

# Introduction to Erlang functions and modules

## Module 8 - Erlang tutorial 2

Function declaration

Module declaration

Compiling modules

Anonymous functions

Interrupting jobs

Function arity

Guards

Tail recursion

Case expressions

`io:{format/1,format/2}`

Recursive loops

List comprehensions

`lists:{seq,map,zip,filter,foldl}`

`random:uniform/2`

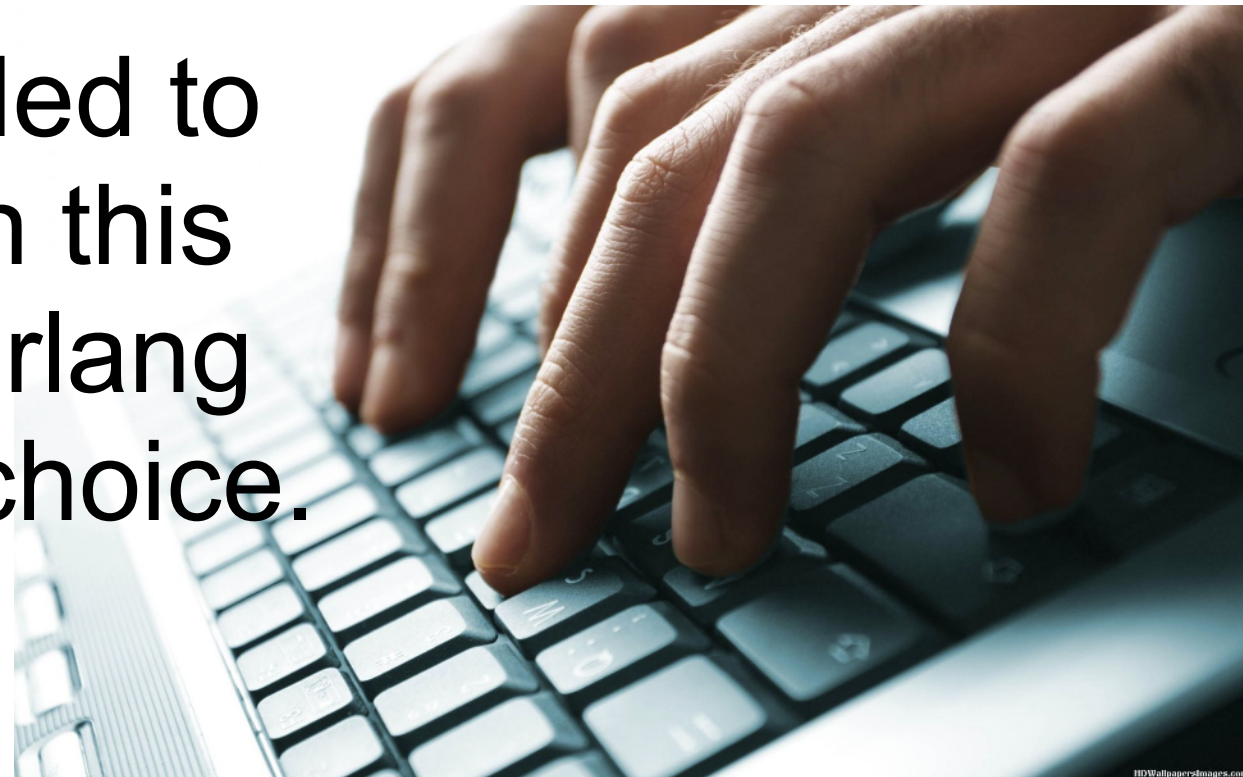
Operating systems and process oriented programming 2018

1DT096

# Try yourself

In this tutorial you will continue to use the Erlang shell. You will also need to write and save code in a text editor, for example Emacs.

You are strongly recommended to type in the examples given in this introduction yourself to the Erlang shell and your text editor of choice.



You may use SSH to log in to the department Linux system to start the Erlang shell and Emacs.

If you prefer, you can install Erlang on your private computer and then start the Erlang shell and use whatever text editor you prefer.

# **Function declaration**

# Function declaration

A function takes zero or more arguments and returns a value.

An example of a function declaration.

The function declaration must end with a period.

`double (X) -> 2*X.`

The **name** of the function, must be an atom.

A single **argument pattern** X.

A **right arrow**.

The **body** of the function, i.e, what the function does.

# Function declarations and the Erlang shell

Named functions can only be declared in **Erlang modules**.

```
1> double(X) -> 2*X.
```

```
* 1: syntax error before: '->'
```

When trying to declare a function in the Erlang shell we get an error message :- (



# **Function evaluation**

# Function evaluation

We have already seen how to declare a function.

```
double(X) -> 2*X.
```

Calling a function is an expression. When evaluating a call to a function, for example `double(7)` the argument `X` is bound to the value `7` and the value of the expression `2*X` is returned as the value of the expression `double(7)`.

```
1> double(7) .
```

```
14
```

```
2>
```

# Modules



So far, we have used the Erlang shell.

Erlang code can be saved in files called modules.

**Named functions**  
**must be declared in**  
**modules**



# Function **arity**

The arity of a function is an integer representing how many arguments can be passed to the function.

Erlang uses the notation **Function/Arity** where **Function** is the name of a function and **Arity** the arity of the function.

```
double(X)    -> 2*X.    %% double/1
```

```
sum(X,Y)     -> X+Y.    %% sum/2
```

```
sum(X,Y,Z)   -> X+Y+Z.  %% sum/3
```



aritet, ställighet

# Our first module

1

Lets create our first module in a file named `test.erl`.

```
%% The name of the module.
```

```
-module(test).
```

```
%% Only exported functions are
```

```
%% visible outside this module.
```

```
-export([double/1, sum/3]).
```

```
double(X)    -> 2*X.
```

```
sum(X,Y)     -> X + Y.
```

```
%% Inside the module, all functions can
```

```
%% be used. We can for example use
```

```
%% sum/2 when declaring sum/3.
```

```
sum(X,Y,Z)   -> sum(X,Y) + Z.
```

# Our first module

2

Lets create our first module in a file named `test.erl`.

`%% The name of the module.`

`-module(test).`

`%% Only exported functions are  
%% visible outside this module.`

`-export([double/1, sum/3]).`

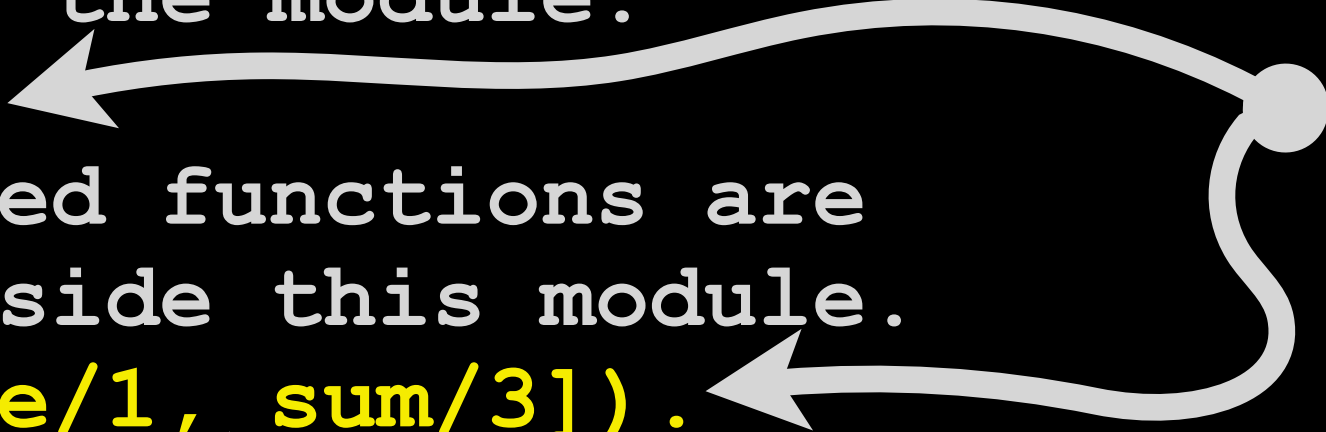
`double(X) -> 2*X.`

`sum(X,Y) -> X + Y.`

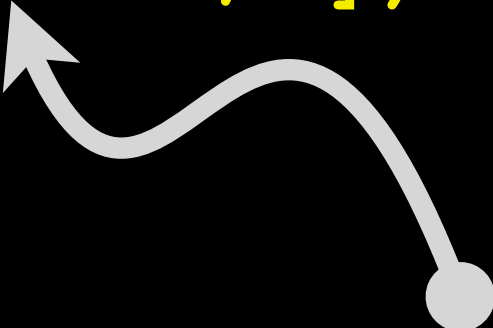
`%% Inside the module, all functions can  
%% be used. We can for example use  
%% sum/2 when declaring sum/3.`

`sum(X,Y,Z) -> sum(X,Y) + Z.`

*Note the  
ending  
periods.*



*A list of exported  
functions.*



# Our first module

3

Lets create our first module in a file named `test.erl`.

```
%% The name of the module.  
-module(test).  
%% Only exported functions are  
%% visible outside this module.  
-export([double/1, sum/3]).
```

```
double(X)    -> 2*X.
```

```
sum(X,Y)     -> X + Y.
```

```
%% Inside the module, all functions can  
%% be used. We can for example use  
%% sum/2 when declaring sum/3.
```

```
sum(X,Y,Z)   -> sum(X,Y) + Z.
```

**Make sure  
to save the  
file before  
you  
continue.**

# Compiling modules

To use functions declared in a module, you must first compile the module.

From the Erlang shell you compile a module by using the `c` function with the name of module as argument.

Make sure the module file is located in the same directory from where you started the Erlang shell.

```
6> c(test) .  
{ok, test}
```

Compiling the test module will return the tuple `{ok, test}` on success.

# Using functions from modules

To call a function in a module, the following syntax is used.

```
Module:Function(Arg1, Arg2, ..., ArgN)
```

, where **Module** is the name of the module,  
**Function** is the name of the function and  
**Arg1 ... ArgN** are the function arguments.

# Testing our module

We can now test our module from the Erlang shell.

```
6> c(test) .  
{ok, test}  
7> test:double(3) .  
6  
8> test:sum(3, 5) .  
** exception error: undefined  
function test:sum/2  
9> test:sum(3, 5, 11) .  
19
```

**Remember:** only exported function can be called. The function `test:sum/2` is not exported.



# The `lists` module

```
33> L = [3,9,2,5,0,3] .  
[3,9,2,5,0,3]  
34> lists:sort(L) .  
[0,2,3,3,5,9]  
35> lists:reverse(L) .  
[3,0,5,2,9,3]  
36> L .  
[3,9,2,5,0,3]  
37> lists:max(L) .  
9  
38> lists:min(L) .  
0  
39> lists:sum(L) .  
22
```

Erlang comes with a large number of modules. One such module is the `lists` module which provides many useful functions for dealing with lists.

**Remember:** Erlang uses single assignment. Calling a function cannot alter the argument.

# **Anonymous functions**

# Anonymous functions

It is possible to create a function without giving it a name using the following syntax.

```
F = fun (Arg1, Arg2, . . . ArgN) ->  
      . . .  
      end
```

This creates an anonymous function of N arguments and binds it to the variable F.

We can evaluate the fun F with the syntax.

```
F (Arg1, Arg2, . . . , ArgN)
```

# Anonymous functions and the shell

Anonymous functions can be defined and bound to variables from the shell using a **fun** expression. A **fun** expression begins with the keyword **fun** and ends with the keyword **end**.

```
3> Tripple = fun(X) -> 3*X end.
```

```
#Fun<erl_eval.6.82930912>
```

```
4> Tripple(5) .
```

```
15
```

```
5> Scripple = fun(X,Y) -> 3*X - 2*Y end.
```

```
#Fun<erl_eval.6.82930912>
```

```
6> Scripple(5,3) .
```

```
9
```

# **Function clauses**

# Function clauses

A function can be declared using multiple clauses.

A function declaration is a sequence of function clauses separated by **semicolons** (;), and terminated by a **period** (.).

```
Name (Pattern11, ..., Pattern1N)  ->  
    Body1;                          Clause # 1
```

```
...
```

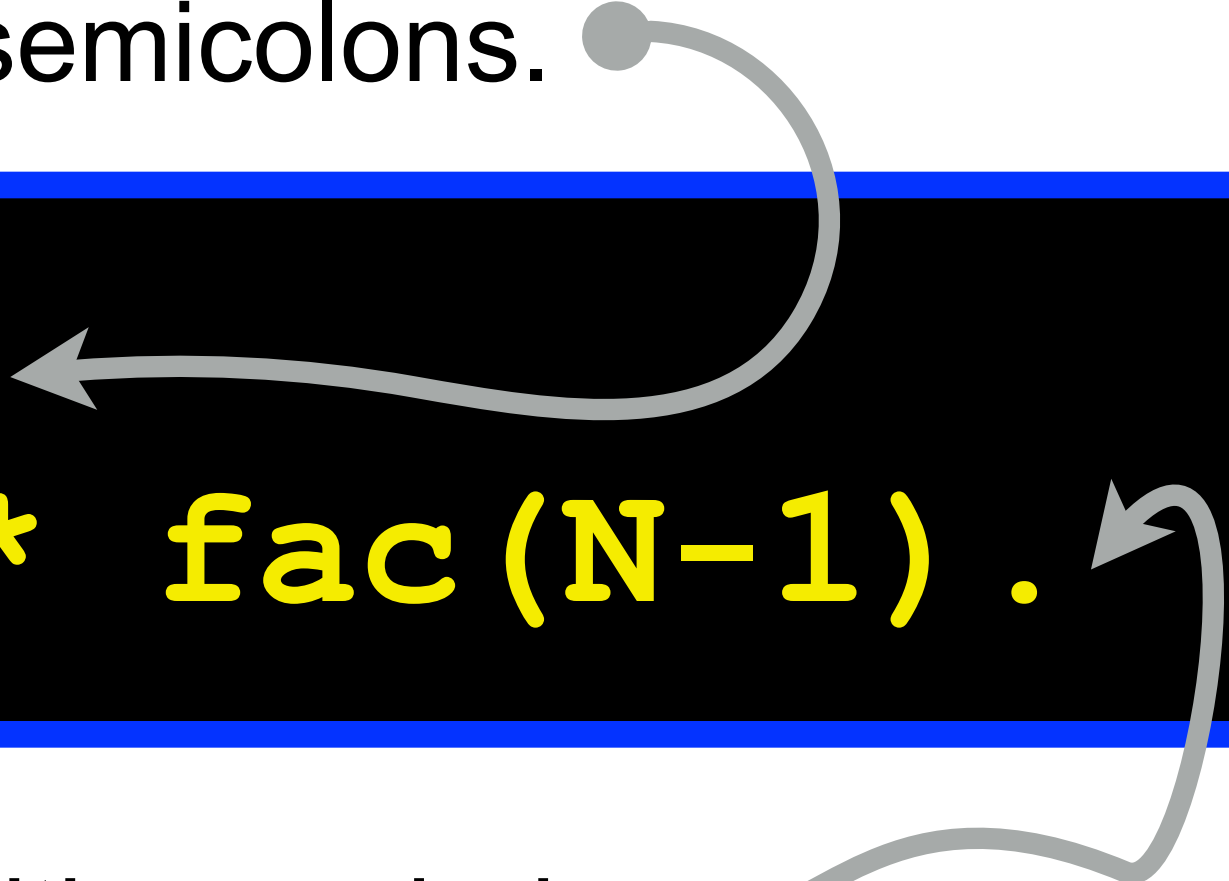
```
Name (PatternK1, ..., PatternKN)  ->  
    BodyK;                          Clause # K
```

# Multiple function clauses (first try)

Lets define a function to calculate the factorial of its argument.

Clauses are separated by semicolons.

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

A diagram with two grey curved arrows. The first arrow starts at the semicolon in the first clause of the code block and points to the text 'Clauses are separated by semicolons.'. The second arrow starts at the period at the end of the second clause and points to the text 'The last clause must end with a period.'.

The last clause must end with a period.

When calling a function, clauses are scanned sequentially until a clause is found that matches the provided argument.

```
fac(0) -> 1;           %% Clause 1
fac(N) -> N * fac(N-1) . %% Clause 2
```

Lets manually evaluate  $\text{fac}(3)$ .

$$\begin{aligned}\text{fac}(3) &= 3 * \text{fac}(3-1) \\ &= 3 * \text{fac}(2) \\ &= 3 * 2 * \text{fac}(2-1) \\ &= 3 * 2 * \text{fac}(1) \\ &= 3 * 2 * 1 * \text{fac}(1-1) \\ &= 3 * 2 * 1 * \text{fac}(0) \\ &= 3 * 2 * 1 * 1 \\ &= 6\end{aligned}$$



# Our second module (my.erl)

Save the following in a file named `my.erl`.

```
-module(my) .  
-export([fac/1]) .  
  
fac(0) -> 1;  
fac(N) -> N * fac(N-1) .
```

The `fac/1` function is a **recursive function**, a function calling itself.

# Compile and test (my.erl)

```
1> c(my) .
```

```
{ok,my}
```

```
2> my:fac(5) .
```

```
120
```

```
3> my:fac(99) .
```

```
93326215443944152681699238856266700490715968
```

```
26438162146859296389521759999322991560894146
```

```
39761565182862536979208272237582511852109168
```

```
6400000000000000000000000000000000000000
```

Erlang uses arbitrary-sized integers for integer arithmetic. In Erlang, integer arithmetic is exact, so you don't have to worry about arithmetic overflows or not being able to represent an integer in a certain word size.

# More testing ...

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

```
4> my:fac(-1) .
```

Calling `my:fac(-1)` will not terminate :- (

# Interrupt job

```
4> my:fac(-1) .
```

Press **Ctrl-g** then, **i** then **c** to kill the non terminating job to get back to the Erlang shell prompt.

```
User switch command
```

```
--> i
```

```
--> c
```

```
** exception exit: killed
```

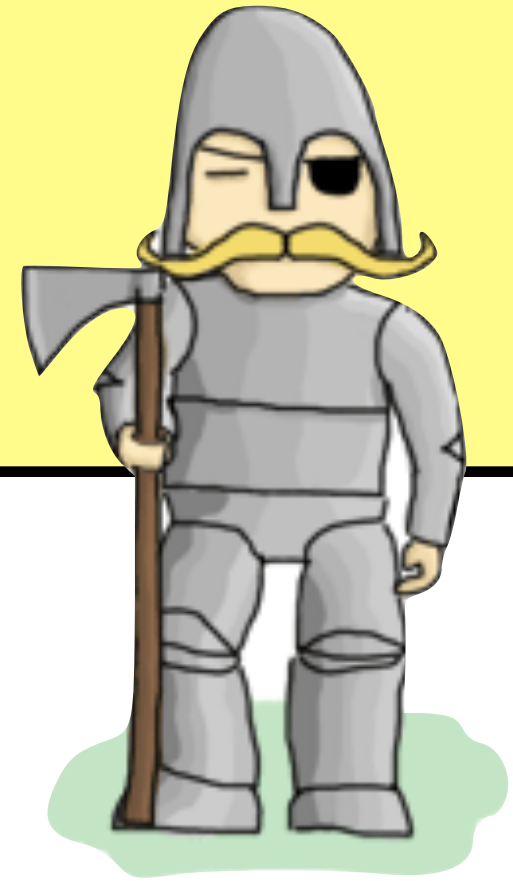
```
2>
```

# Guards

# Guards

Guards are boolean expressions that can be added to function clauses.

When calling a function, in addition for the arguments to match, the value of any guards must be true.



```
Name (Pattern11, ..., Pattern1N) [when GuardSeq1] ->  
    Body1;
```

```
...;
```

```
Name (PatternK1, ..., PatternKN) [when GuardSeqK] ->  
    BodyK.
```

# Add a guard to my:fac/1

```
-Module(my) .  
-export([fac/1]).
```

```
fac(0) ->  
    1;
```

```
fac(N) when N > 0 ->  
    N * fac(N-1) .
```



A guarded  
function  
clause.

# Compile and test (my.erl)

```
1> c(my) .  
{ok,my}  
2> my:fac(6) .  
720  
3> my:fac(5) .  
120  
4> my:fac(-1) .  
** exception error: no function clause  
matching my:fac(-1) (my.erl, line 4)
```

After adding the guard, calling `my:fac(-1)` (or with any other negative number) will give an error.



# **Tail recursion**

# Tail recursion

To write recursive functions efficiently in Erlang they must be tail recursive.

A function is tail recursive (roughly speaking) if for any recursive call the answer returned by the recursive call is the whole answer.

Can we write a tail recursive implementation of the factorial function?



# Tail recursive factorial

To use tail recursion we introduce a second helper function `fact/2`.

```
fact(N)      -> fact(N,1) .
```

```
fact(0,A)    -> A;
```

```
fact(N,A)    -> fact(N-1,N*A) .
```

**$\text{fact}(N) \rightarrow \text{fact}(N, 1).$**

**$\text{fact}(0, A) \rightarrow A;$**

**$\text{fact}(N, A) \rightarrow \text{fact}(N-1, N * A).$**

Lets manually evaluate  **$\text{fact}(3).$**

**$\begin{aligned} \text{fact}(3) &= \text{fact}(3, 1) \\ &= \text{fact}(2, 3 * 1) = \text{fact}(2, 3) \\ &= \text{fact}(1, 2 * 3) = \text{fact}(1, 6) \\ &= \text{fact}(0, 1 * 6) = \text{fact}(0, 6) \\ &= 6 \end{aligned}$**

## my:fact/2

Add the tail recursive solution to the module `my.erl`. Also add a base case for `my:fact(0)` and a guard to `my:fact/1`.

```
-module(my) .  
-export([fac/1, fact/1]).  
  
%% Non tail recursive.  
fac(0) -> 1;  
fac(N) when N > 0 -> N * fac(N-1).  
  
%% Tail recursive.  
fact(0) -> 1;  
fact(N) when N > 0 -> fact(N,1).  
  
fact(0,A) -> A;  
fact(N,A) -> fact(N-1,N*A).
```

# Compile and test (my.erl)

```
3> c(my) .
```

```
{ok,my}
```

```
4> my:fact(0) .
```

```
1
```

```
5> my:fact(5) .
```

```
120
```

```
6>
```

**Function**  
**clause bodies**

# Function clause bodies

A function clause body consists of a sequence of expressions separated by commas. The last expression must end with a period. When calling a function, the value of the last expression will be returned by the function.

```
pyth(A,B) ->
```

```
    A2 = A*A,
```

```
    B2 = B*B,
```

```
    C2 = A2 + B2,
```

```
    math:sqrt(C2) .
```

Here we use  
the **sqrt**  
function from  
the **math**  
module.



# **Case expressions**

# The **Case** expression

Using a case expression, decisions based on guarded patterns can be made.

```
case Expr of
  Pattern1 [when GuardSeq1] ->
    Body1;
  ...;
  PatternN [when GuardSeqN] ->
    BodyN
end
```

**NOTE:** *neither a semicolon  
nor a period after the last  
case bodyN.*

# A first try using Case

Add the following two function declarations to the module `my.erl` and don't forget to export them from the module.

```
guess1(42) ->  
    correct;  
guess1(_) ->  
    wrong.
```

```
guess2(N) ->  
    case N of  
        42 ->  
            correct;  
        _ ->  
            wrong  
    end.
```

```
guess1(42) ->
    correct;
guess1(_) ->
    wrong.
```

```
guess2(N) ->
    case N of
        42 ->
            correct;
        _ ->
            wrong
    end.
```

```
1> c(my) .
{ok,my}
2> my:guess1(13) .
wrong
3> my:guess1(42) .
correct
4> my:guess2(42) .
correct
5> my:guess2(77) .
wrong
```

The two functions  
**guess1/1** and **guess2/1**  
are equivalent.

What do you think the following function does?

```
what(X) ->
  case X of
    {A, B} when A+B > 100 ->
      {huge_tuple, A, B};
    {A, B} ->
      {tuple, A, B};
    [H|T] ->
      {list, H, T};
    X ->
      {unknown, X}
  end.
```

Add the above `what/1` function to `my.erl`, don't forget to add `what/1` to the list of exported functions.

Try it out.

```
6> c(my) .
{ok, my}
7> my:what(99) .
{unknown, 99}
8> my:what({a, b}) .
{tuple, a, b}
9> my:what({99, bar}) .
{tuple, 99, bar}
10> my:what({99, 2}) .
{huge_tuple, 99, 2}
11> 9> my:what({a, b, c}) .
{unknown, {a, b, c}}
12> my:what([1, 2, 3]) .
{list, 1, [2, 3]}
13> my:what("abc") .
{list, 97, "bc"}
```

```
what(X) ->
  case X of
    {A, B} when A+B > 100 ->
      {huge_tuple, A, B};
    {A, B} ->
      {tuple, A, B};
    [H|T] ->
      {list, H, T};
    X ->
      {unknown, X}
  end.
```

**io::format**

## `io:format/1`

This function prints a string to the terminal.

```
1> io:format("Hello world!").  
Hello world!ok
```

The string "Hello world!" was printed.

All expressions must have a value and the value of `io:format` is `ok` which also got printed on the same line.



# io:format/1

~n

To include a new line we use ~n.

```
1> io:format("Hello world!").
```

```
Hello world!ok
```

```
2> io:format("Hello world!~n").
```

```
Hello world!
```

```
ok
```

In the second example, the string "Hello world!" was printed, then a new line and finally the result ok.

# io:format/2

~S, ~W

This version of `format` takes a second argument which must be a list. Elements of this list will be included in the printed string using `~S` for strings and `~W` for numbers.

```
3> X = "Dolly".
```

```
"Dolly"
```

*Replace ~s with the value of the string X.*

```
4> io:format("Hello ~s!", [X]).
```

```
Hello Dolly!ok
```

```
5> Y = 42.
```

```
42
```

*Replace ~w with the value of the number Y.*

```
6> io:format("Hello ~w!", [Y]).
```

```
Hello 42!ok
```

*Replace ~s with the value of X and ~w with the value of Y.*

```
6> io:format("Hello ~s ~w!", [X, Y]).
```

```
Hello Dolly 42!ok
```

*A list!*



**Pattern  
matching  
twice**

# Pattern matching **twice** in a function clause head



```
-module(test).  
  
-compile(export_all).  
  
list_info([]) ->  
    io:format("Empty list [].~n");  
list_info([H|T] = L) ->  
    io:format("List ~w with head ~w and tail ~w.~n",[L,H,T]).
```

```
1> c(test).  
{ok,test}  
2> test:list_info([]).  
Empty list [].  
ok  
3> test:list_info([1,2,3,4,5]).  
List [1,2,3,4,5] with head 1 and tail [2,3,4,5].  
ok
```

**Recursion  
instead of  
loops**

# Recursion instead of loops

In C-like languages such as C, C++ and Java we often use for-loops to iterate.

```
for (i=0; i <= 10; i++) {  
    do_something();  
}
```

In Erlang we use functions and recursive function calls.

```
repeat(0) -> ok; %% We choose to return  
               %% the atom ok when done.  
repeat(N) ->  
    do_something(),  
    my_repeat(N-1).
```

The function `repeat/1` calls itself and counts down from `N` to `0`.

# my:repeat/1

Add the function `repeat/1` to the module `my`. For each recursive call `io:format/2` prints the value of the argument `N`.

```
-module(my) .  
  
-export([fac/1, fact/1, repeat/1]).  
  
%% Non tail recursive.  
fac(0) -> 1;  
fac(N) when N > 0 -> N * fac(N-1).  
  
%% Tail recursive.  
fact(0) -> 1;  
fact(N) when N > 0 -> fact(N,1).  
  
fact(0,A) -> A;  
fact(N,A) -> fact(N-1,N*A).  
  
repeat(0) -> ok;  
repeat(N) ->  
    io:format("N = ~w~n", [N]),  
    repeat(N-1).
```

# Compile and test (my:repeat/1)

```
7> c(my) .
```

```
{ok, my}
```

```
8> my:repeat(5) .
```

```
N = 5
```

```
N = 4
```

```
N = 3
```

```
N = 2
```

```
N = 1
```

```
ok
```

```
9>
```

```
repeat(0) -> ok;
```

```
repeat(N) ->
```

```
    io:format("N = ~w~n", [N]),
```

```
    repeat(N-1) .
```



**Generating  
sequential lists**

**lists : seq/2**

# lists:seq/2

Generating sequential lists.

```
1> L = lists:seq(1,10) .  
[1,2,3,4,5,6,7,8,9,10]  
2> lists:seq($a, $z) .  
"abcdefghijklmnopqrstuvwxyz"  
3> lists:seq(1,10,2) .  
[1,3,5,7,9]  
4> lists:seq(1,10,3) .  
[1,4,7,10]  
5> lists:seq(10,1,-1) .  
[10,9,8,7,6,5,4,3,2,1]
```

**List**

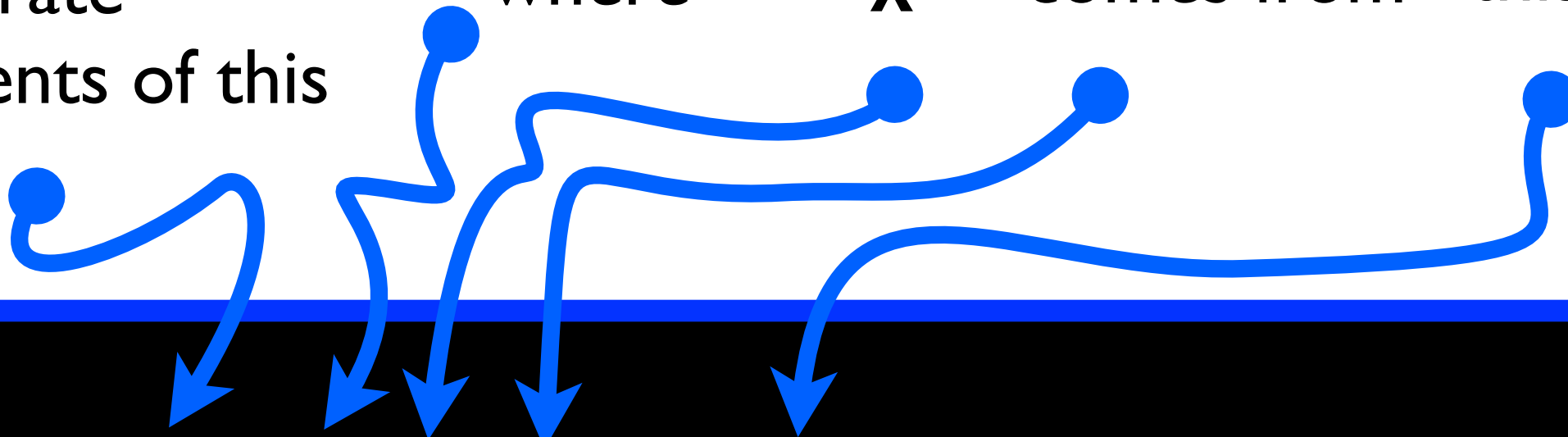
**comprehensions**

# List comprehensions

1

A list comprehension is a mathematical way to construct a list.

Generate elements of this form      where  $x$  comes from this list.



```
2> [2*x || x <- lists:seq(1,10)].  
[2,4,6,8,10,12,14,16,18,20]
```

# List comprehensions

2

One or more conditions can be added to a list comprehension to filter out elements from the source list.

A boolean expression

```
2> [2*X || X <- lists:seq(1,10), X rem 2 == 0].  
[4,8,12,16,20]
```

A comma

# List comprehensions as loops

```
1> L = lists:seq(1,10) .  
[1,2,3,4,5,6,7,8,9,10]  
2> P = fun(X) -> io:format("~w ", [X]) end.  
#Fun<erl_eval.6.82930912>  
3> [P(X) || X <- L].  
1 2 3 4 5 6 7 8 9 10  
[ok,ok,ok,ok,ok,ok,ok,ok,ok,ok]
```

# Transforming lists

`lists : map/2`

## lists:map/2

1

Using `lists:map/2` a list can easily be transformed into a new list by applying a function to every element in the list.

To double every element in a list we can map an anonymous function doing the doubling over the list.

```
1> L = lists:seq(1,10) .  
[1,2,3,4,5,6,7,8,9,10]  
2> lists:map(fun(X) -> 2*X end, L) .  
[2,4,6,8,10,12,14,16,18,20]
```

**Note:** the following list comprehension gives the same result.

```
3> [2*X || X <- L] .  
[2,4,6,8,10,12,14,16,18,20]
```



## lists:map/2

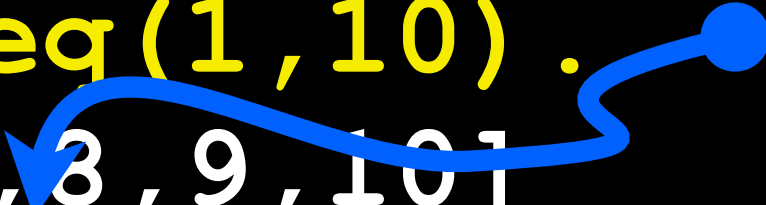
2

We don't need to use anonymous functions.

We can map a function from a module over a list.

```
1> L = lists:seq(1,10) .  
[1,2,3,4,5,6,7,8,9,10]  
2> lists:map(fun my:fact/1, L) .  
[1,2,6,24,120,720,5040,40320,362880,  
3628800]
```

Note: must add the keyword *fun*.



**Note:** the following list comprehension gives the same result.

```
3> [my:fact(X) || X <- L] .  
[1,2,6,24,120,720,5040,40320,362880,  
3628800]
```

# Combining lists

`lists : zip/2`

## lists:zip/2

"Zips" two lists of equal length into one list of two-tuples, where the first element of each tuple is taken from the first list and the second element is taken from corresponding element in the second list.

```
4> lists:zip([1,2,3], [a,b,c])  
[{1,a},{2,b},{3,c}]
```

# Filtering lists

`lists:filter/2`

## lists:filter/2

Checks every element in a list using a function **P** returning a bool ( a predicate function). Creates a new list with all elements for which **P** returns **true**.

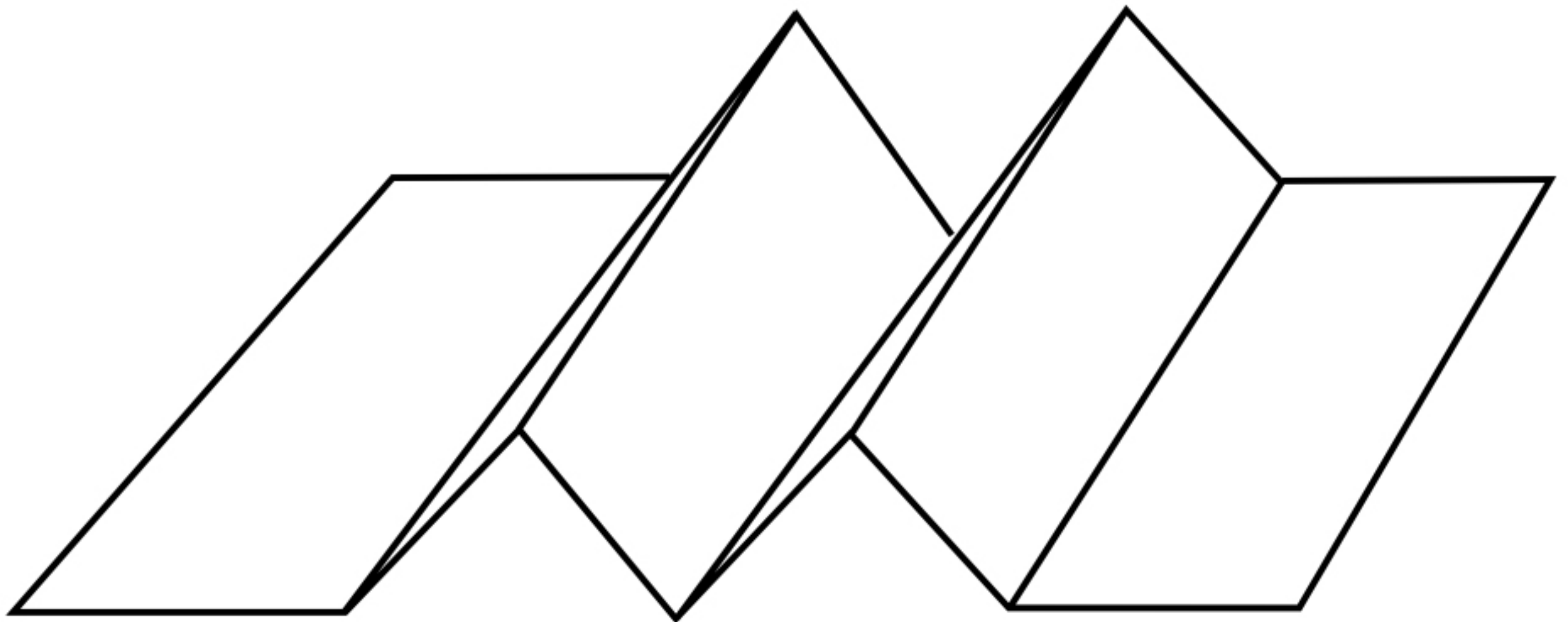
```
1> L = lists:seq(1,10) .  
[1,2,3,4,5,6,7,8,9,10]  
2> P = fun(X) -> X rem 2 == 0 end.  
#Fun<erl_eval.6.82930912>  
3> lists:filter(P, L) .  
[2,4,6,8,10]  
3> [X || X <- L, X rem 2 == 0 ] .  
[2,4,6,8,10]
```

**Combining all  
elements in a list  
to single value**

**lists:fold1/3**

# Fold

A fold is a higher-order function that analyze a recursive data structure and combines the elements of the data structure to construct a new data structure or value.



# lists:foldl/3

## definition

Look at every element of a list one after the other and reduce them to a single value.

**foldl(Fun, Acc0, List) -> Acc1**

Parameter	Type	Description
Fun	<code>fun((Elem :: T, AccIn) -&gt; AccOut)</code>	Combining function
Acc0	<code>Acc1 = AccIn = AccOut = term()</code>	Initial accumulator value
List	<code>[T]</code>	Input list
T	<code>term()</code>	Element of the input list



## lists:foldl/3

## examples

Use a fold to sum all elements of a list.

```
1> lists:foldl(fun(X, Sum) -> X + Sum end,  
               0,  
               [1,2,3,4,5]).
```

15

Use a fold to multiply all elements of a list.

```
2> lists:foldl(  
    fun(X, Prod) -> X * Prod end,  
    1,  
    [1,2,3,4,5]).
```

120

**Functional  
problem  
solving**

# Functional problem solving

Given a problem, try to decompose the original problem into smaller problems which are easier to solve or even already have a solution.

**Problem:** shuffle the elements of a list randomly.

```
1> L = lists:seq(1,10) .  
[1,2,3,4,5,6,7,8,9,10]  
2> L1 = [{random:uniform(100), X} || X <- L] .  
[{45,1},{73,2},{95,3},{51,4},{32,5},{60,6},  
{92,7},{67,8},{48,9},{60,10}]  
3> L2 = lists:sort(L1) .  
[{32,5},{45,1},{48,9},{51,4},{60,6},{60,10},  
{67,8},{73,2},{92,7},{95,3}]  
4> L3 = lists:map(fun({_,X}) -> X end, L2) .  
[5,1,9,4,6,10,8,2,7,3]
```

# my:shuffle/1

Add the following function to the module `my`.

```
shuffle(L) ->  
  L1 = [{random:uniform(100),X} || X <- L],  
  L2 = lists:sort(L1),  
  lists:map(fun({_ ,X}) -> X end, L2).
```

```
20> c(my) .
```

```
{ok,my}
```

```
21> L = lists:seq(1,10) .
```

```
[1,2,3,4,5,6,7,8,9,10]
```

```
22> my:shuffle(L) .
```

```
[4,9,10,5,2,6,8,7,1,3]
```

```
23> my:shuffle(L) .
```

```
[2,8,6,7,1,9,5,4,3,10]
```

```
24> my:shuffle(L) .
```

```
[10,9,1,5,6,8,7,4,3,2]
```