Erlang
and Functional Programming

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

alessio.bechini@unipi.it

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

Functional Programming: Principles and Concurrency

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

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.

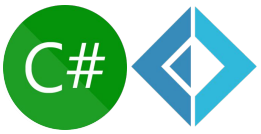
No side-effect in “ideal” computations: the only result is the computed value - even if, in practice, at least I/O is required!


```
   (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)
```

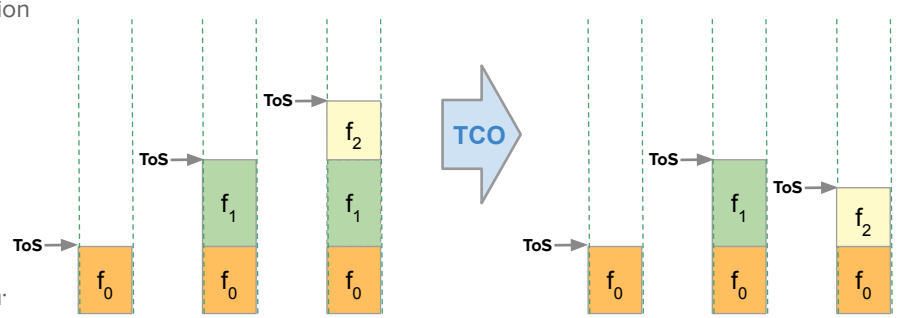
Tail recursion: for dealing with *performance problems* and *stack limits*.

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:
$\mathbf{f_1}$ is called inside $\mathbf{f_0}$;
$\mathbf{f_2}$ is called as last op in $\mathbf{f_1}$.



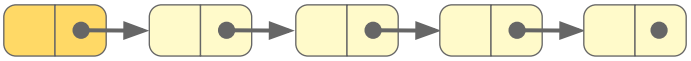
Use of Recursive Data Structures

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

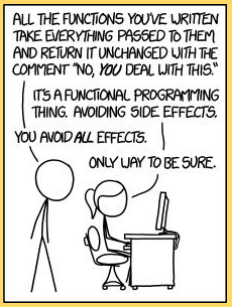
Basic data structure in most functional languages ➔ **single linked list**



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

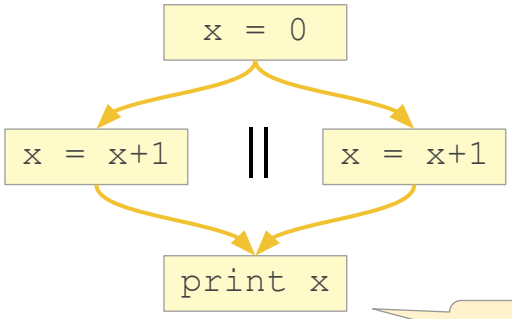
Pause for Thought

No Side-Effects: What Benefits?



Addressing Concurrency (I)

Main problem in concurrent computations: *data races*



1 ? or maybe 2 ?

The problem is related to *data mutability.*

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

Addressing Concurrency (II)

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

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

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*

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

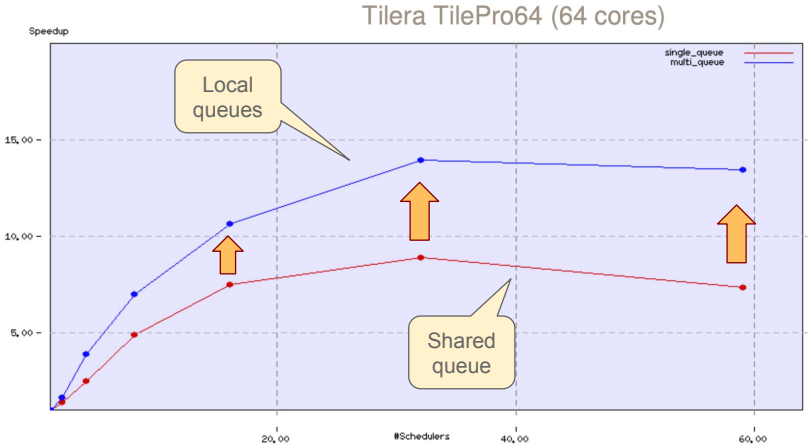
BEAM: one single OS process ➔ concurrency aspects managed **internally**. It uses its own 1+ schedulers (one per CPU core) to distribute the execution of Erlang processes over the "supporting" threads (one per core).

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.



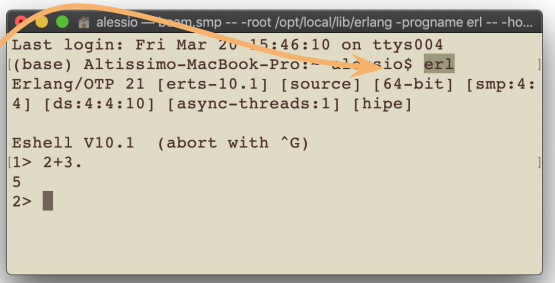
System Overview (II)

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.

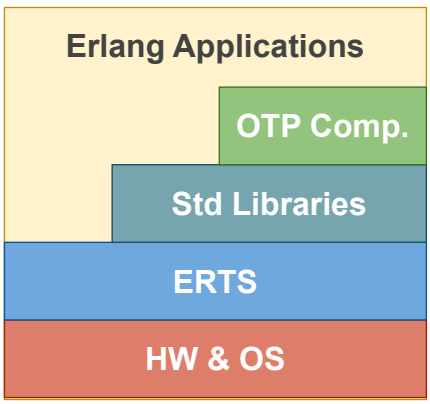
System Overview (III)

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.

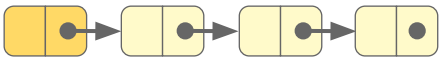
Delimiters: curly brackets - braces; elements separated by commas.

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: `[]`

"cons"

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

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**.

myfirstmod.erl

```
-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").
```

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

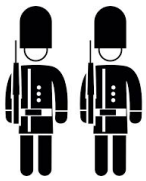
shapes.erl

```
-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 soos 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 ➔

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

Like "lambda" in Python and other languages

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

```
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}
```

Syntax to refer to values of NAMED functions

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


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>
```

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.

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

shell

Function values as params

```
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
```