

# Functional Programming and Erlang

**Lecture 6** of TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg



UNIVERSITY OF  
GOTHENBURG

---

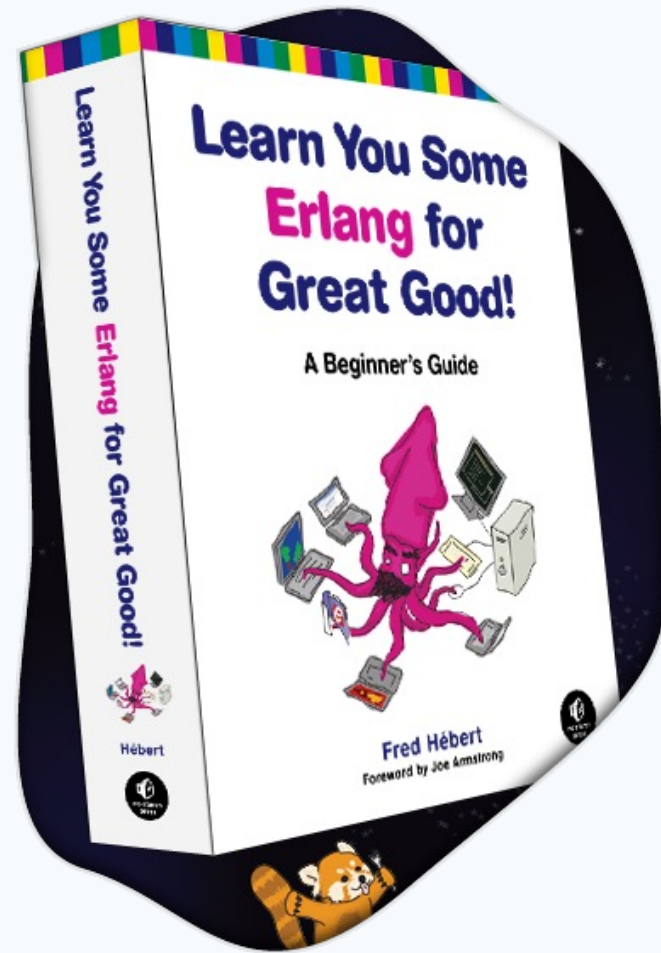


**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

# Today's menu

- What is Erlang?
- Types
- Expressions and patterns
- Function definitions
- Recursion
- Impure and higher-order functions

Don't forget



(<http://learnyousomeerlang.com/>)

# What is Erlang?

# What is Erlang?

Erlang combines a functional language with message-passing features:

- The functional part is sequential, and is used to define the behavior of processes
- The message-passing part is highly concurrent: it implements the actor model, where actors are Erlang processes

This lecture covers the functional/sequential part of Erlang

# Erlang: A minimal history

- 1973** *Hewitt* and others develop the **actor model** – a formal model of concurrent computation
- 1985** *Agha* further refines the actor model
- Mid 1980s** *Armstrong* and others at Ericsson prototype the first version of Erlang (based on the actor model)
- Late 1980s** Erlang's implementation becomes efficient; Erlang code is used in production at Ericsson
- 1998** Ericsson bans Erlang, which becomes open-source
- Late 2000s** Erlang and the actor model make a come-back in mainstream programming



# Erlang in the real world

Erlang has made a significant **impact** in the **practice** of concurrent programming by making the formal actor model applicable to real-world scenarios

- Initially, Erlang was mainly used for **telecommunication software**:
  - Ericsson's AXD301 switch – includes over one million lines of Erlang code; achieves “nine 9s” availability (99.9999999%)
  - Cellular communication infrastructure (services such as SMSs)
- Recently, it has been rediscovered for Internet **communication apps**:
  - WhatsApp's communication services are written in Erlang
  - Facebook Chat (in the past)

# Why Erlang?

*We've faced many challenges in meeting the ever-growing demand for [the **WhatsApp**] messaging services, but [...] **Erlang** continues to prove its capability as a versatile, reliable, high-performance platform.*

*Rick Reed, 2014 - [That's 'Billion' with a 'B': Scaling to the next level at WhatsApp](#)*

*The language itself has many pros and cons, but we chose **Erlang** to power [**Facebook**] **Chat** because its model lends itself well to concurrent, distributed, and robust programming.*

*Chris Piro, 2010 – [Chat Stability and Scalability](#)*



# What is a functional language?

**Functional** languages are based on elements quite **different from** those **imperative** languages are based on

**Imperative languages** (such as Java)  
are based on:

- state – variables
- state modifications - assignments
- iteration – loops

**Functional languages** (such as Erlang)  
are based on:

- data – values
- functions on data – without side effects
- functional forms – function composition, higher-order functions

# What is a functional language?

**Functional** languages are based on elements quite **different from** those **imperative** languages are based on

**Imperative languages** (such as Java)  
are based on:

An imperative program is a sequence of state modifications on variables

```
// compute  $x^n$ 
int power(int x, int n) {
    int result = 1;
    for (int i = n; i < n; i++)
        result *= x;
    return result;
}
```

**Functional languages** (such as Erlang)  
are based on:

A functional program is the side-effect-free application of functions on values

```
% compute  $X^N$ 
power(X, 0) -> 1;
power(X, N) -> X * power(X, N-1)
```

In functional programs, variables store **immutable** values, which can be **copied** but not modified

# The Erlang shell

You can experiment with Erlang using its [shell](#), which can evaluate expressions on the fly without need to define complete programs

```
$ erl
Erlang R16B03 (erts-5.10.4) [source] [64-bit] [smp:2:2]

Eshell V5.10.4 (abort with ^G)

1> 1 + 2.           % evaluate expression `1 + 2'
3
2> c(power).       % compile file `power.erl'
{ok,power}
3> power:power(2, 3). % evaluate power(2, 3)
8
```

- Notice you have to terminate all expressions with a period
- Functions are normally defined in external files, and then used in the shell
- Compilation targets bytecode by default

# Types

# Types, dynamically

A **type** constrains:

1. The (kinds) of **values** that an expression can take
2. The **functions** that can be applied to expressions of that type

For example, the **integer** type:

1. includes integer values (1, -100, 234), but not, say, decimal numbers (10.3, -4.3311) or strings ("hello!", "why not")
2. supports functions such as sum +, but not, say, logical **and**

- Erlang is **dynamically typed**:
  - programs do **not** use **type declarations**
  - the type of an expression is only determined **at runtime**
    - when the expression is evaluated
  - if there is a type mismatch (for example `3+false`) expression evaluation **fails**
- Erlang types include **primitive** and **compound** data types

# An overview of Erlang types

Erlang offers eight **primitive types**:

- **Integers**: arbitrary-size integers with the usual operations
- **Atoms**: roughly corresponding to identifiers
- **Floats**: 64-bit floating point numbers
- **References**: globally unique symbols
- **Binaries**: sequences of bytes
- **Pids**: process identifiers
- **Ports**: for communication
- **Funs**: function closures

And three + two **compound types**  
(a.k.a. **type constructors**):

- **Tuples**: fixed-size containers
- **Lists**: dynamically-sized containers
- **Maps**: key-value associative tables (a.k.a. dictionaries) –recent feature, experimental in Erlang/OTP R17
- **Strings**: syntactic sugar for sequences of characters
- **Records**: syntactic sugar to access tuple elements by name

# Numbers

**Numeric** types include **integers** and **floats**

- We will mainly use *integers*, which are arbitrary-size, and thus do not overflow

EXPRESSION	VALUE	
3	3	explicit constant (“ <b>term</b> ”)
1 + 3	4	addition
1 - 3	-2	subtraction
4 * 2	8	multiplication
5 <b>div</b> 4	1	integer division
5 <b>rem</b> 3	2	integer remainder
5 / 4	1.25	float division
<b>power</b> (10, 1000)	1000000000...	no overflow!
2#101	5	101 in base 2
16#A1	161	<b>A1</b> in base 16

# Atoms

**Atoms** are used to denote **distinguished values**

(they are similar to symbolic uninterpreted constants)

An atom can be:

- A sequence of alphanumeric characters and underscores, starting with a lowercase letter
- An arbitrary sequence of characters (including spaces and escape sequences) between single quotes
  - An atom is to be enclosed in single quotes (') if it does not begin with a lower-case letter or if it contains other characters than alphanumeric characters, underscore (\_), or @

Examples of valid atoms:

```
x  
a_Longer_Atom  
'Uppercase_Ok_in_quotes'  
'This is crazy!'  
true
```



# Booleans

In Erlang there is **no Boolean type**

Instead, the **atoms** `true` and `false` are conventionally used to represent Boolean values

OPERATOR	MEANING
<b>not</b>	negation
<b>and</b>	conjunction (evaluates both arguments/eager)
<b>or</b>	disjunction (evaluates both arguments/eager)
<b>xor</b>	exclusive or (evaluates both arguments/eager)
<b>andalso</b>	conjunction (short-circuited/lazy)
<b>orelse</b>	disjunction (short-circuited/lazy)

Examples:

```
true or      (10 + false)  % error: type mismatch in second argument
true orelse (10 + false)  % true: only evaluates first argument
```

# Relational operators

Erlang's **relational operators** have a few syntactic differences with those of most other programming languages

OPERATOR	MEANING
<	less than
>	greater than
=<	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to
==	numeric equal to
/=	numeric not equal to

Examples:

```
3 == 3      % true: same value, same type
3 == 3.0    % false: same value, different type
3 == 3.0    % true: same value, type not checked
```

# Order between different types

Erlang defines an **order relationship** between values of **any type**

When different types are compared, the following **order** applies:

*number < atom < reference < fun < port < pid < tuple < map < list*

Thus, the following inequalities hold:

```
3 < true           % number < atom
3 < false          % number < atom
999999999 < infinity % number < atom
1000000000000000 < epsilon % number < atom
```

When comparing **tuples to tuples**:

- comparison is by size first
- two tuples with the same size or two lists are compared element by element, and satisfy the comparison only if all (existing) pairs satisfy it

# Tuples

**Tuples** denote **ordered sequences** with a **fixed** (but arbitrary for each tuple instance) **number of elements** (They are written as comma-separated sequences enclosed in **curly braces**)

Examples of valid tuples:

```
{ }           % empty tuple
{ 10, 12, 98 }
{ 8.88, false, aToM } % elements may have different types
{ 10, { -1, true } } % tuples can be nested
```

Functions on a tuple T:

FUNCTION	RETURNED VALUE
<code>element(N, T)</code>	Nth element of T
<code>setelement(N, T, X)</code>	a copy of T, with the Nth element replaced by X
<code>tuple_size(T)</code>	number of elements in T

Examples:

```
element(2, {a, b, c}) % b: tuples are numbered from 1
setelement(1, {a, b}, z) % {z, b}
tuple_size({ }) % 0
```

# Lists

**Lists** denote **ordered sequences** with a **variable** (but immutable for any list instance) **number of elements** (They are written as comma-separated sequences enclosed in **square brackets**)

Examples of valid lists:

```
[ ]           % empty list
[ 10, 12, 98 ]
[ 8.88, false, {1, 2} ] % elements may have different type
[ 10, [ -1, true ] ]   % lists can be nested
```

# List operators

Some useful functions on lists  $L$ :

FUNCTION	RETURNED VALUE
<code>length(L)</code>	number of elements in $L$
<code>[H   L]</code>	a copy of $L$ with $H$ added as first (“head”) element
<code>hd(L)</code>	$L$ ’s first element (the “head”)
<code>tl(L)</code>	a copy of $L$ without the first element (the “tail”)
<code>L1 ++ L2</code>	the concatenation of lists $L1$ and $L2$
<code>L1 -- L2</code>	a copy of $L1$ with all elements in $L2$ removed (without repetitions, and in the order they appear in $L1$ )

Operator `|` is also called `cons`; using it, we can define any list:

```
[1, 2, 3, 4] ::= [1 | [2 | [3 | [4 | []]]]]
hd([H | T]) ::= H
tl([H | T]) ::= T
  % this is an example of --
[1, 2, 3, 4, 2] -- [1, 5, 2] ::= [3, 4, 2]
```

# Strings

**Strings** are sequences of characters enclosed between **double quotation marks**

- Strings are just *syntactic sugar* for lists of character codes

String **concatenation** is implicit whenever multiple strings are juxtaposed without any operators in the middle

Using strings ( $\$c$  denotes the integer code of character  $c$ ):

```
" "           % empty string ::= empty list
"hello!"
"hello" "world" % ::= "helloworld"
"xyz" ::= [$x, $y, $z] ::= [120, 121, 122] % true
[97, 98, 99]   % evaluates to "abc"!
```

# Records

**Records** are **ordered sequences** with a **fixed number of elements**, where each element has an atom as **name**

- Records are just *syntactic sugar* for **tuples** where positions are named

```
% define `person` record type
% with two fields: `name` with default value "add name"
%                   `age` without default value (undefined)
-record(person, { name="add name", age })
% `person` record value with given name and age
#person{name="Joe", age=55}
#person{age=35, name="Jane"} % fields can be given in any order
% when a field is not initialized, the default applies
#person{age=22} ::= #person{name="add name", age=22}
% evaluates to `age` of `Student` (of record type `person`)
Student#person.age
```

- Erlang's shell does not know about records, which can only be used in **modules**
  - In the shell: `#person{age=7, name="x"}` **is** `{person, "x", 7}`.



# Expressions and patterns

# Variables

**Variables** are identifiers that can be **bound to values**

(they are similar to constants in an imperative programming language)

A variable **name** is a sequence of alphanumeric characters, underscores, and @, **starting** with an **uppercase letter** or an **underscore**

In the **shell**, you can directly bind values to variable:

- Evaluating  $\text{var} = \text{expr}$  binds the value of expression  $\text{expr}$  to variable  $\text{var}$ , and returns such value as value of the whole **binding expression**
- Each variable can only be bound **once**
- To clear the binding of variable  $\text{var}$  evaluate  $\text{f}(\text{var})$
- Evaluating  $\text{f}()$  clears all variable bindings
- The anonymous variable `_` (“any”) is used like a variable whose value can be ignored

In **modules**, variables are used with **pattern matching** (which we present later)

# Expressions and evaluation

- **Expressions** are evaluated exhaustively to a **value** – sometimes called (ground) term: a number, an atom, a list, ...

The **order of evaluation** is given by the usual **precedence rules**

(using **parentheses** forces the evaluation order to be inside-out of the nesting structure)


Some precedence rules to be aware of:

- **and** has higher precedence than **or**
- **andalso** has higher precedence than **orelse**
- when lazy (**andalso**, **orelse**) and eager (**and**, **or**) Boolean operators are mixed, they all have the same precedence and are left-associative
- ++ and -- are right-associative (concatenation and subtraction in lists)
- relational operators have lower precedence than Boolean operators; thus you have to use parentheses in expressions such as  $(3 > 0)$  **and**  $(2 == 2.0)$

# Precedence rules: Examples

```

3 + 2 * 4           % is 11
3 + (2 * 4)        % is 11
(3 + 2) * 4        % is 20
true or false and false % is true
true orelse false andalso false % is true
true or false andalso false % is false
true orelse false and false % is true (why?)
  
```



After evaluating the first "true"  
 there is no need to evaluate the rest

# Patterns

**Pattern matching** is a flexible and concise mechanism to **bind values to variables**

It is widely used in functional programming languages to define functions on data (especially lists); Erlang is no exception

A **pattern** has the same structure as a term, but in a pattern some parts of the term **may** be replaced by free **variables**

Examples of patterns:

```
3  
A  
{X, Y}  
{X, 3}  
[H | T]  
[H | [2]]
```

- Note that a pattern may contain bound variables
  - in this case, evaluating the pattern implicitly evaluates its bound variables

# Pattern matching

**Pattern matching** is the process that, given a pattern  $P$  and a term  $T$ , **binds the variables** in  $P$  to match the values in  $T$  according to  $P$  and  $T$ 's structure

If  $P$ 's structure (or type) cannot match  $T$ 's, pattern matching **fails**

PATTERN = TERM	BINDINGS
$3 = 3$	none
$A = 3$	$A: 3$
$A = B$	if $B$ is bound then $A ::= B$ ; otherwise fail
$\{X, Y\} = 3$	fail (structure mismatch)
$\{X, Y\} = \{1, 2\}$	$X: 1, Y: 2$
$\{X, Y\} = \{"a", [2, 3]\}$	$X: "a", Y: [2, 3]$
$[H T] = [1, 2]$	$H: 1, T: [2]$
$[H [2]] = [1, 2]$	$H: 1$
$[F, S] = [\text{foo}, \text{bar}]$	$F: \text{foo}, S: \text{bar}$
$\{X, Y\} = [1, 2]$	fail (type mismatch)

# Pattern matching: Notation

Given a **pattern**  $P$  and a **term**  $T$ , we write  $\langle P \triangleq T \rangle$  to denote the **pattern match of  $T$  to  $P$**

- If the match is successful, it determines bindings of the variables in  $P$  to terms
- Given an expression  $E$ , we write  $E\langle P \triangleq T \rangle$  to denote the term obtained by applying the bindings of the pattern match  $\langle P \triangleq T \rangle$  to the variables in  $E$  with the same names
- If the pattern match fails,  $E\langle P \triangleq T \rangle$  is undefined

## Examples:

- $(X + Y)\langle\{X, Y\} \triangleq \{3, 2\}\rangle$  is 5
- $(T ++ [2])\langle[H|T] \triangleq [8]\rangle$  is  $[2]$
- $H\langle[H|T] \triangleq [ ]\rangle$  is undefined

NOTE: The notation  $E\langle P \triangleq T \rangle$  is **not** valid Erlang, but we use it to illustrate Erlang's semantics

# Multiple expressions

**Multiple expressions**  $E_1, \dots, E_n$  can be combined in a **compound expression** obtained by separating them using commas

- **Evaluating** the compound expression entails evaluating all component expressions in the order they appear, and returning **the value** of the **last** component expression as the value of the whole compound expression
- A single failing evaluation makes the whole compound expression evaluation **fail**

Examples:

```
3 < 0, 2.           % evaluates 3 < 0
                    % returns 2

3 + true, 2.        % evaluates 3 + true
                    % fails

R=10, Pi=3.14, 2*Pi*R. % binds 10 to R,
                    % binds 3.14 to Pi
                    % returns 62.8..
```



# Multiple expression blocks

Using **blocks** delimited by **begin... end**, we can introduce **multiple** expressions where **commas** would normally be interpreted in a different way

This may be useful in function calls:

```
power(2, begin X=3, 4*X end) % returns power(2, 12)
```

Without **begin...end**, the expression would be interpreted as calling a function `power` with three arguments

# List comprehensions

List comprehensions provide a convenient syntax to **define lists** using pattern matching

It is an **expression** of the form: `[ Expression || P1 <- L1, ..., Pm <- Ln, C1, ..., Cn ]` where:

- each  $P_k$  is a pattern
  - each  $L_k$  is a list expression
  - each  $C_k$  is a condition (a Boolean expression)
- Intuitively, each pattern  $P_k$  is matched to every element of  $L_k$ , thus determining a binding  $B$ 
    - if substituting all bound values makes all conditions evaluate to true, the value obtained by substituting all bound values in `Expression` is accumulated in the list result;
    - otherwise the binding is ignored

Examples:

```
[X*X || X <- [1, 2, 3, 4]]           % is [1, 4, 9, 16]
[X   || X <- [1, -3, 10], X > 0]    % is [1, 10]
[{A, B} || A <- [carl, sven], B <- [carlsson, svensson]]
% is [{carl, carlsson}, {carl, svensson},
%     {sven, carlsson}, {sven, svensson}]
```

# Modules

A **module** is a **collection of function definitions** grouped in a file

Modules are the only places where functions can be defined – they cannot directly be defined in the shell

The **main elements** of a module are as follows:

```
-module(foo).    % module with name `foo' in file `foo.erl'
-export([double/1,up_to_5/0]). % exported functions
    % each f/n refers to the function with name `f' and arity `n'
-import(lists, [seq/2]). % functions imported from module `lists'
    % function definitions:
double(X) -> 2*X.
up_to_5() -> seq(1, 5).    % uses imported lists:seq
```

**Compiling** and using a module in the **shell**:

```
1> c(foo).          % compile module `foo' in current directory
{ok,foo}.          % compilation successful
2> foo:up_to_5().  % call `up_to_5' in module `foo'
[1,2,3,4,5]
```

# Function definitions

# Function definitions: basics

In Erlang (and all functional prog. lang.) **functions** are the fundamental units of computation

- A **function** defines how to map values to other values
  - Unlike in imperative programming languages, most functions in Erlang have **no side effects**: they do not change the state of the program executing them (especially their arguments)

The basic definition of an  $n$ -argument function  $f$  (arity  $n$ ), denoted by  $f/n$ , has the form:

$$\begin{array}{ccc} \text{Head} & & \text{Body} \\ \underbrace{\hspace{1.5cm}} & & \underbrace{\hspace{1.5cm}} \\ f(P_1, \dots, P_n) & \rightarrow & E. \end{array}$$

- The function **name**  $f$  is an atom
- The function's formal **arguments**  $P_1, \dots, P_n$  are patterns
- The **body**  $E$  is an expression – normally including variables that appear in the arguments

Examples:     `identity(X) -> X.            % the identity function`  
                  `sum(X, Y)     -> X + Y.        % the sum function`

# Examples of function definitions

The basic definition of an  $n$ -argument function  $f$  (arity  $n$ ), denoted by  $f/n$ , has the form:

$$f(P_1, \dots, P_n) \rightarrow E.$$

More examples:

```
zero()      -> 0.           % integer zero
identity(X) -> X.          % identity
sum(X, Y)   -> X + Y.      % sum
head([H|_]) -> H.          % head
tail([_|T]) -> T.          % tail
second({_, Y}) -> Y.       % 2nd of pair
positives(L) -> [X || X <- L, X > 0]. % filter positive
```

# Function call/evaluation

Given the definition of a function  $f/n$ :

$$f(P_1, \dots, P_n) \rightarrow E.$$

a **call expression** to  $f/n$  has the form:

$$f(A_1, \dots, A_n)$$

and is **evaluated** as follows:

1. For each  $1 \leq k \leq n$ , evaluate  $A_k$ , which gives a term  $T_k$
2. For each  $1 \leq k \leq n$ , pattern match  $T_k$  to  $P_k$
3. If all pattern matches are successful, the call expression evaluates to  $E(P_1, \dots, P_n \triangleq T_1, \dots, T_n)$
4. Otherwise, the evaluation of the call expression fails

# Examples of function calls

DEFINITIONS	CALLS	VALUE
<code>zero()</code> -> 0.	<code>zero()</code>	0
<code>identity(X)</code> -> X.	<code>identity({1,2,3})</code>	{1,2,3}
<code>sum(X, Y)</code> -> X + Y.	<code>sum(zero(), second({2,3}))</code>	3
<code>head([H _])</code> -> H.	<code>head([])</code>	fail
<code>head([H _])</code> -> H.	<code>head([3,4,5])</code>	3
<code>tail([_ T])</code> -> T.	<code>tail([])</code>	fail
<code>positives(L)</code> -> [X    X <- L, X > 0].	<code>positives([-2,3,-1,6,0])</code>	[3,6]



# Function definition: clauses

Function definitions can include multiple **clauses**, separated by semicolons:

```
f(P11, ..., P1n) -> E1;
f(P21, ..., P2n) -> E2;
.
.
.
f(Pm1, ..., Pmn) -> Em.
```

A **call expression** is evaluated against each clause in textual order; the first successful *match* is returned as the result of the call

Therefore, we should enumerate clauses from more to less specific

```
lazy_or(true, _) -> true;
lazy_or(_, true) -> true;
lazy_or(_, _) -> false.
```

← This function does not work as expected unless this clause is listed last

# Pattern matching with records

**Pattern matching** an expression  $R$  of **record type**  $rec$

$$\#rec\{f_1=P_1, \dots, f_n=P_n\} = R$$

succeeds if, for all  $1 \leq k \leq n$ , field  $f_k$  in  $R$ 's evaluation (i.e.,  $R\#name.f_k$ ) matches to pattern  $P_k$

If record type  $rec$  has fields **other** than  $f_1, \dots, f_n$ , they are **ignored** in the match

Thanks to this behavior, using **arguments of record type** provides a simple way to **extend data** definitions **without** having to **change** the signature of all functions that use that datatype

# Flexible arguments with records: Example

```
-record(error, {code}).  
error_message(#error{code=100}) -> io.format("Wrong address");  
error_message(#error{code=101}) -> io.format("Invalid username");  
...  
error_message(_) -> io.format("Unknown error").
```

If we want to add more information to the type `error`, we only have to change the record definition, and the clauses using the new information:

```
-record(error, {code, line_number}).  
error_message(#error{code=100}) -> io.format("Wrong address");  
error_message(#error{code=101}) -> io.format("Invalid username");  
...  
error_message(#error{code=C, line_number=L}) -> io.format("Unknown error p", [C, L]).
```

Compare this to the case where we would have had to change `error_message` from a unary to a binary function!

# Function definition: guards

Clauses in function definitions can include any number of **guards** (also called **conditions**):

$$f(Pk1, \dots, Pkn) \textbf{ when } Ck1, Ck2, \dots \textbf{ -> } Ek;$$

A guarded clause is selected only if **all guards**  $Ck1, Ck2, \dots$  evaluate to **true** under the match, that is if  $Cki(Pk1, \dots, Pkn \triangleq Tk1, \dots, Tkn)$  evaluates to true for all guards  $Cki$  in the clause

More generally, two guards can be separated by either a comma or a semicolon: **commas** behave like lazy **and** (both guards have to hold); **semicolon** behave like lazy **or** (at least one guard has to hold)

```
can_drive(Name, Age) when Age >= 18 -> Name ++ " can drive";  
can_drive(Name, _) -> Name ++ " cannot drive".
```

```
same_sign(X, Y) when X > 0, Y > 0; X < 0, Y < 0 -> true;  
same_sign(_, _) -> false.
```

## Type checking -- at runtime

Since Erlang is dynamically typed, there are cases where we have to **test** the **actual type** of an expression

- For example, because a certain operation is only applicable to values of a certain type

To this end, Erlang provides several **test functions** whose names are self-explanatory:

```
is_atom/1  
is_boolean/1  
is_float/1  
is_integer/1  
is_list/1  
is_number/1  
is_pid/1  
is_port/1  
is_tuple/1
```

Use these only when necessary: in most cases defining implicitly partial functions is enough

## Function definition: local binding

The expression **body** in a function definition can include **compound** expressions with **bindings**:

$$f(Pk1, \dots, Pkn) \rightarrow V1=E1, \dots, Vw=EW, Ek;$$

Such bindings are **only visible** within the function definition

They are useful to define shorthands in the definition of complex expressions

```
volume({cylinder, Radius, Height}) ->  
  Pi=3.1415,  
  BaseArea=Pi*Radius*Radius,  
  Volume=BaseArea*Height,  
  Volume.
```

# If expressions (guard patterns)

**ifs** provide a way to express conditions alternative to guards (in fact, *ifs* are called – somewhat confusingly – *guard patterns* in Erlang)

An **if** expression:

```
if
    C1 -> E1;
    ⋮
    Cm -> Em
end
```

evaluates to the expression  $E_k$  of the first guard  $c_k$  in **textual order** that evaluates to true; if no guard evaluates to true, evaluating the `if` expression fails

```
age(Age) ->
  if Age > 21 -> adult;
     Age > 11 -> adolescent;
     Age > 2  -> child;
     true    -> baby end.
```

# Case expressions

**Cases** provide an additional way to use pattern matching to define expressions. A **case expression**:

```
case E of
  P1 -> E1;
  ⋮
  Pm -> Em
end
```

evaluates to  $E_k(P_k \triangleq T)$ , where  $E$  evaluates to  $T$ , and  $P_k$  is the first pattern in **textual order** that  $T$  matches to; if  $T$  matches no pattern, evaluating the case expression fails

Patterns may include `when` clauses, with the same meaning as in function definitions

```
years(X) ->
  case X of {human, Age} -> Age;
            {dog, Age}   -> 7*Age;
            _             -> cant_say
  end.
```



# Which one should I use?

Having several **different ways** of defining a function can be confusing. There are no absolute rules, but here are some **guidelines** that help you write idiomatic code:

- the **first** option to try is using **pattern matching** directly in a function's arguments, using different clauses for different cases
- if parts of a pattern expression depend on others, you may consider using **case expressions** to have nested patterns
- you do not need **if expressions** very often (but it's good to know what they mean, and sometimes they may be appropriate)

# Recursion

# Recursion in programming

- Recursion is a style of programming where functions are defined in terms of themselves

The **definition** of a function  $f$  is **recursive** if it includes a call to  $f$  (directly or indirectly)

```
% compute  $X^n$   
power(X, 0) -> 1;  
power(X, N) -> X * power(X, N-1).
```

↑  
Recursive call

# Recursion in mathematics

**Recursion** is a **style of programming** where functions are defined in terms of themselves

The **definition** of a function  $f$  is **recursive** if it includes a call to  $f$  (directly or indirectly)

Definition of **natural numbers**:

- $0$  is a natural number;
- if  $n$  is a natural number then  $n + 1$  is a natural number.



Recursive/inductive definition

# Recursion: from math to programming

Recursion in programming provides a natural way of implementing recursive definitions in mathematics

**Factorial** of a nonnegative integer  $n$ :

$$n! = n \cdot \underbrace{(n-1) \dots 1}_{n \text{ terms}} = n \cdot \underbrace{(n-1) \dots 1}_{n-1 \text{ terms}}$$

$$n! = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 & \leftarrow \text{Base case} \\ n \cdot (n-1)! & \text{if } n > 1 & \leftarrow \text{Recursive/inductive case} \end{cases}$$

# Recursion: from math to programming

Recursion in programming provides a natural way of implementing recursive definitions in mathematics

**Factorial** of a nonnegative integer  $n$ :

$$n! = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 & \leftarrow \text{Base case} \\ n \cdot (n-1)! & \text{if } n > 1 & \leftarrow \text{Recursive/inductive case} \end{cases}$$

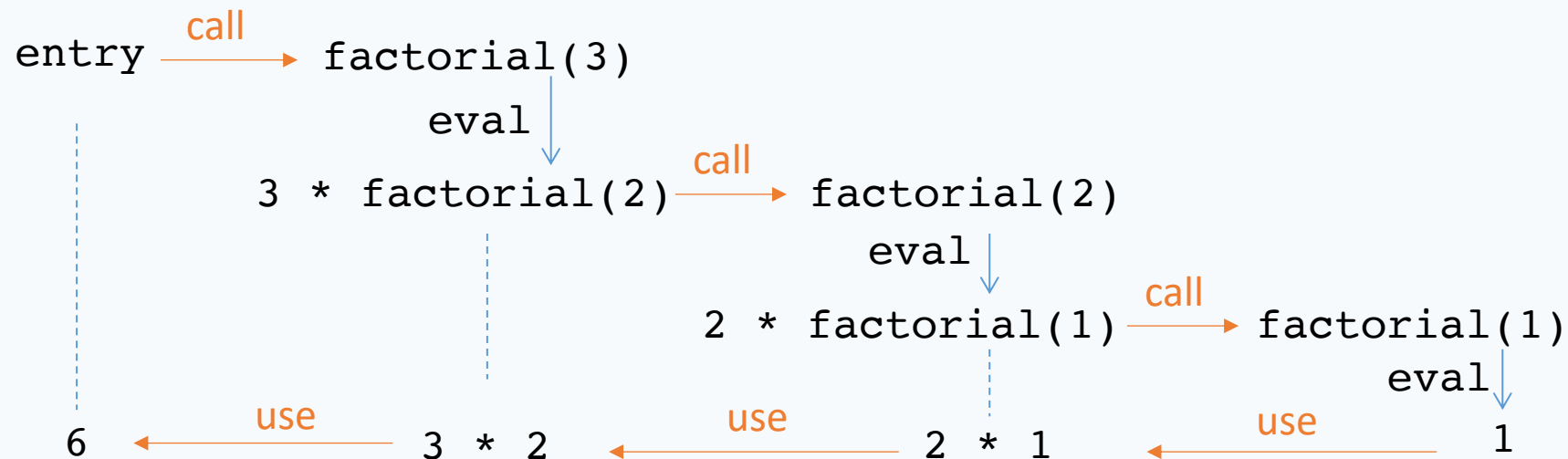
```
factorial(N) when N =< 1 -> 1;           % base case
factorial(N)           -> N *factorial(N-1). % recursive case
```

↑  
Recursive call

# How does recursion work?

Each recursive call triggers an **independent evaluation** of the recursive function  
 (Independent means that it works on its own private copy of actual argument expressions)

When a recursive instance terminates evaluation, its value is used in the calling instance **for its own evaluation**



# Recursion as a design technique

**Recursion** as a programming technique is useful to design programs using the **divide and conquer** approach:

To solve a **problem instance**  $P$ , **split**  $P$  into problem instances  $P_1, \dots, P_n$  chosen such that:

1. Solving  $P_1, \dots, P_n$  is **simpler** than solving  $P$  directly
2. The solution to  $P$  is a **simple combination** of the solutions to  $P_1, \dots, P_n$

In functional programming, **recursion** goes hand in hand with **pattern matching**:

- **Pattern matching** splits a function argument's into **smaller bits** according to the input's structure
- **Recursive** function definitions define the **base cases directly**, and **combine** simpler cases into more complex ones



# Recursive functions: Sum of list

Define a function `sum(L)` that returns the **sum of all numbers** in `L`

1. The base case (the simplest possible) is when `L` is empty: `sum([ ]) -> 0`
2. Let now `L` be non-empty: a non empty list matches the pattern `[H|T]`
  - `H` is a single number, which we must add to the result
  - `T` is a list, which we can sum by calling `sum` recursively

```
sum([ ])      -> 0;           % base case
sum([H|T])   -> H + sum(T). % recursive case
```

Can we switch the order of clauses?

In this case, YES

To make the function more robust, we can skip over all non-numeric elements:

```
sum([ ])      -> 0;           % base case
sum([H|T]) when is_number(H) -> H + sum(T); % recursive case 1
sum([_|T])    -> sum(T).      % recursive case 2
```

# Recursive functions: Last list element

Define a function `last(L)` that returns the **last element** of `L`

1. When `L` is empty, `last` is undefined, so we can ignore this case
2. The simplest case is then when `L` is one element: `last([E]) -> E`
3. Let now `L` be non-empty: a non empty list matches the pattern `[H|T]`
  - `E` is the first element, which we throw away
  - `T` is a list, whose last element we get by calling `last` recursively

Can `T` match the empty list?

**No**, because neither of the clauses match the empty list

```
last([E]) -> E;           % base case
last([_|T]) -> last(T). % recursive case
```

To make this explicit, we could write:

```
last([E|[]]) -> E;       % base case
last([_|T]) -> last(T). % recursive case
```

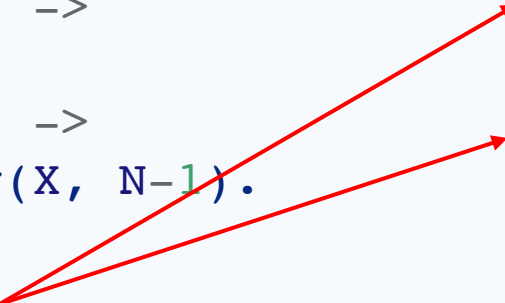
# Tail recursion

A **recursive function**  $f$  is **tail recursive** if the evaluation of  $f$ 's body evaluates the recursive call **last**

```

% general recursive:
power(_, 0) ->
    1;
power(X, N) ->
    X * power(X, N-1).

% tail recursive:
power(X, N) ->
    power(X, N, 1).
power(_, 0, Accumulator) ->
    Accumulator;
power(X, N, Accumulator) ->
    power(X, N-1, X*Accumulator).
  
```



**Overloading:**

**two functions `power/2` and `power/3`**

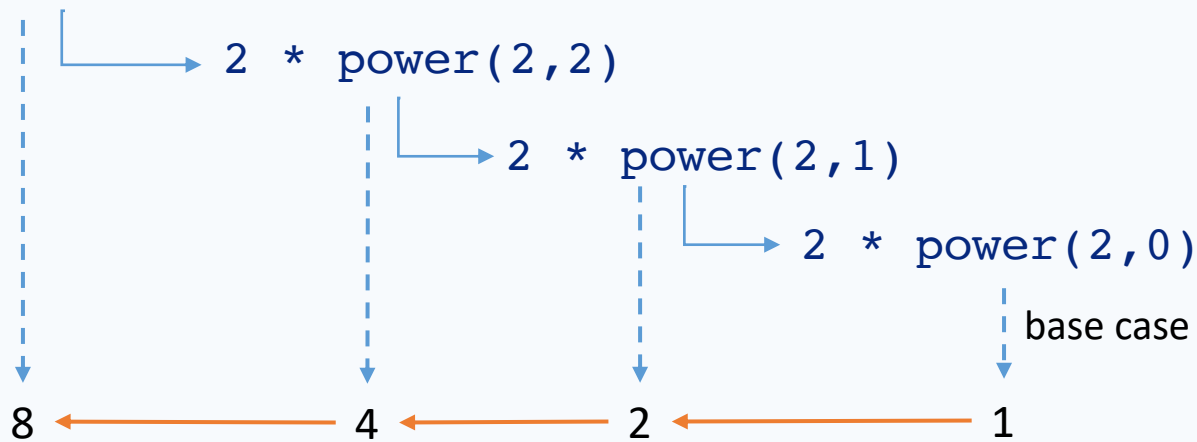
- Tail-recursive functions are generally more efficient than general-recursive functions
- When efficiency is **not an issue**, there is no need to use a tail-recursive style; but we will use tail-recursive functions extensively (and naturally) when implementing servers

# General Recursion vs Tail Recursion

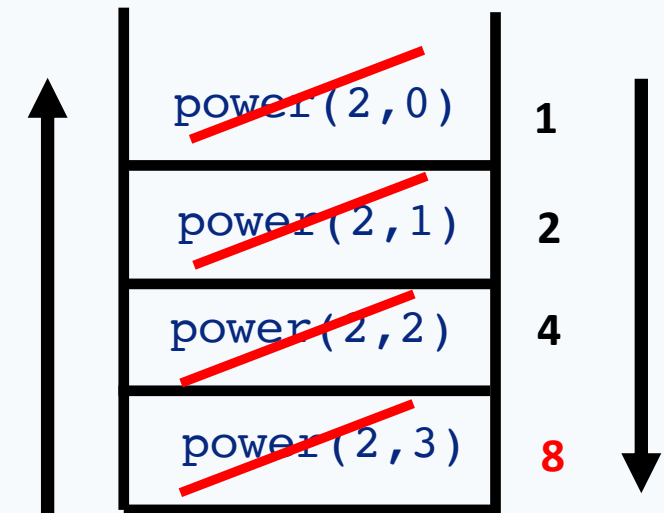
## General recursion:

```
% general recursive:
power(_, 0) -> 1;
power(X, N) -> X * power(X, N-1).
```

power(2,3) = ??



Stack



# General Recursion vs Tail Recursion

## Tail recursion:

```
% tail recursive:
```

```
power(X, N) -> power(X, N, 1).
```

```
power(_, 0, Accumulator) -> Accumulator;
```

```
power(X, N, Accumulator) -> power(X, N-1, X*Accumulator).
```

```
power(2, 3) = ??
```

```
└─> power(2, 3, 1)
```

```
└─> power(2, 2, 2*1)
```

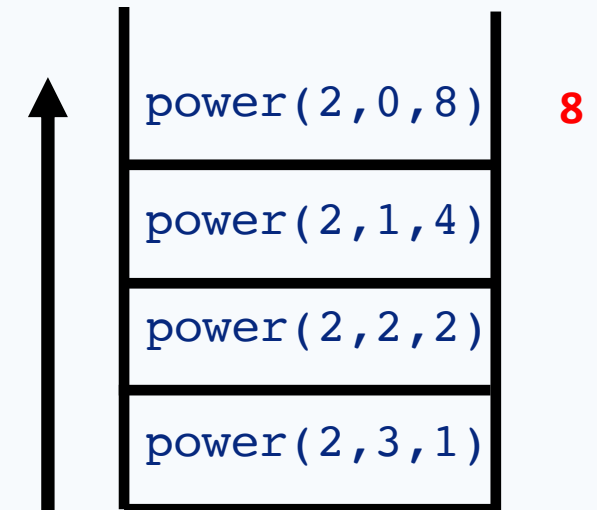
```
└─> power(2, 1, 2*2)
```

```
└─> power(2, 0, 2*4)
```

```
└─ base case
```

```
8
```

Stack



# Impure and higher-order functions

# Where are all the statements, assignments, loops?

Statements, assignments, and loops are not available as such in Erlang

Everything is an **expression** that gets **evaluated**:

- (Side-effect free) expressions are used instead of statements
- (Pure) functions return **modified copies** of their arguments instead of modifying the arguments themselves
- One-time bindings are used instead of assignments that change values to variables
- Recursion is used instead of loops

The sparse presence of side effects helps make functional programs **higher level** than imperative ones

# Printing to screen

The expressions we have used so far have no **side effects**, that is they do not change the state but simply evaluate to a value

- Not all expressions are side-effect free in Erlang
  - **Input/output** is an obvious exception: to print something to screen, we **evaluate** an expression call, whose side effect is printing

`io:format(Format, Data)` % *print the string Format, interpreting control sequences on Data*

CONTROL SEQUENCE	DATA
~B	integer
~g	float
~s	string
~p	any Erlang term
~n	line break

You can use `fwrite`  
instead of `format`

```
1> io:format("~s ~B. ~p~n~s ~B~n", ["line", 1, true, "line", 2]).
line 1. true
line 2
```



# Exception handling

Erlang has an **exception handling mechanism** that is similar to a functional version of Java's `try/catch/finally` blocks:

```
try Expr of
    Success1 -> Expr1;
    ...
catch
    Error1:Fail1 -> Recov1;
    ...
after After end
```

- The `try` blocks behaves like a case block
- If evaluating `Expr` raises an exception, it gets pattern matched against the clauses in `catch` (`Errork`'s are error types, `Failk`'s are patterns, and `Recovk`'s are expressions)
- Expression `After` in the `after` clause always gets evaluated in the end (but does not return any value: used to close resources)

# Exception handling: Example

Function `safe_plus` tries to evaluate the sum of its arguments:

- if evaluation succeeds, it returns the result
- if evaluation raises a `badarith` exception, it returns `false`

```
safe_plus(X, Y) ->  
  try X + Y of  
    N -> N  
  catch  
    error:badarith -> false  
  end.
```

Example of using it:

```
1> safe_plus(2, 3).  
5
```

```
2> safe_plus(2, []).  
false
```

# Functions are values too

**Functions** are **first-class** objects in Erlang: they can be passed around like any other values, and they can be **arguments** of functions

- A function  $f/k$  defined in module  $m$  is passed as argument `fun m:f/k`

This makes it easy to define functions that apply other functions to values following a pattern

```
% apply function F to all elements in list L  
map(F, []) -> [];  
map(F, [H|T]) -> [F(H) | map(F,T)].
```

```
1> map(fun m:age/1, [12, 1, 30, 56]).  
[adolescent, baby, adult, adult]
```

```
age(Age) ->  
  if Age > 21 -> adult;  
      Age > 11 -> adolescent;  
      Age > 2 -> child;  
      true -> baby end.
```

A function that takes another function as argument is called **higher-order**

# High-Order Functions

```
% apply function F to all elements in list L  
map(F, []) -> [];  
map(F, [H|T]) -> [F(H)|map(F,T)].
```

Let's define a function:

```
doub(X) -> 2*X;
```

What is the result of calling `map (doub/1, [12,1,30,56])` ?

```
map (doub/1, [12,1,30,56]) = ??
```

```
↳ [doub(12) | map(doub, [1,30,56])]
```

```
↳ [doub(12) | [doub(1) | map(doub, [30,56])]]
```

```
↳ [doub(12) | [doub(1) | [doub(30) | map(doub, [56])]]]
```

```
↳ [doub(12) | [doub(1) | [doub(30) | [doub(56) | map(doub, [])]]]]
```

⏟  
[]

⏟  
[112]

⏟  
[60,112]

⏟  
[2,60,112]

⏟  
[24,2,60,112]

# Inline functions

Sometimes it is necessary to **define** a function **directly in an expression** where it is used

For this we can use **anonymous functions** – also called lambdas, closures, or funcs (the last is Erlang jargon):

```
fun
    (A1) -> E1;
    ⋮
    (An) -> En
end
```

where each  $A_k$  is a sequence of patterns, and each  $E_k$  is a body

```
% double every number in the list
1> map(fun (X)->2*X end, [12, 1, 30, 56]).
[24, 2, 50, 112]
```

# Working on lists

Module `lists` includes many useful predefined functions to work on lists

These are some you should know about – but check out the full module documentation at <http://erlang.org/doc/man/lists.html>:

```
all(Pred, List)      % do all elements E of List satisfy Pred(E)?
any(Pred, List)     % does any element E of List satisfy Pred(E)?
filter(Pred, List)  % all elements E of List that satisfy Pred(E)
last(List)          % last element of List
map(Fun, List)      % apply Fun to all elements of List
member(Elem, List)  % is Elem an element of List?
reverse(List)       % List in reverse order
seq(From, To)       % list [From, From+1, ..., To]
seq(From, To, I)    % list [From, From+I, ...,
```

# Folds

Several functions compute their result by recursively accumulating values from a list:

```
sum([ ])      -> 0;          len([ ])      -> 0;
sum([H|T])    -> H + sum(T). len([H|T])    -> 1 + len(T).
```

We can generalize this pattern into a single higher-order function `fold(F, R, L)`: starting from an **initial value** `R`, combine all elements of **list** `L` using **function** `F` and accumulate the result:

```
fold(_, Result, [ ])      -> Result;
fold(F, Result, [H|T])    -> F(H, fold(F, Result, T)).
```

Using `fold`, we can define `sum` and `len`:

```
sum(L) ->          len(L) ->
  fold(fun (X,Y)->X+Y end, 0, L).      fold(fun (X,Y)->1+Y end, 0, L).
```

Erlang module `lists` offers functions `foldr/3` (which behaves like our `fold`) and `foldl/3` (a tail-recursive version of `fold`, with the same arguments)

# Folds: Example

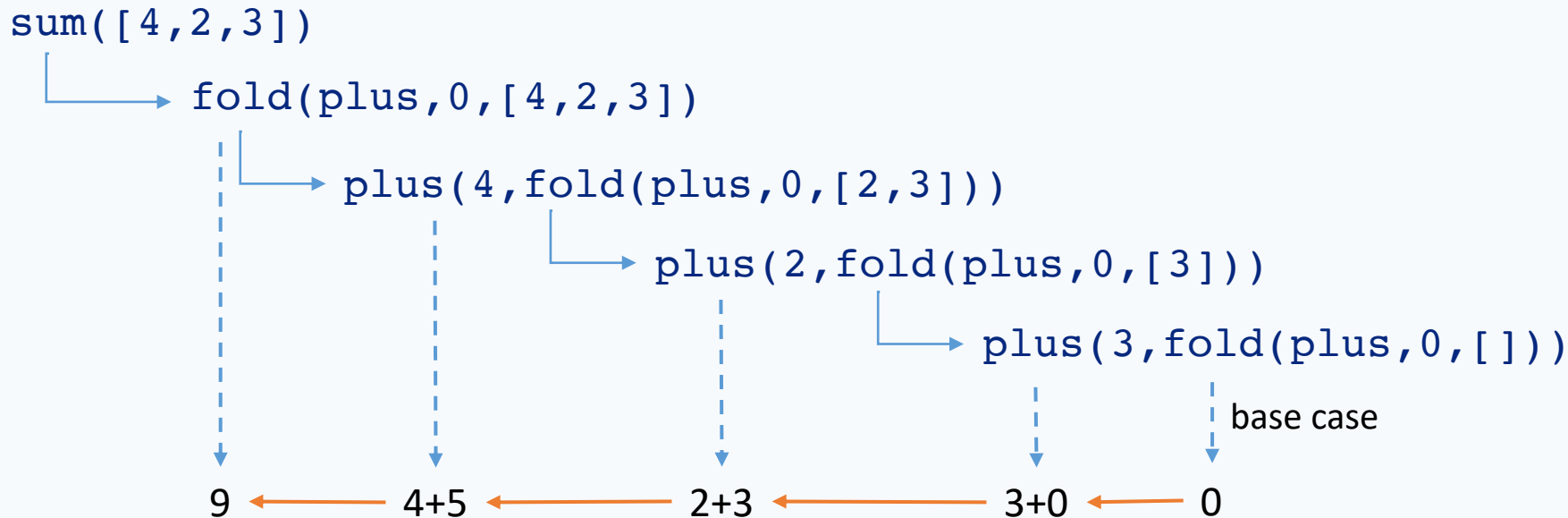
```
fold(_, Result, []) -> Result;
fold(F, Result, [H|T]) -> F(H, fold(F, Result, T)).
```

Let's call this function **plus**

Let's define a **sum** of a list using **fold**

```
sum(L) -> fold( fun(X,Y)-> X+Y end, 0, L )
```

Let's try it!





Remember:

Erlang tutorial!

Friday February 3 at 8:00

# Quiz

## Basic Erlang