System: **Mnesia**.

It can store any kind of Erlang data structure.

In a mnesia db, data is organized in tables; query language capabilities (as in SQL) are available in ordinary Erlang syntax.

Mnesia provides transaction support as well.

© A. Bechini 2021

A. Bechini - UniPi



Dealing with Errors in Erlang

© A. Bechini 2021

Handling of Runtime Errors in Erlang

Erlang is intended to support the development of fault-tolerant systems, thus **runtime** errors must be handled (i.e., detected, corrected, etc.).

Specific mechanisms have been introduced for different scenarios:

- **Error Handling in SEQUENTIAL programs** - we need to understand when, within a process, something goes wrong (a function “crashes”), and to specify what to do in such exceptional cases.
- **Error Handling in CONCURRENT programs** - we need to deal with errors at the process level, in collection of processes. In case of a process crash, we want another process may detect the event, and take over the job of the crashed process.

Typical case in fault-tolerant systems

Raising Exceptions, Explicitly

An error can be explicitly generated with these BIFs (one per error type):

- **error(Why)** - used for denoting *severe errors* (“crashing errors”), i.e. programmatically *emulated internal errors*
- **throw(Why)** - an exception is generated, and the caller might want to catch it.
- **exit(Why)** - used to terminate (kill) the (current) process.

a.k.a.
generated
errors

“Why” denotes a term to describe the relative reason.

Erlang provides two ways to catch exceptions:

- **Try ... catch** expressions
- **Catch** expressions

NOTE:
They are expressions, as it always happens in Erlang!

Motto: “Let It Crash!”

In designing a fault-tolerant system, we assume that errors will occur, processes will crash, machines will fail.

In Erlang, the focus is in the cure, not on prevention!

The behavior of a function should always be described in terms of *valid* input values; anomalous cases should automatically determine exception raising.

It is up to the caller handling the exception.

In applications with many processes, we can let a faulty process die, but *we should be able to correct the error in another process.*



Trapping Exceptions with try...catch

It is adopted a trapping mechanism alike the one present in other languages, but designed according to a “functional structure.”

Note: tail recursion is not admitted in the protected exception.

The “of” part is optional...

```
try ExprList
catch
  TypeOfError:ExcPattern -> Expr1
end.
```

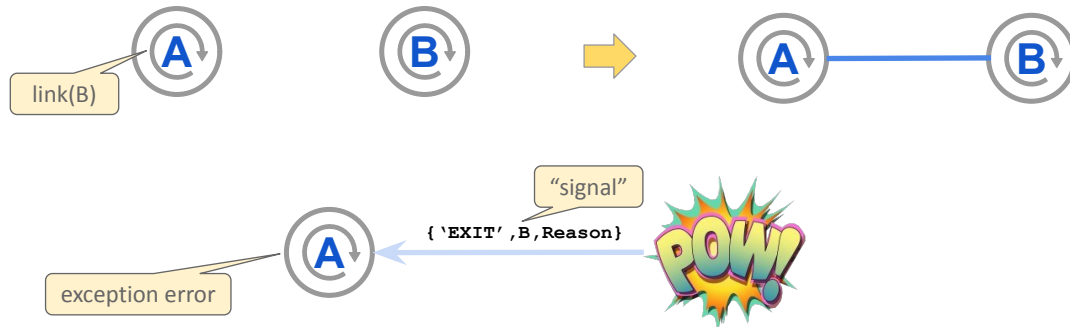
error|throw|exit

“Protected” expression

```
try Expr of
  OkPattern1 [Guards] -> Expr1;
  OkPattern2 [Guards] -> Expr2
catch
  TypeOfError:ExcPattern1 -> Expr3;
  TypeOfError:ExcPattern2 -> Expr4
after %like “finally” in other langs
  Expr5
end.
```


Links between Processes

A special relationship between processes can be set up, to let us terminate a set of related processes in case one of them would fail.
BIF: `link/1`, and `spawn_link/1-3`



© A. Bechini 2021

A. Bechini - UniPi

Trapping Signals

The exit signals seen so far make the recipient die.

How to trap them? Making the recipient a *system process*!

```
process_flag(trap_exit, true)
```

Any exit signal to a system process is converted to a regular message. Upon receiving it, we can decide what to do - maybe, restarting the dead process!

Special case: `kill` reason → never trapped; an ultimate weapon!

The recipient of `"kill"` will send out the `"killed"` reason instead: Why?

© A. Bechini 2021

A. Bechini - UniPi

Process Monitoring

A more flexible kind of relationship can be set up via `monitor/2-3`, and `spawn_monitor/1-3`.

Differently from links, monitors

```
monitor(process, Pid)
spawn_monitor(Pid2)
```

- are unidirectional;
- can be stacked (1+ instances per ordered pair).

As a monitored process goes down, the monitoring process receives a message whose pattern is `{'DOWN', MonitorRef, process, Pid, Reason}`.

Supervisors and Supervision Trees

Links & monitors are used to implement supervision of a process by another one.

Organizing an application according to a supervision tree let us get full control over it.

One specific OTP behavior is aimed at providing a robust/flexible implementation of a supervisor.

