HANDBOOK

Bjarne Däcker
Robert Virding

Erlang Handbook

by Bjarne Däcker and Robert Virding

Revision:

Wed Sep 17 22:30:30 2014 +0200

Latest version of this handbook can be found at:
http://opensource.erlang-solutions.com/erlang-handbook

ISBN: 978-1-938616-04-4

Editor

Omer Kilic

Contributors

The list of contributors can be found on the project repository.

Conventions

Syntax specifications are set using `this monotype font`. Square brackets ([ ]) enclose optional parts. Terms beginning with an uppercase letter like *Integer* shall then be replaced by some suitable value. Terms beginning with a lowercase letter like `end` are reserved words in Erlang. A vertical bar (|) separates alternatives, like Integer | Float.

Errata and Improvements

This is a live document so please file corrections and suggestions for improvement about the content using the issue tracker at https://github.com/esl/erlang-handbook. You may also fork this repository and send a pull request with your suggested fixes and improvements. New revisions of this document will be published after major corrections.

Contents

1

Background, or Why Erlang is that it is

Erlang is the result of a project at Ericsson's Computer Science Laboratory to improve the programming of telecommunication applications. A critical requirement was supporting the characteristics of such applications, that include:

- Massive concurrency
- Fault-tolerance
- Isolation
- Dynamic code upgrading at runtime
- Transactions

Throughout the whole of Erlang's history the development process has been extremely pragmatic. The characteristics and properties of the types of systems in which Ericsson was interested drove Erlang's development. These properties were considered to be so fundamental that it was decided to build support for them into the language itself, rather than in libraries. Because of the pragmatic development process, rather than a result of prior planning, Erlang "became" a functional language — since the features of functional languages fitted well with the properties of the systems being developed.

2 Structure of an Erlang program

2.1 Module syntax

An Erlang program is made up of **modules** where each module is a text file with the extension **.erl**. For small programs, all modules typically reside in one directory. A module consists of module attributes and function definitions.

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

double(X) -> times(X, 2).

times(X, N) -> X * N.
```

The above module `demo` consists of the function `times/2` which is local to the module and the function `double/1` which is exported and can be called from outside the module.

`demo:double(10)` ⇒ `20` (the arrow ⇒ should be read as "resulting in")

`double/1` means the function "double" with *one* argument. A function `double/2` taking *two* arguments is regarded as a different function. The number of arguments is called the **arity** of the function.

2.2 Module attributes

A **module attribute** defines a certain property of a module and consists of a **tag** and a **value**:

`-Tag(Value).`

`Tag` must be an atom, while `Value` must be a literal term (see chapter 3). Any module attribute can be specified. The attributes are stored in the compiled code and can be retrieved by calling the function `Module:module_info(attributes).`

2.2.1 Pre-defined module attributes

Pre-defined module attributes must be placed before any function declaration.

- `-module(Module).`

This attribute is mandatory and must be specified first. It defines the name of the module. The name Module, an atom (see section 3.1.1), should be the same as the filename without the '`.erl`' extension.

- `-export([Func1/Arity1, ..., FuncN/ArityN]).`

  This attribute specifies which functions in the module that can be called from outside the module. Each function name `FuncX` is an atom and `ArityX` an integer.

- `-import(Module,[Func1/Arity1, ..., FuncN/ArityN]).`

  This attribute indicates a `Module` from which a list of functions are imported. For example:

  `-import(demo, [double/1]).`

  This means that it is possible to write `double(10)` instead of the longer `demo:double(10)` which can be impractical if the function is used frequently.

- `-compile(Options).`

  Compiler options.

- `-vsn(Vsn).`

  Module version. If this attribute is not specified, the version defaults to the checksum of the module.

- `-behaviour(Behaviour).`

  This attribute either specifies a user defined behaviour or one of the OTP standard behaviours `gen_server`, `gen_fsm`, `gen_event` or `supervisor`. The spelling "behavior" is also accepted.

2.2.2 Macro and record definitions

Records and macros are defined in the same way as module attributes:

```
-record(Record,Fields).

-define(Macro,Replacement).
```

Records and macro definitions are also allowed between functions, as long as the definition comes before its first use. (About records see section 3.2.2 and about macros see chapter 11.)

2.2.3 File inclusion

File inclusion is specified in the same way as module attributes:

```
-include(File).

-include_lib(File).
```

`File` is a string that represents a file name. Include files are typically used for record and macro definitions that are shared by several modules. By convention, the extension `.hrl` is used for include files.

```
-include("my_records.hrl").
-include("incdir/my_records.hrl").
-include("/home/user/proj/my_records.hrl").
```

If `File` starts with a path component `$Var`, then the value of the environment variable `Var` (returned by `os:getenv(Var)`) is substituted for `$Var`.

```
-include("$PROJ_ROOT/my_records.hrl").
```

`include_lib` is similar to `include`, but the first path component is assumed to be the name of an application.

```
-include_lib("kernel/include/file.hrl").
```

The code server uses `code:lib_dir(kernel)` to find the directory of the current (latest) version of `kernel`, and then the subdirectory `include` is searched for the file `file.hrl`.

2.3 Comments

Comments may appear anywhere in a module except within strings and quoted atoms. A comment begins with the percentage character (%) and covers the rest of the line but not the end-of-line. The terminating end-of-line has the effect of a blank.

2.4 Character Set

Erlang handles the full Latin-1 (ISO-8859-1) character set. Thus all Latin-1 printable characters can be used and displayed without the escape backslash. Atoms and variables can use all Latin-1 characters.

Character classes			
Octal	Decimal		Class
40 - 57	32 - 47	! " # $ % & ' /	Punctuation characters
60 - 71	48 - 57	0 - 9	Decimal digits
72 - 100	58 - 64	: ; < = > @	Punctuation characters
101 - 132	65 - 90	A - Z	Uppercase letters
133 - 140	91 - 96	[\ ] ^ _ ‘	Punctuation characters
141 - 172	97 - 122	a - z	Lowercase letters
173 - 176	123 - 126	{ | } ~	Punctuation characters
200 - 237	128 - 159		Control characters
240 - 277	160 - 191	- ¿	Punctuation characters
300 - 326	192 - 214	À - Ö	Uppercase letters
327	215	×	Punctuation character
330 - 336	216 - 222	Ø - Þ	Uppercase letters
337 - 366	223 - 246	ß - ö	Lowercase letters
367	247	÷	Punctuation character
370 - 377	248 - 255	ø - ÿ	Lowercase letters

2.5 Reserved words

The following are reserved words in Erlang:

```
after and andalso band begin bnot bor bsl bsr bxor case catch cond
div end fun if let not of or orelse receive rem try when xor
```

3 Data types (terms)

3.1 Unary data types

3.1.1 Atoms

An **atom** is a symbolic name, also known as a *literal*. Atoms begin with a lower-case letter, and may contain alphanumeric characters, underscores (`_`) or at-signs (`@`). Alternatively atoms can be specified by enclosing them in single quotes (`'`), necessary when they start with an uppercase character or contain characters other than underscores and at-signs. For example:

```
hello
```

```
phone_number
```

```
'Monday'
```

```
'phone number'
```

```
'Anything inside quotes \n\012'
```

(see section 3.3)

3.1.2 Booleans

There is no **boolean** data type in Erlang. The atoms `true` and `false` are used instead.

```
2 =< 3   ⇒  true
true or false   ⇒  true
```

3.1.3 Integers

In addition to the normal way of writing **integers** Erlang provides further notations. `$Char` is the Latin-1 numeric value of the character '`Char`' (that may be an escape sequence) and `Base#Value` is an integer in base `Base`, which must be an integer in the range 2..36.

```
42   ⇒  42
$A   ⇒  65
$\n   ⇒  10
2#101   ⇒  5
16#1f   ⇒  31
```

(see section 3.3)

3.1.4 Floats

A **float** is a real number written `Num[eExp]` where `Num` is a decimal number between 0.01 and 10000 and `Exp` (optional) is a signed integer specifying the power-of-10 exponent. For example:

`2.3e-3` ⇒ `2.30000e-3` (corresponding to $2.3*10^{-3}$)

3.1.5 References

A **reference** is a term which is unique in an Erlang runtime system, created by the built-in function `make_ref/0`. (For more information on built-in functions, or *BIF*s, see section 5.8.)

3.1.6 Ports

A **port identifier** identifies a port (see chapter 9).

3.1.7 Pids

A **process identifier**, *pid*, identifies a process (see chapter 6).

3.1.8 Funs

A *fun* identifies a **functional object** (see section 5.7).

3.2 Compound data types

3.2.1 Tuples

A **tuple** is a compound data type that holds a **fixed number of terms** enclosed within curly braces.

`{Term1,...,TermN}`

Each `TermX` in the tuple is called an **element**. The number of elements is called the **size** of the tuple.

BIFs to manipulate tuples	
`size(Tuple)`	Returns the size of `Tuple`
`element(N,Tuple)`	Returns the N^{th} element in `Tuple`
`setelement(N,Tuple,Expr)`	Returns a new tuple copied from `Tuple` except that the N^{th} element is replaced by `Expr`

```
P = {adam, 24, {july, 29}}   ⇒  P is bound to {adam, 24, {july, 29}}
element(1, P)   ⇒  adam
element(3, P)   ⇒   {july,29}
P2 = setelement(2, P, 25)   ⇒  P2 is bound to {adam, 25, {july, 29}}
size(P)   ⇒  3
size({})   ⇒  0
```

3.2.2 Records

A **record** is a *named tuple* with named elements called **fields**. A record type is defined as a module attribute, for example:

```
-record(Rec, {Field1 [= Value1],
              ...
              FieldN [= ValueN]}).
```

`Rec` and `Fields` are atoms and each `FieldX` can be given an optional default `ValueX`. This definition may be placed amongst the functions of a module, but only before it is used. If a record type is used by several modules it is advisable to put it in a separate file for inclusion.

A new record of type `Rec` is created using an expression like this:

```
# Rec{Field1=Expr1, ..., FieldK=ExprK [, _=ExprL]}
```

The fields need not be in the same order as in the record definition. Fields omitted will get their respective default values. If the final clause is used, omitted fields will get the value `ExprL`. Fields without default values and that are omitted will have their value set to the atom `undefined`.

The value of a field is retrieved using the expression "`Variable#Rec.Field`".

```
-module(employee).
-export([new/2]).
-record(person, {name, age, employed=erixon}).

new(Name, Age) -> #person{name=Name, age=Age}.
```

The function `employee:new/2` can be used in another module which must also include the same record definition of `person`.

```
{P = employee:new(ernie,44)}   ⇒  {person, ernie, 44, erixon}
P#person.age   ⇒  44
P#person.employed   ⇒  erixon
```

When working with records in the Erlang shell, the functions `rd(RecordName, RecordDefinition)` and `rr(Module)` can be used to define and load record definitions. Refer to the *Erlang Reference Manual* for more information.

3.2.3 Lists

A **list** is a compound data type that holds a *variable* number of **terms** enclosed within square brackets.

`[Term1,...,TermN]`

Each term `TermX` in the list is called an **element**. The **length** of a list refers to the number of elements. Common in functional programming, the first element is called the **head** of the list and the remainder (from the 2nd element onwards) is called the **tail** of the list. Note that individual elements within a list do not have to have the same type, although it is common (and perhaps good) practice to do so — where mixed types are involved, **records** are more commonly used.

BIFs to manipulate lists	
`length(List)`	Returns the length of `List`
`hd(List)`	Returns the 1st (head) element of `List`
`tl(List)`	Returns `List` with the 1st element removed (tail)

The vertical bar operator (|) separates the leading elements of a list (one or more) from the remainder. For example:

```
[H | T] = [1, 2, 3, 4, 5]   ⇒  H=1 and T=[2, 3, 4, 5]
[X, Y | Z] = [a, b, c, d, e]   ⇒  X=a, Y=b and Z=[c, d, e]
```

Implicitly a list will end with an empty list, i.e. `[a, b]` is the same as `[a, b | []]`. A list looking like `[a, b | c]` is **badly formed** and should be avoided (because the atom 'c' is *not* a list). Lists lend themselves naturally to recursive functional programming. For example, the following function '`sum`' computes the sum of a list, and '`double`' multiplies each element in a list by 2, constructing and returning a new list as it goes.

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

double([]) -> [];
double([H | T]) -> [H*2 | double(T)].
```

The above definitions introduce *pattern matching*, described in chapter 4. Patterns of this form are common in recursive programming, explicitly providing a "base case" (for the empty list in these examples).

For working with lists, the operator `++` joins two lists together (appends the second argument to the first) and returns the resulting list. The operator `--` produces a list that is a copy of its first argument, except that for each element in the second argument, the first occurrence of this element (if any) in the resulting list is removed.

```
[1,2,3] ++ [4,5]   ⇒  [1,2,3,4,5]
[1,2,3,2,1,2] -- [2,1,2]   ⇒  [3,1,2]
```

A collection of list processing functions can be found in the `STDLIB` module `lists`.

3.2.4 Strings

Strings are character strings enclosed within double quotes but are, in fact, stored as lists of integers.

`"abcdefghi"` is the same as `[97,98,99,100,101,102,103,104,105]`

`""` is the same as `[]`

Two adjacent strings will be concatenated into one at compile-time and do not incur any runtime overhead.

```
"string" "42"   ⇒  "string42"
```

3.2.5 Binaries

A binary is a chunk of untyped memory by default a sequence of 8-bit bytes.

`<<Elem1,...,ElemN>>`

Each `ElemX` is specified as `Value[:Size][/TypeSpecifierList]`.

Element specification		
`Value`	`Size`	`TypeSpecifierList`
Should evaluate to an integer, float or binary	Should evaluate to an integer	A sequence of optional type specifiers, in any order, separated by hyphens (-)

Type specifiers		
Type	`integer` | `float` | `binary`	Default is `integer`
Signedness	`signed` | `unsigned`	Default is `unsigned`
Endianness	`big` | `little` | `native`	CPU dependent. Default is `big`
Unit	`unit:`*IntegerLiteral*	Allowed range is 1..256. Default is 1 for integer and float, and 8 for binary

The value of `Size` multiplied by the unit gives the number of bits for the segment. Each segment can consist of zero or more bits but the total number of bits must be a multiple of 8, or a `badarg` run-time error will occur. Also, a segment of type binary must have a size evenly divisible by 8.

Binaries cannot be nested.

```
<<1, 17, 42>>        % <<1, 17, 42>>
<<"abc">>            % <<97, 98, 99>> (The same as <<$a, $b, $c>>)
<<1, 17, 42:16>>     % <<1,17,0,42>>
<<>>                 % <<>>
<<15:8/unit:10>>     % <<0,0,0,0,0,0,0,0,0,15>>
<<(-1)/unsigned>>    % <<255>>
```

3.3 Escape sequences

Escape sequences are allowed in strings and quoted atoms.

Escape sequences	
\b	Backspace
\d	Delete
\e	Escape
\f	Form feed
\n	New line
\r	Carriage return
\s	Space
\t	Tab
\v	Vertical tab
\XYZ, \XY, \X	Character with octal representation XYZ, XY or X
\^A .. \^Z	Control A to control Z
\^a .. \^z	Control A to control Z
\'	Single quote
\"	Double quote
\\	Backslash

3.4 Type conversions

There are a number of built-in functions for type conversion:

Type conversions								
	atom	integer	float	pid	fun	tuple	list	binary
atom		-	-	-	-	-	X	X
integer	-		X	-	-	-	X	X
float	-	X		-	-	-	X	X
pid	-	-	-		-	-	X	X
fun	-	-	-	-		-	X	X
tuple	-	-	-	-	-		X	X
list	X	X	X	X	X	X		X
binary	X	X	X	X	X	X	X	

The BIF `float/1` converts integers to floats. The BIFs `round/1` and `trunc/1` convert floats to integers.

The BIFs `Type_to_list/1` and `list_to_Type/1` convert to and from lists.

The BIFs `term_to_binary/1` and `binary_to_term/1` convert to and from binaries.

```
atom_to_list(hello)          % "hello"
list_to_atom("hello")        % hello
float_to_list(7.0)           % "7.00000000000000000000e+00"
list_to_float("7.000e+00") % 7.00000
integer_to_list(77)          % "77"
list_to_integer("77")        % 77
tuple_to_list({a, b ,c})     % [a,b,c]
list_to_tuple([a, b, c])     % {a,b,c}
pid_to_list(self())          % "<0.25.0>"
term_to_binary(<<17>>)       % <<131,109,0,0,0,1,17>>
term_to_binary({a, b ,c})  % <<131,104,3,100,0,1,97,100,0,1,98,100,0,1,99>>
binary_to_term(<<131,104,3,100,0,1,97,100,0,1,98,100,0,1,99>>)  % {a,b,c}
term_to_binary(math:pi())  % <<131,99,51,46,49,52,49,53,57,50,54,53,51,...>>
```

4 Pattern Matching

4.1 Variables

Variables are introduced as arguments to a function or as a result of pattern matching. Variables begin with an uppercase letter or underscore (_) and may contain alphanumeric characters, underscores and at-signs (@). Variables can only be bound (assigned) once.

```
Abc
A_long_variable_name
AnObjectOrientatedVariableName
_Height
```

An **anonymous variable** is denoted by a single underscore (_) and can be used when a variable is required but its value can be ignored.

```
[H|_] = [1,2,3]          % H=1 and the rest is ignored
```

Variables beginning with underscore like `_Height` are normal variables, not anonymous. They are however ignored by the compiler in the sense that they will not generate any warnings for unused variables. Thus it is possible to write:

```
member(_Elem, []) ->
    false.
```

instead of:

```
member(_, []) ->
    false.
```

which can make for more readable code.

The *scope* for a variable is its function clause. Variables bound in a branch of an `if`, `case`, or `receive` expression must be bound in all branches to have a value outside the expression, otherwise they will be regarded as *unsafe* (possibly undefined) outside the expression.

4.2 Pattern Matching

A **pattern** has the same structure as a term but may contain new unbound variables.

```
Name1
[H|T]
{error,Reason}
```

Patterns occur in *function heads*, *case*, *receive*, and *try* expressions and in match operator (=) expressions. Patterns are evaluated through **pattern matching** against an expression and this is how variables are defined and bound.

```
Pattern = Expr
```

Both sides of the expression must have the same structure. If the matching succeeds, all unbound variables, if any, in the pattern become bound. If the matching fails, a `badmatch` run-time error will occur.

```
> {A, B} = {answer, 42}.
{answer,42}
> A.
answer
> B.
42
```

4.2.1 Match operator (=) in patterns

If `Pattern1` and `Pattern2` are valid patterns, then the following is also a valid pattern:

```
Pattern1 = Pattern2
```

The `=` introduces an **alias** which when matched against an expression, both `Pattern1` and `Pattern2` are matched against it. The purpose of this is to avoid the reconstruction of terms.

```
foo({connect,From,To,Number,Options}, To) ->
    Signal = {connect,From,To,Number,Options},
    fox(Signal),
    ...;
```

which can be written more efficiently as:

```
foo({connect,From,To,Number,Options} = Signal, To) ->
    fox(Signal),
    ...;
```

4.2.2 String prefix in patterns

When matching strings, the following is a valid pattern:

```
f("prefix" ++ Str) -> ...
```

which is equivalent to and easier to read than:

```
f([$p,$r,$e,$f,$i,$x | Str]) -> ...
```

You can only use strings as prefix expressions; patterns such as `Str ++ "postfix"` are not allowed.

4.2.3 Expressions in patterns

An arithmetic expression can be used within a pattern, provided it only uses numeric or bitwise operators and its value can be evaluated to a constant at compile-time.

```
case {Value, Result} of
    {?Threshold+1, ok} -> ...    % ?Threshold is a macro
```

4.2.4 Matching binaries

```
Bin = <<1, 2, 3>>                % <<1,2,3>> All elements are 8-bit bytes
<<A, B, C>> = Bin                % A=1, B=2 and C=3
<<D:16, E>> = Bin                % D=258 and E=3
<<F, G/binary>> = Bin            % F=1 and G=<<2,3>>
```

In the last line, the variable `G` of unspecified size matches the rest of the binary `Bin`.

Always put a space between (=) and (<<) so as to avoid confusion with the (=<) operator.

5 Functions

5.1 Function definition

A function is defined as a sequence of one or more **function clauses**. The function name is an atom.

```
Func(Pattern11,...,Pattern1N) [when GuardSeq1] -> Body1;
    ...;
    ...;
Func(PatternK1,...,PatternKN) [when GuardSeqK] -> BodyK.
```

The function clauses are separated by semicolons (`;`) and terminated by full stop (`.`). A function clause consists of a **clause head** and a **clause body** separated by an arrow (`->`). A clause head consists of the function name (an atom), arguments within parentheses and an optional guard sequence beginning with the keyword `when`. Each argument is a pattern. A clause body consists of a sequence of expressions separated by commas (`,`).

```
Expr1,
...,
ExprM
```

The number of arguments `N` is the **arity** of the function. A function is uniquely defined by the module name, function name and arity. Two different functions in the same module with different arities may have the same name. A function `Func` in `Module` with arity `N` is often denoted as `Module:Func/N`.

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

area({square, Side}) -> Side * Side;
area({circle, Radius}) -> math:pi() * Radius * Radius;
area({triangle, A, B, C}) ->
    S = (A + B + C)/2,
    math:sqrt(S*(S-A)*(S-B)*(S-C)).
```

5.2 Function calls

A function is called using:

```
[Module:]Func(Expr1, ..., ExprN)
```

`Module` evaluates to a module name and `Func` to a function name or a *fun*. When calling a function in another module, the module name must be provided and the function must be exported. This is referred to as a **fully qualified function name**.

```
lists:keysearch(Name, 1, List)
```

The module name can be omitted if `Func` evaluates to the name of a local function, an imported function, or an auto-imported BIF. In such cases, the function is called using an **implicitly qualified function name**.

Before calling a function the arguments `ExprX` are evaluated. If the function cannot be found, an `undef` run-time error will occur. Next the function clauses are scanned sequentially until a clause is found such that the patterns in the clause head can be successfully matched against the given arguments and that the guard sequence, if any, is true. If no such clause can be found, a `function_clause` run-time error will occur.

If a matching clause is found, the corresponding clause body is evaluated, i.e. the expressions in the body are evaluated sequentially and the value of the last expression is returned.

The fully qualified function name must be used when calling a function with the same name as a BIF (built-in function, see section 5.8). The compiler does not allow defining a function with the same name as an imported function. When calling a local function, there is a difference between using the implicitly or fully qualified function name, as the latter always refers to the latest version of the module (see chapter 10).

5.3 Expressions

An **expression** is either a term or the invocation of an operator, for example:

```
Term
op Expr
Expr1 op Expr2
(Expr)
begin
   Expr1,
   ...,
   ExprM            % no comma (,) before end
end
```

The simplest form of expression is a term, i.e. an *integer*, *float*, *atom*, *string*, *list* or *tuple* and the return value is the term itself. There are both *unary* and *binary* operators. An expression may contain *macro* or *record* expressions which will expanded at compile time.

Parenthesised expressions are useful to override operator precedence (see section 5.3.5):

```
1 + 2 * 3           % 7
(1 + 2) * 3         % 9
```

Block expressions within `begin...end` can be used to group a sequence of expressions and the return value is the value of the last expression `ExprM`.

All subexpressions are evaluated before the expression itself is evaluated, but the order in which subexpressions are evaluated is undefined.

Most operators can only be applied to arguments of a certain type. For example, arithmetic operators can only be applied to integers or floats. An argument of the wrong type will cause a `badarg` run-time error.

5.3.1 Term comparisons

```
Expr1 op Expr2
```

A **term comparison** returns a *boolean* value, in the form of atoms `true` or `false`.

Comparison operators			
==	Equal to	=<	Less than or equal to
/=	Not equal to	<	Less than
=:=	Exactly equal to	>=	Greater than or equal to
=/=	Exactly not equal to	>	Greater than

```
1==1.0              % true
1=:=1.0             % false
1 > a               % false
```

The arguments may be of different data types. The following order is defined:

`number < atom < reference < fun < port < pid < tuple < list < binary`

Lists are compared element by element. Tuples are ordered by size, two tuples with the same size are compared element by element. When comparing an integer and a float, the integer is first converted to a float. In the case of `=:=` or `=/=` there is no type conversion.

5.3.2 Arithmetic expressions

```
op Expr
Expr1 op Expr2
```

An **arithmetic expression** returns the result after applying the operator.

Arithmetic operators		
+	Unary +	`Integer | Float`
-	Unary -	`Integer | Float`
+	Addition	`Integer | Float`
-	Subtraction	`Integer | Float`
*	Multiplication	`Integer | Float`
/	Floating point division	`Integer | Float`
bnot	Unary bitwise not	`Integer`
div	Integer division	`Integer`
rem	Integer remainder of X/Y	`Integer`
band	Bitwise and	`Integer`
bor	Bitwise or	`Integer`
bxor	Arithmetic bitwise xor	`Integer`
bsl	Arithmetic bitshift left	`Integer`
bsr	Bitshift right	`Integer`

```
+1                  % 1
4/2                 % 2.00000
5 div 2             % 2
5 rem 2             % 1
2#10 band 2#01      % 0
2#10 bor 2#01       % 3
```

5.3.3 Boolean expressions

```
op Expr
Expr1 op Expr2
```

A **boolean expression** returns the value `true` or `false` after applying the operator.

Boolean operators	
`not`	Unary logical not
`and`	Logical and
`or`	Logical or
`xor`	Logical exclusive or

```
not true            % false
true and false      % false
true xor false      % true
```

5.3.4 Short-circuit boolean expressions

```
Expr1 orelse Expr2
Expr1 andalso Expr2
```

These are boolean expressions where `Expr2` is evaluated only if necessary. In an `orelse` expression `Expr2` will be evaluated only if `Expr1` evaluates to false. In an `andalso` expression `Expr2` will be evaluated only if `Expr1` evaluates to true.

```
if A >= 0 andalso math:sqrt(A) > B -> ...

if is_list(L) andalso length(L) == 1 -> ...
```

5.3.5 Operator precedences

In an expression consisting of subexpressions the operators will be applied according to a defined **operator precedence** order:

Operator precedence (from high to low)	
`:`	
`#`	
`Unary + - bnot not`	
`/ * div rem band and`	Left associative
`+ - bor bxor bsl bsr or xor`	Left associative
`++ --`	Right associative
`== /= =< < >= > =:= =/=`	
`andalso`	
`orelse`	
`= !`	Right associative
`catch`	

The operator with the highest priority is evaluated first. Operators with the same priority are evaluated according to their **associativity**. The left associative arithmetic operators are evaluated left to right:

`6 + 5 * 4 - 3 / 2` ⇒ `6 + 20 - 1.5` ⇒ `26 - 1.5` ⇒ `24.5`

5.4 Compound expressions

5.4.1 If

```
if
    GuardSeq1 ->
        Body1;
    ...;
    GuardSeqN ->
        BodyN                    % Note no semicolon (;) before end
end
```

The branches of an `if` expression are scanned sequentially until a guard sequence `GuardSeq` which evaluates to `true` is found. The corresponding `Body` (sequence of expressions separated by commas) is then evaluated. The return value of `Body` is the return value of the `if` expression.

If no guard sequence is true, an `if_clause` run-time error will occur. If necessary, the guard expression `true` can be used in the last branch, as that guard sequence is always true (known as a "catch all").

```
is_greater_than(X, Y) ->
    if
        X>Y ->
            true;
        true ->                  % works as an 'else' branch
            false
    end
```

It should be noted that pattern matching in function clauses can be used to replace `if` cases (most of the time). Overuse of `if` sentences withing function bodies is considered a bad Erlang practice.

5.4.2 Case

Case expressions provide for inline pattern matching, similar to the way in which function clauses are matched.

```
case Expr of
    Pattern1 [when GuardSeq1] ->
        Body1;
        ...;
    PatternN [when GuardSeqN] ->
        BodyN                    % Note no semicolon (;) before end
end
```

The expression `Expr` is evaluated and the patterns `Pattern1`...`PatternN` are sequentially matched against the result. If a match succeeds and the optional guard sequence `GuardSeqX` is `true`, then the corresponding `BodyX` is evaluated. The return value of `BodyX` is the return value of the case expression.

If there is no matching pattern with a true guard sequence, a `case_clause` run-time error will occur.

```
is_valid_signal(Signal) ->
    case Signal of
        {signal, _What, _From, _To} ->
            true;
        {signal, _What, _To} ->
            true;
        _Else ->                 % 'catch all'
            false
    end.
```

5.4.3 List comprehensions

List comprehensions are analogous to the `setof` and `findall` predicates in Prolog.

```
[Expr || Qualifier1,...,QualifierN]
```

`Expr` is an arbitrary expression, and each `QualifierX` is either a **generator** or a **filter**. A generator is written as:

```
Pattern <- ListExpr
```

where `ListExpr` must be an expression which evaluates to a list of terms. A filter is an expression which evaluates to `true` or `false`. Variables in list generator expressions *shadow* variables in the function clause surrounding the list comprehension.

The qualifiers are evaluated from left to right, the generators creating values and the filters constraining them. The list comprehension then returns a list where the elements are the result of evaluating `Expr` for each combination of the resulting values.

```
> [{X, Y} || X <- [1,2,3,4,5,6], X > 4, Y <- [a,b,c]].
[{5,a},{5,b},{5,c},{6,a},{6,b},{6,c}]
```

5.5 Guard sequences

A **guard sequence** is a set of **guards** separated by semicolons (`;`). The guard sequence is `true` if at least one of the guards is `true`.

```
Guard1; ...; GuardK
```

A **guard** is a set of **guard expressions** separated by commas (`,`). The guard is `true` if all guard expressions evaluate to `true`.

```
GuardExpr1, ..., GuardExprN
```

The permitted **guard expressions** (sometimes called guard tests) are a subset of valid Erlang expressions, since the evaluation of a guard expression must be guaranteed to be free of side-effects.

Valid guard expressions:	
The atom `true`;	
Other constants (terms and bound variables), are all regarded as `false`;	
Term comparisons;	
Arithmetic and boolean expressions;	
Calls to the BIFs specified below.	
Type test BIFs	Other BIFs allowed in guards:
`is_atom/1`	`abs(Integer | Float)`
`is_constant/1`	`float(Term)`
`is_integer/1`	`trunc(Integer | Float)`
`is_float/1`	`round(Integer | Float)`
`is_number/1`	`size(Tuple | Binary)`
`is_reference/1`	`element(N, Tuple)`
`is_port/1`	`hd(List)`
`is_pid/1`	`tl(List)`
`is_function/1`	`length(List)`
`is_tuple/1`	`self()`
`is_record/2` The 2nd argument is the record name	`node()`
`is_list/1`	`node(Pid | Ref |Port)`
`is_binary/1`	

A small example:

```
fact(N) when N>0 ->             % first clause head
    N * fact(N-1);              % first clause body
fact(0) ->                      % second clause head
    1.                          % second clause body
```

5.6 Tail recursion

If the last expression of a function body is a function call, a **tail recursive** call is performed in such a way that no system resources (like the call stack) are consumed. This means that an infinite loop like a server can be programmed provided it only uses tail recursive calls.

The function `fact/1` above could be rewritten using tail recursion in the following manner:

```
fact(N) when N>1 -> fact(N, N-1);
fact(N) when N==1; N==0 -> 1.

fact(F,0) -> F;                 % The variable F is used as an accumulator
fact(F,N) -> fact(F*N, N-1).
```

5.7 Funs

A **fun** defines a *functional object*. Funs make it possible to pass an entire function, not just the function name, as an argument. A 'fun' expression begins with the keyword `fun` and ends with the keyword `end` instead of a full stop (`.`). Between these should be a regular function declaration, except that no function name is specified.

```
fun
    (Pattern11,...,Pattern1N) [when GuardSeq1] ->
        Body1;
        ...;
    (PatternK1,...,PatternKN) [when GuardSeqK] ->
        BodyK
end
```

Variables in a `fun` head *shadow* variables in the function clause surrounding the `fun` but variables bound in a `fun` body are local to the body. The return value of the expression is the resulting function. The expression `fun Name/N is` equivalent to:

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

The expression `fun Module:Func/Arity` is also allowed, provided that `Func` is exported from `Module`.

```
Fun1 = fun (X) -> X+1 end.
Fun1(2)          % 3

Fun2 = fun (X) when X>=1000 -> big; (X) -> small end.
Fun2(2000)       % big
```

Anonymous `funs`: When a `fun` is anonymous, i.e. there is no function name in the definition of the `fun`, the definition of a recursive `fun` has to be done in two steps. This example shows how to define an anonymous `fun sum(List)` (see section 3.2.3) as an anonymous `fun`.

```
Sum1 = fun ([], _Foo) -> 0;([H|T], Foo) -> H + Foo(T, Foo) end.
Sum = fun (List) -> Sum1(List, Sum1) end.
Sum([1,2,3,4,5])     % 15
```

The definition of `Sum` is done in a way such that it takes *itself* as a parameter, matched to `_Foo` (empty list) or `Foo`, which it then calls recursively. The definition of `Sum` calls `Sum1`, also passing `Sum1` as a parameter.

Names in `funs`: In Erlang you can use a name inside a `fun` before the name has been defined. The syntax of `funs with names` allows a variable name to be consistently present before each argument list. This allows `funs` to be recursive in one steps. This example shows how to define the function `sum(List)` (see section 3.2.3) as a `funs with names`.

```
Sum = fun Sum([])-> 0;Sum([H|T]) -> H + Sum(T) end.
Sum([1,2,3,4,5])     % 15
```

5.8 BIFs — Built-in functions

The **built-in functions**, BIFs, are implemented in the C code of the runtime system and do things that are difficult or impossible to implement in Erlang. Most of the built-in functions belong to the module `erlang` but there are also built-in functions that belong to other modules like `lists` and `ets`. The most commonly

used BIFs belonging to the module `erlang` are **auto-imported**, i.e. they do not need to be prefixed with the module name.

Some useful BIFs	
`date()`	Returns today's date as `{Year, Month, Day}`
`now()`	Returns current time in microseconds. System dependent
`time()`	Returns current time as `{Hour, Minute, Second}` System dependent
`halt()`	Stops the Erlang system
`processes()`	Returns a list of all processes currently known to the system
`process_info(Pid)`	Returns a dictionary containing information about `Pid`
`Module:module_info()`	Returns a dictionary containing information about the code in Module

A **dictionary** is a list of `{Key, Value}` terms (see also section 6.8).

```
size({a, b, c})              % 3
atom_to_list('Erlang')       % "Erlang"
date()                       % {2013,5,27}
time()                       % {01,27,42}
```

6 Processes

A **process** corresponds to one *thread of control*. Erlang permits very large numbers of concurrent processes, each executing like it had an own virtual processor. When a process executing `functionA` calls another `functionB`, it will wait until `functionB` is finished and then retrieve its result. If instead it *spawns* another process executing `functionB`, both will continue in parallel (concurrently). `functionA` will not wait for `functionB` and the only way they can communicate is through *message passing*.

Erlang processes are light-weight with a small memory footprint, fast to create and shut-down, and the scheduling overhead is low. A **process identifier**, `Pid`, identifies a process. The BIF `self/0` returns the `Pid` of the calling process.

6.1 Process creation

A process is created using the BIF `spawn/3`.

```
spawn(Module, Func, [Expr1, ..., ExprN])
```

`Module` should evaluate to a module name and `Func` to a function name in that module. The list `Expr1...ExprN` are the arguments to the function. `spawn` creates a new process and returns the process identifier, `Pid`. The new process starts by executing:

```
Module:Func(Expr1, ..., ExprN)
```

The function `Func` has to be exported even if it is spawned by another function in the same module. There are other spawn BIFs, for example `spawn/4` for spawning a process on another node.

6.2 Registered processes

A process can be associated with a name. The name must be an atom and is automatically unregistered if the process terminates. Only static (cyclic) processes should be registered.

Name registration BIFs	
`register(Name, Pid)`	Associates the atom `Name` with the process `Pid`
`registered()`	Returns a list of names which have been registered
`whereis(Name)`	Returns the `Pid` registered under `Name` or `undefined` if the name is not registered

6.3 Process communication

Processes communicate by sending and receiving **messages**. Messages are sent using the send operator (`!`) and are received using `receive`. Message passing is asynchronous and reliable, i.e. the message is guaranteed to eventually reach the recipient, provided that the recipient exists.

6.3.1 Send

```
Pid ! Expr
```

The send (`!`) operator sends the value of `Expr` as a message to the process specified by `Pid` where it will be placed last in its **message queue**. The value of `Expr` is also the return value of the (`!`) expression. `Pid` must evaluate to a process identifier, a registered name or a tuple `{Name,Node}`, where `Name` is a registered process at `Node` (see chapter 8). The message sending operator (`!`) never fails, even if it addresses a non-existent process.

6.3.2 Receive

```
receive
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...
    PatternN [when GuardSeqN] ->
        BodyN                    % Note no semicolon (;) before end
end
```

This expression receives messages sent to the process using the send operator (`!`). The patterns `PatternX` are sequentially matched against the first message in time order in the message queue, then the second and so on. If a match succeeds and the optional guard sequence `GuardSeqX` is true, then the message is removed from the message queue and the corresponding `BodyX` is evaluated. It is the order of the pattern clauses that decides the order in which messages will be received prior to the order in which they arrive. This is called *selective receive*. The return value of `BodyX` is the return value of the receive expression.

`receive` never fails. The process may be suspended, possibly indefinitely, until a message arrives that matches one of the patterns and with a true guard sequence.

```
wait_for_onhook() ->
    receive
        onhook ->
            disconnect(),
            idle();
        {connect, B} ->
            B ! {busy, self()},
            wait_for_onhook()
    end.
```

6.3.3 Receive with timeout

```
receive
    Pattern1 [when GuardSeq1] ->
        Body1;
        ...;
    PatternN [when GuardSeqN] ->
        BodyN
after
    ExprT ->
        BodyT
end
```

`ExprT` should evaluate to an integer between `0` and `16#ffffffff` (the value must fit in 32 bits). If no matching message has arrived within `ExprT` milliseconds, then `BodyT` will be evaluated and its return value becomes the return value of the receive expression.

```
wait_for_onhook() ->
    receive
        onhook ->
            disconnect(),
            idle();
        {connect, B} ->
            B ! {busy, self()},
            wait_for_onhook()
    after
        60000 ->
            disconnect(),
            error()
    end.
```

A `receive...after` expression with no branches can be used to implement simple timeouts.

```
receive
after
    ExprT ->
        BodyT
end
```

Two special cases for the timeout value `ExprT`	
`infinity`	This is equivalent to not using a timeout and can be useful for timeout values that are calculated at run-time
`0`	If there is no matching message in the mailbox, the timeout will occur immediately

6.4 Process termination

A process always terminates with an **exit reason** which may be any term. If a process terminates normally, i.e. it has run to the end of its code, then the reason is the atom `normal`. A process can terminate itself by calling one of the following BIFs.

```
exit(Reason)

erlang:error(Reason)

erlang:error(Reason, Args)
```

A process terminates with the exit reason `{Reason,Stack}` when a run-time error occurs.

A process may also be terminated if it receives an exit signal with a reason other than `normal` (see section 6.5.3).

6.5 Process links

Two processes can be **linked** to each other. Links are bidirectional and there can only be one link between two distinct processes (unique `Pids`). A process with `Pid1` can link to a process with `Pid2` using the BIF `link(Pid2)`. The BIF `spawn_link(Module, Func, Args)` spawns and links a process in one atomic operation.

A link can be removed using the BIF `unlink(Pid)`.

6.5.1 Error handling between processes

When a process terminates it will send **exit signals** to all processes that it is linked to. These in turn will also be terminated *or handle the exit signal in some way*. This feature can be used to build hierarchical program structures where some processes are supervising other processes, for example restarting them if they terminate abnormally.

6.5.2 Sending exit signals

A process always terminates with an exit reason which is sent as an exit signal to all linked processes. The BIF `exit(Pid, Reason)` sends an exit signal with the reason `Reason` to `Pid`, without affecting the calling process.

6.5.3 Receiving exit signals

If a process receives an exit signal with an exit reason other than `normal` it will also be terminated, and will send exit signals with the same exit reason to its linked processes. An exit signal with reason `normal` is ignored. This behaviour can be changed using the BIF `process_flag(trap_exit, true)`.

The process is then able to **trap exits**. This means that an exit signal will be transformed into a message `{'EXIT', FromPid, Reason}` which is put into the process's mailbox and can be handled by the process like a regular message using `receive`.

However, a call to the BIF `exit(Pid, kill)` unconditionally terminates the process `Pid` regardless whether it is able to trap exit signals or not.

6.6 Monitors

A process `Pid1` can create a **monitor** for `Pid2` using the BIF:

```
erlang:monitor(process, Pid2)
```

which returns a reference `Ref`. If `Pid2` terminates with exit reason `Reason`, a message as follows will be sent to `Pid1`:

```
{'DOWN', Ref, process, Pid2, Reason}
```

If `Pid2` does not exist, the `'DOWN'` message is sent immediately with `Reason` set to `noproc`. Monitors are unidirectional in that if `Pid1` monitors `Pid2` then it will receive a message when `Pid2` dies but `Pid2` will **not** receive a message when `Pid1` dies. Repeated calls to `erlang:monitor(process, Pid)` will create several, independent monitors and each one will be sent a `'DOWN'` message when `Pid` terminates.

A monitor can be removed by calling `erlang:demonitor(Ref)`. It is possible to create monitors for processes with registered names, also at other nodes.

6.7 Process priorities

The BIF `process_flag(priority, Prio)` defines the priority of the current process. `Prio` may have the value `normal`, which is the default, `low`, `high` or `max`.

Modifying a process's priority is discouraged and should only be done in special circumstances. A problem that requires changing process priorities can generally be solved by another approach.

6.8 Process dictionary

Each process has its own process dictionary which is a list of `{Key, Value}` terms.

Process dictionary BIFs	
`put(Key, Value)`	Saves the `Value` under the `Key` or replaces an older value
`get(Key)`	Retrieves the value stored under `Key` or `undefined`
`get()`	Returns the entire process dictionary as a list of `{Key, Value}` terms
`get_keys(Value)`	Returns a list of keys that have the value `Value`
`erase(Key)`	Deletes `{Key, Value}`, if any, and returns `Key`
`erase()`	Returns the entire process dictionary and deletes it

Process dictionaries could be used to keep global variables within an application, but the extensive use of them for this is usually regarded as poor programming style.

7 Error handling

This chapter deals with error handling within a process. Such errors are known as **exceptions**.

7.1 Exception classes and error reasons

Exception classes	
`error`	Run-time error for example when applying an operator to the wrong types of arguments. Run-time errors can be raised by calling the BIFs `erlang:error(Reason)` or `erlang:error(Reason, Args)`
`exit`	The process calls `exit(Reason)`, see section 6.4
`throw`	The process calls `throw(Expr)`, see section 7.2

An exception will cause the process to crash, i.e. its execution is stopped and it is removed from the system. It is also said to *terminate*. Then exit signals will be sent to any linked processes. An exception consists of its class, an exit reason and a stack. The stack trace can be retrieved using the BIF `erlang:get_stacktrace/0`.

Run-time errors and other exceptions can be prevented from causing the process to terminate by using the expressions `try` and `catch`.

For exceptions of class `error`, for example normal run-time errors, the **exit reason** is a tuple `{Reason, Stack}` where `Reason` is a term indicating which type of error.

Exit reasons	
`badarg`	Argument is of wrong type.
`badarith`	Argument is of wrong type in an arithmetic expression.
`{badmatch, Value}`	Evaluation of a match expression failed. `Value` did not match.
`function_clause`	No matching function clause is found when evaluating a function call.
`{case_clause, Value}`	No matching branch is found when evaluating a case expression. `Value` did not match.
`if_clause`	No true branch is found when evaluating an `if` expression.
`{try_clause, Value}`	No matching branch is found when evaluating the of section of a `try` expression. `Value` did not match.
`undef`	The function cannot be found when evaluating a function call
`{badfun, Fun}`	There is something wrong with `Fun`
`{badarity, Fun}`	A fun is applied to the wrong number of arguments. `Fun` describes it and the arguments
`timeout_value`	The timeout value in a `receive...after` expression is evaluated to something else than an integer or infinity
`noproc`	Trying to link to a non-existant process
`{nocatch, Value}`	Trying to evaluate a `throw` outside of a `catch`. `Value` is the thrown term
`system_limit`	A system limit has been reached

`Stack` is the stack of function calls being evaluated when the error occurred, given as a list of tuples `{Module, Name, Arity}` with the most recent function call first. The most recent function call tuple may in some cases be `{Module, Name, Args}`.

7.2 Catch and throw

```
catch Expr
```

This returns the value of `Expr` unless an exception occurs during its evaluation. Then the return value will be a tuple containing information about the exception.

```
{'EXIT', {Reason, Stack}}
```

Then the exception is *caught*. Otherwise it would terminate the process. If the exception is caused by a function call `exit(Term)` the tuple `{'EXIT',Term}` is returned. If the exception is caused by calling `throw(Term)` then `Term` will be returned.

```
catch 1+2   ⇒  3
catch 1+a    ⇒  {'EXIT',{badarith,[...]}}
```

`catch` has low precedence and catch subexpressions often need to be enclosed in a block expression or in parentheses.

```
A = (catch 1+2)   ⇒  3
```

The BIF `throw(Expr)` is used for *non-local* return from a function. It must be evaluated within a `catch`, which returns the result from evaluating `Expr`.

`catch begin 1,2,3,throw(four),5,6 end` ⇒ `four`

If `throw/1` is not evaluated within a `catch`, a `nocatch` run-time error will occur.

A `catch` will not prevent a process from terminating due to an exit signal from another linked process (unless it has been set to trap exits).

7.3 Try

The `try` expression is able to distinguish between different exception classes. The following example emulates the behaviour of `catch Expr`:

```
try Expr
catch
    throw:Term -> Term;
    exit:Reason -> {'EXIT', Reason};
    error:Reason -> {'EXIT',{Reason, erlang:get_stacktrace()}}
end
```

The full description of `try` is as follows:

```
try Expr [of
    Pattern1 [when GuardSeq1] -> Body1;
    ...;
    PatternN [when GuardSeqN] -> BodyN]
[catch
    [Class1:]ExceptionPattern1 [when ExceptionGuardSeq1] -> ExceptionBody1;
    ...;
    [ClassN:]ExceptionPatternN [when ExceptionGuardSeqN] -> ExceptionBodyN]
[after AfterBody]
end
```

There has to be at least one `catch` or an `after` clause. There may be an `of` clause following the `Expr` which adds a `case` expression on the value of `Expr`.

`try` returns the value of `Expr` unless an exception occurs during its evaluation. Then the exception is *caught* and the patterns `ExceptionPattern` with the right exception `Class` are sequentially matched against the caught exception. An omitted `Class` is shorthand for `throw`. If a match succeeds and the optional guard sequence `ExceptionGuardSeq` is true, the corresponding `ExceptionBody` is evaluated and becomes the return value.

If there is no matching `ExceptionPattern` of the right `Class` with a true guard sequence, the exception is passed on as if `Expr` had not been enclosed in a `try` expression. An exception occurring during the evaluation of an `ExceptionBody` it is not caught.

If none of the of `Patterns` match, a `try_clause` run-time error will occur.

If defined then `AfterBody` is always evaluated **last** irrespective of whether and error occurred or not. Its return value is ignored and the return value of the `try` is the same as without an `after` section. `AfterBody` is evaluated even if an exception occurs in `Body` or `ExceptionBody`, in which case the exception is passed on.

An exception that occurs during the evaluation of `AfterBody` itself is not caught, so if the `AfterBody` is evaluated due to an exception in `Expr`, `Body` or `ExceptionBody`, that exception is lost and masked by the new exception.

8 Distributed Erlang

A **distributed Erlang system** consists of a number of Erlang runtime systems communicating with each other. Each such runtime system is called a **node**. Nodes can reside on the same host or on different hosts connected through a network. The standard distribution mechanism is implemented using TCP/IP sockets but other mechanisms can also be implemented.

Message passing between processes on different nodes, as well as links and monitors, is transparent when using `Pid`s. However, registered names are local to each node. A registered process at a particular node is referred to as `{Name,Node}`.

The Erlang Port Mapper Daemon **epmd** is automatically started on every host where an Erlang node is started. It is responsible for mapping the symbolic node names to machine addresses.

8.1 Nodes

A **node** is an executing Erlang runtime system which has been given a name, using the command line flag `-name` (long name) or `-sname` (short name).

The format of the node name is an atom `Name@Host` where `Name` is the name given by the user and `Host` is the full host name if long names are used, or the first part of the host name if short names are used. `node()` returns the name of the node. Nodes using long names cannot communicate with nodes using short names.

8.2 Node connections

The nodes in a distributed Erlang system are fully connected. The first time the name of another node is used, a connection attempt to that node will be made. If a node A connects to node B, and node B has a connection to node C, then node A will also try to connect to node C. This feature can be turned off using the command line flag:

```
-connect_all false
```

If a node goes down, all connections to that node are removed. The BIF:

```
erlang:disconnect(Node)
```

disconnects `Node`. The BIF `nodes()` returns the list of currently connected (visible) nodes.

8.3 Hidden nodes

It is sometimes useful to connect to a node without also connecting to all other nodes. For this purpose, a **hidden node** may be used. A hidden node is a node started with the command line flag `-hidden`. Connections between hidden nodes and other nodes must be set up explicitly. Hidden nodes do not show up in the list of nodes returned by `nodes()`. Instead, `nodes(hidden)` or `nodes(connected)` must be used. A hidden node will not be included in the set of nodes that the module global keeps track of.

A **C node** is a C program written to act as a hidden node in a distributed Erlang system. The library `erl_interface` contains functions for this purpose.

8.4 Cookies

Each node has its own **magic cookie**, which is an atom. The Erlang network authentication server (auth) reads the cookie in the file `$HOME/.erlang.cookie`. If the file does not exist, it will be created with a random string as content.

The permissions of the file must be set to octal 400 (read-only by user). The cookie of the local node may also be set using the BIF `erlang:set_cookie(node(), Cookie)`.

The current node is only allowed to communicate with another node `Node2` if it knows its cookie. If this is different from the current node (whose cookie will be used by default) it must be explicitly set with the BIF `erlang:set_cookie(Node2, Cookie2)`.

8.5 Distribution BIFs

Distribution BIFs	
`node()`	Returns the name of the current node. Allowed in guards
`is_alive()`	Returns true if the runtime system is a node and can connect to other nodes, false otherwise
`erlang:get_cookie()`	Returns the magic cookie of the current node
`set_cookie(Node, Cookie)`	Sets the magic cookie used when connecting to `Node`. If `Node` is the current node, `Cookie` will be used when connecting to all new nodes
`nodes()`	Returns a list of all visible nodes to which the current node is connected to
`nodes(connected|hidden)`	Returns a list not only of visible nodes, but also hidden nodes and previously known nodes, etc.
`monitor_node(Node, true|false)`	Monitors the status of `Node`. A message `{nodedown, Node}` is received if the connection to it is lost
`node(Pid|Ref|Port)`	Returns the node where the argument is located
`erlang:disconnect_node(Node)`	Forces the disconnection of `Node`
`spawn[_link|_opt](Node, Module, Function, Args)`	Creates a process at a remote node
`spawn[_link|_opt](Node, Fun)`	Creates a process at a remote node

8.6 Distribution command line flags

Distribution command line flags	
`-connect_all false`	Only explicit connection set-ups will be used
`-hidden`	Makes a node into a hidden node
`-name Name`	Makes a runtime system into a node, using long node names
`-setcookie Cookie`	Same as calling `erlang:set_cookie(node(), Cookie))`
`-sname Name`	Makes a runtime system into a node, using short node names

8.7 Distribution modules

There are several modules available which are useful for distributed programming:

Kernel modules useful for distribution	
`global`	A global name registration facility
`global_group`	Grouping nodes to global name registration groups
`net_adm`	Various net administration routines
`net_kernel`	Erlang networking kernel
STDLIB modules useful for distribution	
`slave`	Start and control of slave nodes

9 Ports and Port Drivers

Ports provide a byte-oriented interface to external programs and communicate with Erlang processes by sending and receiving lists of bytes as messages. The Erlang process that creates a port is called the **port owner** or the **connected process** of the port. All communication to and from the port should go via the port owner. If the port owner terminates, so will the port (and the external program, if it has been programmed correctly).

The external program forms another OS process. By default, it should read from standard input (file descriptor 0) and write to standard output (file descriptor 1). The external program should terminate when the port is closed.

9.1 Port Drivers

Drivers are normally programmed in C and are dynamically linked to the Erlang runtime system. The linked-in driver behaves like a port and is called a **port driver**. However, an erroneous port driver might cause the entire Erlang runtime system to leak memory, hang or crash.

9.2 Port BIFs

Port creation BIF	
`open_port(PortName, PortSettings)`	Returns a **port identifier** `Port` as the result of opening a new Erlang port. Messages can be sent to and received from a port identifier, just like a Pid. Port identifiers can also be linked to or registered under a name using `link/1` and `register/2`.

`PortName` is usually a tuple `{spawn,Command}` where the string `Command` is the name of the external program. The external program runs outside the Erlang workspace unless a port driver with the name `Command` is found. If the driver is found, it will be started.

`PortSettings` is a list of settings (options) for the port. The list typically contains at least a tuple `{packet,N}` which specifies that data sent between the port and the external program are preceded by an N-byte length indicator. Valid values for `N` are 1, 2 or 4. If binaries should be used instead of lists of bytes, the option `binary` must be included.

The port owner Pid communicates with Port by sending and receiving messages. (Any process could send the messages to the port, but messages from the port will always be sent to the port owner).

Messages sent to a port	
`{Pid, {command, Data}}`	Sends Data to the port.
`{Pid, close}`	Closes the port. Unless the port is already closed, the port replies with `{Port, closed}` when all buffers have been flushed and the port really closes.
`{Pid,{connect,NewPid}}`	Sets the port owner of `Port` to `NewPid`. Unless the port is already closed, the port replies with `{Port, connected}` to the old port owner. Note that the old port owner is still linked to the port, but the new port owner is not.

Data must be an I/O list. An I/O list is a binary or a (possibly deep) list of binaries or integers in the range 0..255.

Messages received from a port	
`{Port, {data, Data}}`	Data is received from the external program
`{Port, closed}`	Reply to `Port ! {Pid,close}`
`{Port, connected}`	Reply to `Port ! {Pid,{connect, NewPid}}`
`{'EXIT', Port, Reason}`	If Port has terminated for some reason.

Instead of sending and receiving messages, there are also a number of BIFs that can be used. These can be called by any process, not only the port owner.

Port BIFs	
`port_command(Port, Data)`	Sends Data to Port
`port_close(Port)`	Closes Port
`port_connect(Port, NewPid)`	Sets the port owner of `Port` to `NewPid`. The old port owner Pid stays linked to the port and has to call `unlink(Port)` if this is not desired.
`erlang:port_info(Port, Item)`	Returns information as specified by `Item`
`erlang:ports()`	Returns a list of all ports on the current node

There are some additional BIFs that only apply to port drivers: `port_control/3` and `erlang:port_call/3`.

10 Code loading

Erlang supports code updating in a running system. Code replacement is performed at module level.

The code of a module can exist in two versions in a system: **current** and **old**. When a module is loaded into the system for the first time, the code becomes *current*. If a new instance of the module is loaded, the code of the previous instance becomes *old* and the new instance becomes *current*. Normally a module is automatically loaded the first time a function in it is called. If the module is already loaded then it must explicitly be loaded again to a new version.

Both old and current code are valid, and may be used concurrently. Fully qualified function calls will always refer to the current code. However, the old code may still be run by other processes.

If a third instance of the module is loaded, the code server will remove (*purge*) the old code and any processes lingering in it are terminated. Then the third instance becomes *current* and the previously current code becomes *old*.

To change from old code to current code, a process must make a fully qualified function call.

```
-module(mod).
-export([loop/0]).

loop() ->
    receive
        code_switch ->
            mod:loop();
        Msg ->
            ...
            loop()
    end.
```

To make the process change code, send the message `code_switch` to it. The process then will make a fully qualified call to `mod:loop()` and change to the current code. Note that `mod:loop/0` must be exported.

11 Macros

11.1 Defining and using macros

```
-define(Const, Replacement).
-define(Func(Var1, ..., VarN), Replacement).
```

A **macro** must be defined before it is used but a macro definition may be placed anywhere among the attributes and function declarations of a module. If a macro is used in several modules it is advisable to put the macro definition in an include file. A macro is used as follows:

```
?Const
?Func(Arg1,...,ArgN)
```

Macros are expanded during compilation. A macro reference `?Const` is replaced by `Replacement` like this:

```
-define(TIMEOUT, 200).
...
call(Request) ->
    server:call(refserver, Request, ?TIMEOUT).
```

is expanded to:

```
call(Request) ->
    server:call(refserver, Request, 200).
```

A macro reference `?Func(Arg1, ..., ArgN)` will be replaced by `Replacement`, where all occurrences of a variable `VarX` from the macro definition are replaced by the corresponding argument `ArgX`.

```
-define(MACRO1(X, Y), {a, X, b, Y}).
...
bar(X) ->
    ?MACRO1(a, b),
    ?MACRO1(X, 123).
```

will be expanded to:

```
bar(X) ->
    {a, a, b, b},
```

```
    {a, X, b, 123}.
```

To view the result of macro expansion, a module can be compiled with the 'P' option:

```
compile:file(File, ['P']).
```

This produces a listing of the parsed code after preprocessing and parse transforms, in the file File.P.

11.2 Predefined macros

Predefined macros	
?MODULE	The name of the current module
?MODULE_STRING	The name of the current module, as a string
?FILE	The file name of the current module
?LINE	The current line number
?MACHINE	The machine name, 'BEAM'

11.3 Flow Control in Macros

```
-undef(Macro).      % This inhibits the macro definition.

-ifdef(Macro).
    %% Lines that are evaluated if Macro was defined
-else.
    %% If the condition was false, these lines are evaluated instead.
-endif.
```

ifndef(Macro) can be used instead of ifdef and means the opposite.

```
-ifdef(debug).
-define(LOG(X), io:format("{~p,~p}:~p~n",[?MODULE,?LINE,X])).
-else.
-define(LOG(X), true).
-endif.
```

If debug is defined when the module is compiled, ?LOG(Arg) will expand to a call to io:format/2 and provide the user with some simple trace output.

11.4 Stringifying Macro Arguments

??Arg, where Arg is a macro argument expands to the argument in the form of a string.

```
-define(TESTCALL(Call), io:format("Call ~s: ~w~n", [??Call, Call])).

?TESTCALL(myfunction(1,2)),
?TESTCALL(you:function(2,1)),
```

results in:

```
io:format("Call ~s: ~w~n", ["myfunction(1,2)", m:myfunction(1,2)]),
io:format("Call ~s: ~w~n", ["you:function(2,1)", you:function(2,1)]),
```

That is, a trace output with both the function called and the resulting value.

12 Further Reading and Resources

Following websites provide in-depth explanation of topics and concepts briefly covered in this document:

- Official Erlang documentation: http://www.erlang.org/doc/
- Learn You Some Erlang for Great Good: http://learnyousomeerlang.com/
- Tutorials section at Erlang Central: https://erlangcentral.org/wiki/index.php?title=Category:HowTo

Still have questions? erlang-questions mailing list (http://erlang.org/mailman/listinfo/erlang-questions) is a good place for general discussions about Erlang/OTP, the language, implementation, usage and beginners questions.