

Erlang Reference Manual

version 5.3

Typeset in L^AT_EX from SGML source using the DOCBUILDER 3.2.2 Document System.

Contents

1	Erlang Reference Manual	1
1.1	Introduction	1
1.1.1	Purpose	1
1.1.2	Prerequisites	1
1.1.3	Document Conventions	1
1.2	Data Types	2
1.2.1	Terms	2
1.2.2	Number	2
1.2.3	Atom	2
1.2.4	Binary	3
1.2.5	Reference	3
1.2.6	Fun	3
1.2.7	Port Identifier	3
1.2.8	Pid	4
1.2.9	Tuple	4
1.2.10	List	5
1.2.11	String	6
1.2.12	Record	6
1.2.13	Boolean	6
1.2.14	Escape Sequences	7
1.2.15	Type Conversions	8
1.3	Pattern Matching	9
1.3.1	Pattern Matching	9
1.4	Modules	9
1.4.1	Module Syntax	9
1.4.2	Module Attributes	10
1.4.3	Comments	12
1.5	Functions	12
1.5.1	Function Declaration Syntax	12
1.5.2	Function Evaluation	13

1.5.3	Built-In Functions, BIFs	14
1.6	Expressions	14
1.6.1	Expression Evaluation	14
1.6.2	Terms	14
1.6.3	Variables	15
1.6.4	Patterns	16
1.6.5	Match	17
1.6.6	Function Calls	17
1.6.7	If	18
1.6.8	Case	18
1.6.9	Send	19
1.6.10	Receive	19
1.6.11	Term Comparisons	21
1.6.12	Arithmetic Expressions	22
1.6.13	Boolean Expressions	23
1.6.14	Short-Circuit Boolean Expressions	23
1.6.15	List Operations	24
1.6.16	Bit Syntax Expressions	24
1.6.17	Fun Expressions	24
1.6.18	Catch and Throw	25
1.6.19	Parenthesized Expressions	26
1.6.20	Block Expressions	26
1.6.21	List Comprehensions	27
1.6.22	Guard Sequences	27
1.6.23	Operator Precedence	28
1.7	Macros	29
1.7.1	Defining and Using Macros	29
1.7.2	Predefined Macros	30
1.7.3	Flow Control in Macros	30
1.7.4	Stringifying Macro Arguments	31
1.8	Records	31
1.8.1	Defining Records	32
1.8.2	Creating Records	32
1.8.3	Accessing Record Fields	33
1.8.4	Updating Records	33
1.8.5	Records in Guards	33
1.8.6	Records in Patterns	33
1.8.7	Internal Representation of Records	34
1.9	Errors and Error Handling	34
1.9.1	Terminology	34

1.9.2	Handling of Run-Time Errors in Erlang	35
1.9.3	Exit Reasons	35
1.10	Processes	35
1.10.1	Processes	35
1.10.2	Process Creation	36
1.10.3	Registered Processes	36
1.10.4	Process Termination	36
1.10.5	Message Sending	36
1.10.6	Error Handling	37
1.10.7	Monitoring Processes	37
1.10.8	Process Dictionary	37
1.11	Distributed Erlang	38
1.11.1	Distribution BIFs	38
1.11.2	Distribution Command Line Flags	38
1.11.3	Distribution Modules	39
1.11.4	Nodes	39
1.11.5	Node Connections	40
1.11.6	Hidden Nodes	40
1.11.7	epmd	40
1.11.8	Security	40
1.12	Compilation and Code Loading	41
1.12.1	Compilation	41
1.12.2	Code Loading	41
1.12.3	Code Replacement	42

List of Tables	43
-----------------------	-----------

Chapter 1

Erlang Reference Manual

1.1 Introduction

1.1.1 Purpose

This reference manual describes the Erlang programming language. The focus is on the language itself, not the implementation. The language constructs are described in text and with examples rather than formally specified, with the intention to make the manual more readable. The manual is not intended as a tutorial.

Many references are made to built-in functions, BIFs. For a complete list of BIFs, their arguments and return values, refer to `erlang(3)`.

For documentation about this implementation of Erlang, please refer to ERTS User's Guide.

1.1.2 Prerequisites

It is assumed that the reader has done some programming and is familiar with concepts such as data types and programming language syntax.

1.1.3 Document Conventions

In the document, the following terminology is used:

- A *sequence* is one or more items. For example, a clause body consists of a sequence of expressions. This means that there must be at least one expression.
- A *list* is any number of items. For example, an argument list can consist of zero, one or more arguments.

If a feature has been added recently, in Erlang 5.0/OTP R7 or later, this is mentioned in the text.

1.2 Data Types

1.2.1 Terms

Erlang provides a number of data types which are listed in this chapter. A piece of data of any data type is called a *term*.

1.2.2 Number

There are two types of numeric literals, *integers* and *floats*. Besides the conventional notation, there are two Erlang-specific notations:

- `$char`
ASCII value of the character `char`.
- `base#value`
Integer with the base `base`, which must be an integer in the range 2..32.
In Erlang 5.2/OTP R9B and earlier versions, the allowed range is 2..16.

Examples:

```
1> 42.
42
2> $A.
65
3> $_n.
10
4> 2#101.
5
5> 16#1f.
31
6> 2.3.
2.30000
7> 2.3e3.
2300.00
8> 2.3e-3.
2.30000e-3
```

1.2.3 Atom

An atom is a literal, a constant with name. An atom should 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:

```
hello
phone_number
'Monday'
'phone number'
```

1.2.4 Binary

A binary is used to store an area of untyped memory.

Binaries are expressed using the bit syntax [page 24].

Examples:

```
1> <<10,20>> .  
<<10,20>>  
2> <<"ABC">> .  
<<65,66,67>>
```

More examples can be found in Programming Examples.

1.2.5 Reference

A reference is a term which is unique in an Erlang runtime system, created by calling `make_ref/0`.

1.2.6 Fun

A fun is a functional object. Funs make it possible to create an anonymous function and pass the function itself – not its name – as argument to other functions.

Example:

```
1> Fun1 = fun (X) -> X+1 end.  
#Fun<erl_eval.6.39074546>  
2> Fun1(2).  
3
```

Read more about funs in Fun Expressions [page 24]. More examples can be found in Programming Examples.

1.2.7 Port Identifier

A port identifier identifies an Erlang port. `open_port/2`, which is used to create ports, will return a value of this type.

Read about Erlang ports in the Interoperability Tutorial and in the documentation of `open_port/2` in `erlang(3)`.

1.2.8 Pid

A process identifier, `pid`, identifies a process. `spawn/1,2,3,4`, `spawn_link/1,2,3,4` and `spawn_opt/4`, which are used to create processes, return values of this type. Example:

```
1> spawn(m, f, []).  
<0.51.0>
```

The BIF `self()` returns the pid of the calling process. Example:

```
-module(m).  
-export([loop/0]).  
  
loop() ->  
    receive  
        who_are_you ->  
            io:format("I am ~p~n", [self()]),  
            loop()  
    end.  
  
1> P = spawn(m, loop, []).  
<0.58.0>  
2> P ! who_are_you.  
I am <0.58.0>  
who_are_you
```

Read more about processes in Processes [page 35].

1.2.9 Tuple

Compound data type with a fixed number of terms:

`{Term1, ..., TermN}`

Each term `Term` in the tuple is called an *element*. The number of elements is said to be the *size* of the tuple.

There exists a number of BIFs to manipulate tuples.

Examples:

```
1> P = {adam,24,{july,29}}.  
{adam,24,{july,29}}  
2> element(1,P).  
adam  
3> element(3,P).  
{july,29}  
4> P2 = setelement(2,P,25).  
{adam,25,{july,29}}  
5> size(P).
```

```

3
6> size({}).
0

```

1.2.10 List

Compound data type with a variable number of terms.

`[Term1, ..., TermN]`

Each term `Term` in the list is called an *element*. The number of elements is said to be the *length* of the list.

Formally, a list is either the empty list `[]` or consists of a *head* (first element) and a *tail* (remainder of the list) which is also a list. The latter can be expressed as `[H|T]`. The notation `[Term1, ..., TermN]` above is actually shorthand for the list `[Term1|[...|[TermN|[]]]]`.

Example:

`[]` is a list, thus

`[c|[]]` is a list, thus

`[b|[c|[]]]` is a list, thus

`[a|[b|[c|[]]]]` is a list, or in short `[a,b,c]`.

A list where the tail is a list is sometimes called a *proper list*. It is allowed to have a list where the tail is not a list, for example `[a|b]`. However, this type of list is of little practical use.

Examples:

```

1> L1 = [a,2,{c,4}].
[a,2,{c,4}]
2> [H|T] = L1.
[a,2,{c,4}]
3> H.
a
4> T.
[2,{c,4}]
5> L2 = [d|T].
[d,2,{c,4}]
6> length(L1).
3
7> length([]).
0

```

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

1.2.11 String

Strings are enclosed in double quotes ("), but is not a data type in Erlang. Instead a string "hello" is shorthand for the list [\$h,\$e,\$l,\$l,\$o], that is [104,101,108,108,111].

Two adjacent string literals are concatenated into one. This is done at compile-time and does not incur any runtime overhead. Example:

```
"string" "42"
```

is equivalent to

```
"string42"
```

1.2.12 Record

A record is a data structure for storing a fixed number of elements. It has named fields and is similar to a struct in C. However, record is not a true data type. Instead record expressions are translated to tuple expressions during compilation. Therefore, record expressions are not understood by the shell.

Examples:

```
-module(person).  
-export([new/2]).  
  
-record(person, {name, age}).  
  
new(Name, Age) ->  
    #person{name=Name, age=Age}.  
  
1> person:new(ernie, 44).  
    {person,ernie,44}
```

Read more about records in Records [page 31]. More examples can be found in Programming Examples.

1.2.13 Boolean

There is no Boolean data type in Erlang. Instead the atoms true and false are used to denote Boolean values.

Examples:

```
1> 2=<3.  
true  
2> true or false.  
true
```

1.2.14 Escape Sequences

Within strings and quoted atoms, the following escape sequences are recognized:

<i>Sequence</i>	<i>Description</i>
<code>\b</code>	backspace
<code>\d</code>	delete
<code>\e</code>	escape
<code>\f</code>	form feed
<code>\n</code>	newline
<code>\r</code>	carriage return
<code>\s</code>	space
<code>\t</code>	tab
<code>\v</code>	vertical tab
<code>\XYZ, \YZ, \Z</code>	character with octal representation XYZ, YZ or Z
<code>\^a...\^z \^A...\^Z</code>	control A to control Z
<code>\'</code>	single quote
<code>\"</code>	double quote
<code>\\</code>	backslash

Table 1.1: Recognized Escape Sequences.

1.2.15 Type Conversions

There are a number of BIFs for type conversions. Examples:

```
1> atom_to_list(hello).
"hello"
2> list_to_atom("hello").
hello
3> binary_to_list(<<"hello">>).
"hello"
4> binary_to_list(<<104,101,108,108,111>>).
"hello"
5> list_to_binary("hello").
<<104,101,108,108,111>>
6> float_to_list(7.0).
"7.0000000000000000000000e+00"
7> list_to_float("7.000e+00").
7.00000
8> integer_to_list(77).
"77"
9> list_to_integer("77").
77
10> tuple_to_list({a,b,c}).
[a,b,c]
11> list_to_tuple([a,b,c]).
{a,b,c}
12> term_to_binary({a,b,c}).
<<131,104,3,100,0,1,97,100,0,1,98,100,0,1,99>>
13> binary_to_term(<<131,104,3,100,0,1,97,100,0,1,98,100,0,1,99>>).
{a,b,c}
```

1.3 Pattern Matching

1.3.1 Pattern Matching

Variables are bound to values through the *pattern matching* mechanism. Pattern matching occurs when evaluating a function call, case- and receive- expressions and match operator (=) expressions.

In a pattern matching, a lefthand side pattern is matched against a righthand side term. If the matching succeeds, any unbound variables in the pattern become bound. If the matching fails, a run-time error occurs.

Examples:

```
1> X.
** 1: variable 'X' is unbound **
2> X = 2.
2
3> X + 1.
3
4> {X, Y} = {1, 2}.
** exited: {{badmatch,{1,2}},...} **
5> {X, Y} = {2, 3}.
{2,3}
6> Y.
3
```

1.4 Modules

1.4.1 Module Syntax

Erlang code is divided into *modules*. A module consists of a sequence of attributes and function declarations, each terminated by period (.). Example:

```
-module(m).           % module attribute
-export([fact/1]).    % module attribute

fact(N) when N>0 ->  % beginning of function declaration
    N * fact(N-1);   % |
fact(0) ->           % |
    1.               % end of function declaration
```

See the Functions [page 12] chapter for a description of function declarations.

1.4.2 Module Attributes

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

`-Tag(Term).`

Any module attribute can be specified. The attributes are stored in the compiled code and can be retrieved by using, for example, the function `beam_lib:chunks/2`.

There exists a number of module attributes with predefined meanings.

Pre-Defined Module Attributes

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

`-module(Module).` Module declaration, defining the name of the module. The name `Module`, an atom, should be the same as the file name minus the extension `erl`. Otherwise code loading [page 41] will not work as intended.

This attribute should be specified first and is the only attribute which is mandatory.

`-export(Functions).` Exported functions. Specifies which of the functions defined within the module that are visible outside the module.

`Functions` is a list `[Name1/Arity1, ..., NameN/ArityN]`, where each `NameI` is an atom and `ArityI` an integer.

`-import(Module,Functions).` Imported functions. Imported functions can be called the same way as local functions, that is without any module prefix.

`Module`, an atom, specifies which module to import functions from. `Functions` is a list similar as for `export` above.

`-compile(Options).` Compiler options. `Options`, which is a single option or a list of options, will be added to the option list when compiling the module. See `compile(3)`.

Behaviour Module Attribute

It is possible to specify that the module is the callback module for a *behaviour*:

`-behaviour(Behaviour).`

The atom `Behaviour` gives the name of the behaviour, which can be 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.

Read more about behaviours and callback modules in OTP Design Principles.

Macro and Record Definitions

The same syntax as for module attributes is used for macro and record definitions:

```
-define(Macro,Replacement).  
-record(Record,Fields).
```

Macro and record definitions are allowed anywhere in a module, also among the function declarations. Read more in [Macros](#) [page 29] and [Records](#) [page 31].

File Inclusion

The same syntax as for module attributes is used for file inclusion:

```
-include(File).  
-include_lib(File).
```

`File`, a string, should point out a file. The contents of this file are included as-is, at the position of the directive.

Include files are typically used for record- and macro definitions that are shared by several modules. It is recommended that the file name extension `.hrl` be used for include files.

`File` may start with a path component `$VAR`, for some string `VAR`. If that is the case, the value of the environment variable `VAR` as returned by `os:getenv(VAR)` is substituted for `$VAR`. If `os:getenv(VAR)` returns `false`, `$VAR` is left as is.

If the filename `File` is absolute (possibly after variable substitution), the include file with that name is included. Otherwise, the specified file is searched for in the current working directory, in the same directory as the module being compiled, and in the directories given by the `include` option, in that order. See `erlc(1)` and `compile(3)` for details.

Examples:

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

`include_lib` is similar to `include`, but should not point out an absolute file. Instead, the first path component (possibly after variable substitution) is assumed to be the name of an application. Example:

```
-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`.

1.4.3 Comments

Comments are allowed anywhere within a module. Comments start with the character % and stop at end of line.

1.5 Functions

1.5.1 Function Declaration Syntax

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

A *function clause* consists of a clause head and a clause body, separated by `->`.

A clause *head* consists of the function name, an argument list, and an optional guard sequence beginning with the keyword `when`.

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

The function name is an atom. Each argument is a pattern.

The number of arguments *N* is the *arity* of the function. A function is uniquely defined by the module name, function name and arity. That is, two functions with the same name and in the same module, but with different arities are two completely different functions.

A function named *f* in the module *m* and with arity *N* is often denoted as *m:f/N*.

A clause *body* consists of a sequence of expressions separated by comma (,):

```
Expr1,
...,
ExprN
```

Valid Erlang expressions and guard sequences are described in Erlang Expressions [page 14].

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
```

1.5.2 Function Evaluation

When a function $m:f/N$ is called, first the code for the function is located. If the function cannot be found, an `undef` run-time error will occur. Note that the function must be exported to be visible outside the module it is defined in.

If the function is found, the function clauses are scanned sequentially until a clause is found that fulfills the following two conditions:

1. the patterns in the clause head can be successfully matched against the given arguments, and
2. the guard sequence, if any, is true.

If such a clause cannot be found, a `function_clause` run-time error will occur.

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

Example: Consider the function `fact`:

```
-module(m).
-export([fact/1]).

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

Assume we want to calculate factorial for 1:

```
1> m:fact(1).
```

Evaluation starts at the first clause. The pattern `N` is matched against the argument 1. The matching succeeds and the guard (`N>0`) is true, thus `N` is bound to 1 and the corresponding body is evaluated:

```
N * fact(N-1) => (N is bound to 1)
1 * fact(0)
```

Now `fact(0)` is called and the function clauses are scanned sequentially again. First, the pattern `N` is matched against 0. The matching succeeds, but the guard (`N>0`) is false. Second, the pattern 0 is matched against 0. The matching succeeds and the body is evaluated:

```
1 * fact(0) =>
1 * 1 =>
1
```

Evaluation has succeed and `m:fact(1)` returns 1.

If `m:fact/1` is called with a negative number as argument, no clause head will match. A `function_clause` run-time error will occur.

1.5.3 Built-In Functions, BIFs

Built-in functions, BIFs, are implemented in C code in 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 belonging to a few other modules, for example `lists` and `ets`.

The most commonly used BIFs belonging to `erlang` are *auto-imported*, they do not need to be prefixed with the module name. Which BIFs are auto-imported is specified in `erlang(3)`. For example, standard type conversion BIFs like `atom_to_list` and BIFs allowed in guards can be called without specifying the module name. Examples:

```
1> size({a,b,c}).
3
2> atom_to_list('Erlang').
"Erlang"
```

Note that normally it is the set of auto-imported built-in functions that is referred to when talking about 'BIFs'.

1.6 Expressions

In this chapter, all valid Erlang expressions are listed. When writing Erlang programs, it is also allowed to use macro- and record expressions. However, these expressions are expanded during compilation and are in that sense not true Erlang expressions. Macro- and record expressions are covered in separate chapters: Macros [page 29] and Records [page 31].

1.6.1 Expression Evaluation

All subexpressions are evaluated before an expression itself is evaluated, unless explicitly stated otherwise. For example, consider the expression:

`Expr1 + Expr2`

`Expr1` and `Expr2`, which are also expressions, are evaluated first - in any order - before the addition is performed.

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

1.6.2 Terms

The simplest form of expression is a term, that is an integer, float, atom, string, list or tuple. The return value is the term itself.

1.6.3 Variables

A variable is an expression. If a variable is bound to a value, the return value is this value. Unbound variables are only allowed in patterns.

Variables start with an uppercase letter or underscore (`_`) and may contain alphanumeric characters, underscore and `@`. Examples:

```
X
Name1
PhoneNumber
Phone_number
-
_Height
```

Variables are bound to values using pattern matching [page 9]. Erlang uses *single assignment*, a variable can only be bound once.

The *anonymous variable* is denoted by underscore (`_`) and can be used when a variable is required but its value can be ignored. Example:

```
[H|_] = [1,2,3]
```

Variables starting with underscore (`_`), for example `_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. Example: The following code

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

can be rewritten to be more readable:

```
member(Elem, []) ->
    [].
```

This will however cause a warning for an unused variable `Elem`, if the code is compiled with the flag `warn_unused_vars` set. Instead, the code can be rewritten to:

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

The scope for a variable is the body it is introduced in. Also, variables introduced in every branch of an `if`, `case` or `receive` expression are implicitly exported from this expression.

1.6.4 Patterns

A pattern has the same structure as a term but may contain unbound variables. Example:

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

Patterns are allowed in clause heads, case and receive expressions, and match expressions.

Match Operator = in Patterns

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

```
Pattern1 = Pattern2
```

When matched against a term, both `Pattern1` and `Pattern2` will be matched against the term. The idea behind this feature is to avoid reconstruction of terms. Example:

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

can instead be written as

```
f({connect,_,To,_,_} = Signal, To) ->
    ...;
f(Signal, To) ->
    ignore.
```

String Prefix in Patterns

When matching strings, the following is a valid pattern:

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

This is syntactic sugar for the equivalent, but harder to read

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

Expressions in Patterns

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

```
case {Value, Result} of
    {?THRESHOLD+1, ok} -> ...
```

This feature was added in Erlang 5.0/OTP R7.

1.6.5 Match

`Expr1 = Expr2`

Matches `Expr1`, a pattern, against `Expr2`. If the matching succeeds, any unbound variable in the pattern becomes bound and the value of `Expr2` is returned.

If the matching fails, a `badmatch` run-time error will occur.

Examples:

```
1> {A, B} = {answer, 42}.
{answer,42}
2> A.
answer
3> {C, D} = [1, 2].
** exited: {{badmatch,[1,2]},{erl_eval,expr,3}} **
```

1.6.6 Function Calls

`ExprF(Expr1, ..., ExprN)`
`ExprM:ExprF(Expr1, ..., ExprN)`

`ExprM` should evaluate to a module name and `ExprF` to a function name or a fun.

When including the module name, the function is said to be called by using the *fully qualified function name*. This is often referred to as a *remote* or *external function call*.

The module name can be omitted, if `ExprF` evaluates to the name of a local function, an imported function, or an auto-imported BIF. Then the function is said to be called by using the *implicitly qualified function name*.

To avoid possible ambiguities, the fully qualified function name must be used when calling a function with the same name as a BIF, and the compiler does not allow defining a function with the same name as an imported function.

Note that when calling a local function, there is a difference between using the implicitly or fully qualified function name, as the latter always refer to the latest version of the module. See *Compilation and Code Loading* [page 41].

If `ExprF` evaluates to a fun, only the format `ExprF(Expr1, ..., ExprN)` is correct.

See also the chapter about *Function Evaluation* [page 13].

1.6.7 If

```
if
    GuardSeq1 ->
        Body1;
    ...;
    GuardSeqN ->
        BodyN
end
```

The branches of an if-expression are scanned sequentially until a guard sequence `GuardSeq` which evaluates to true is found. Then the corresponding `Body` (sequence of expressions separated by ',') is 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.

Example:

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

1.6.8 Case

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

The expression `Expr` is evaluated and the patterns `Pattern` are sequentially matched against the result. If a match succeeds and the optional guard sequence `GuardSeq` is true, the corresponding `Body` is evaluated.

The return value of `Body` 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.

Example:

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

1.6.9 Send

`Expr1 ! Expr2`

Sends the value of `Expr2` as a message to the process specified by `Expr1`. The value of `Expr2` is also the return value of the expression.

`Expr1` must evaluate to a pid, a registered name (atom) or a tuple `{Name, Node}`, where `Name` is an atom and `Node` a node name, also an atom.

- If `Expr1` evaluates to a name, but this name is not registered, a `badarg` run-time error will occur.
- Sending a message to a pid never fails, even if the pid identifies a non-existing process.
- Distributed message sending, that is if `Expr1` evaluates to a tuple `{Name, Node}` (or a pid located at another node), also never fails.

1.6.10 Receive

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

Receives messages sent to the process using the send operator (!). The patterns `Pattern` are sequentially matched against the first message in time order in the mailbox, then the second, and so on. If a match succeeds and the optional guard sequence `GuardSeq` is true, the corresponding `Body` is evaluated. The matching message is consumed, that is removed from the mailbox, while any other messages in the mailbox remain unchanged.

The return value of `Body` is the return value of the `receive` expression.

`receive` never fails. Execution is suspended, possibly indefinitely, until a message arrives that does match one of the patterns and with a true guard sequence.

Example:

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

It is possible to augment the receive expression with a timeout:

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

ExprT should evaluate to an integer. `receive..after` works exactly as `receive`, except that if no matching message has arrived within ExprT milliseconds, then BodyT is evaluated instead and its return value becomes the return value of the `receive..after` expression.

Example:

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

It is legal to use a `receive..after` expression with no branches:

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

This construction will not consume any messages, only suspend execution in the process for `ExprT` milliseconds and can be used to implement simple timers.

Example:

```
timer() ->
    spawn(m, timer, [self()]).

timer(Pid) ->
    receive
    after
        5000 ->
            Pid ! timeout
    end.
```

There are two special cases for the timeout value `ExprT`:

infinity The process should wait indefinitely for a matching message – this is the same as not using a timeout. 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.

1.6.11 Term Comparisons

`Expr1` op `Expr2`

<i>op</i>	<i>Description</i>
<code>==</code>	equal to
<code>/=</code>	not equal to
<code>=<</code>	less than or equal to
<code><</code>	less than
<code>>=</code>	greater than or equal to
<code>></code>	greater than
<code>==:</code>	exactly equal to
<code>=/=</code>	exactly not equal to

Table 1.2: Term Comparison Operators.

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.

All comparison operators except `==` and `!=` are of type coerce: When comparing an integer and a float, the integer is first converted to a float. In the case of `==` and `!=`, there is no type conversion.

Returns the Boolean value of the expression, `true` or `false`.

Examples:

```
1> 1==1.0.
true
2> 1:=1.0.
false
3> 1 > a.
false
```

1.6.12 Arithmetic Expressions

op Expr
Expr1 op Expr2

<i>op</i>	<i>Description</i>	<i>Argument type</i>
+	unary +	number
-	unary -	number
+		number
-		number
*		number
/	floating point division	number
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

Table 1.3: Arithmetic Operators.

Examples:

```
1> +1.
1
2> -1.
-1
3> 1+1.
2
4> 4/2.
```

```

2.00000
5> 5 div 2.
2
6> 5 rem 2.
1
7> 2#10 band 2#01.
0
8> 2#10 bor 2#01.
3

```

1.6.13 Boolean Expressions

```

op Expr
Expr1 op Expr2

```

<i>op</i>	<i>Description</i>
not	unary logical not
and	logical and
or	logical or
xor	logical xor

Table 1.4: Logical Operators.

Examples:

```

1> not true.
false
2> true and false.
false
3> true xor false.
true

```

1.6.14 Short-Circuit Boolean Expressions

```

Expr1 orelse Expr2
Expr1 andalso Expr2

```

Boolean expressions where Expr2 is evaluated only if necessary. That is, Expr2 is evaluated only if Expr1 evaluates to false in an orelse expression, or only if Expr1 evaluates to true in an andalso expression. Returns the Boolean value of the expression, that is true or false.

Short-circuit boolean expressions are not allowed in guards. In guards, however, evaluation is always short-circuited since guard tests are known to be free of side effects.

Example 1:

```
case A >= -1.0 andalso math:sqrt(A+1) > B of
```

This will work even if A is less than -1.0, since in that case, `math:sqrt/1` is never evaluated.

Example 2:

```
OnlyOne = is_atom(L) orelse  
          (is_list(L) andalso length(L) == 1),
```

This feature was added in Erlang 5.1/OTP R8.

1.6.15 List Operations

```
Expr1 ++ Expr2  
Expr1 -- Expr2
```

The list concatenation operator `++` appends its second argument to its first and returns the resulting list.

The list subtraction operator `--` produces a list which is a copy of the first argument, subjected to the following procedure: for each element in the second argument, the first occurrence of this element (if any) is removed.

Example:

```
1> [1,2,3]++[4,5].  
[1,2,3,4,5]  
2> [1,2,3,2,1,2]--[2,1,2].  
[3,1,2]
```

1.6.16 Bit Syntax Expressions

This chapter is under construction. Read about the bit syntax in Programming Examples.

1.6.17 Fun Expressions

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

A fun expression begins with the keyword `fun` and ends with the keyword `end`. Between them should be a function declaration, similar to a regular function declaration [page 12], except that no function name is specified.

The return value of the expression is the resulting fun.

Examples:

```
1> Fun1 = fun (X) -> X+1 end.
#Fun<erl_eval.6.39074546>
2> Fun1(2).
3
3> Fun2 = fun (X) when X>=5 -> gt; (X) -> lt end.
#Fun<erl_eval.6.39074546>
4> Fun2(7).
gt
```

The following fun expression is also allowed:

```
fun Name/Arity
```

`Name` is an atom and `Arity` is an integer `N` which should specify an existing local function. The expression is syntactic sugar for:

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

More examples can be found in Programming Examples.

1.6.18 Catch and Throw

```
catch Expr
```

Returns the value of `Expr` unless a run-time error occurs during the evaluation. In that case, the error is caught and `{'EXIT', {Reason, Stack}}` is returned instead.

`Reason` depends on the type of error that occurred, and `Stack` is the stack of recent function calls, see Errors and Error Handling [page 35].

Examples:

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

Note that `catch` has low precedence and catch subexpressions often needs to be enclosed in brackets:

```
3> A = catch 1+2.
** 1: syntax error before: 'catch' **
4> A = (catch 1+2).
3
```

The BIF `throw(Any)` can be used for non-local return from a function. It must be evaluated within a `catch`, which will return the value `Any`. Example:

```
5> catch throw(hello).
hello
```

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

1.6.19 Parenthesized Expressions

(Expr)

Parenthesized expressions are useful to override operator precedences [page 28], for example in arithmetic expressions:

```
1> 1 + 2 * 3.
7
2> (1 + 2) * 3.
9
```

1.6.20 Block Expressions

```
begin
  Expr1,
  ...,
  ExprN
end
```

Block expressions provide a way to group a sequence of expressions, similar to a clause body. The return value is the value of the last expression `ExprN`.

1.6.21 List Comprehensions

List comprehensions are a feature of many modern functional programming languages. Subject to certain rules, they provide a succinct notation for generating elements in a list.

List comprehensions are analogous to set comprehensions in Zermelo-Frankel set theory and are called ZF expressions in Miranda. They are analogous to the `setof` and `findall` predicates in Prolog.

List comprehensions are written with the following syntax:

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

`Expr` is an arbitrary expression, and each `Qualifier` is either a generator or a filter.

- A *generator* is written as:
`Pattern <- ListExpr.`
`ListExpr` must be an expression which evaluates to a list of terms.
- A *filter* is an expression which evaluates to `true` or `false`.

The variables in the generator patterns shadow variables in the body surrounding the list comprehensions.

A list comprehension returns a list, where the elements are the result of evaluating `Expr` for each combination of generator list elements for which all filters are true.

Example:

```
1> [X*2 || X <- [1,2,3]].  
[2,4,6]
```

More examples can be found in Programming Examples.

1.6.22 Guard Sequences

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

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

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

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

The set of valid *guard expressions* (sometimes called guard tests) is a subset of the set of valid Erlang expressions. The reason for restricting the set of valid expressions is that evaluation of a guard expression must be guaranteed to be free of side effects. Valid guard expressions are:

- the atom `true`,
- other constants (terms and bound variables), all regarded as `false`,
- calls to the BIFs specified below,
- term comparisons,
- arithmetic expressions, and
- boolean expressions.

<code>is_atom/1</code>
<code>is_binary/1</code>
<code>is_constant/1</code>
<code>is_float/1</code>
<code>is_function/1</code>
<code>is_integer/1</code>
<code>is_list/1</code>
<code>is_number/1</code>
<code>is_pid/1</code>
<code>is_port/1</code>
<code>is_reference/1</code>
<code>is_tuple/1</code>
<code>is_record/2</code>

Table 1.5: Type Test BIFs.

Note that each type test BIF has an older equivalent, without the `is_` prefix. These old BIFs are retained for backwards compatibility only and should not be used in new code. They are also only allowed at top level. For example, they are not allowed in boolean expressions in guards.

<code>abs(Number)</code>
<code>element(N, Tuple)</code>
<code>float(Term)</code>
<code>hd(List)</code>
<code>length(List)</code>
<code>node()</code>
<code>node(Pid Ref Port)</code>
<code>round(Number)</code>
<code>self()</code>
<code>size(Tuple Binary)</code>
<code>tl(List)</code>
<code>trunc(Number)</code>

Table 1.6: Other BIFs Allowed in Guard Expressions.

1.6.23 Operator Precedence

Operator precedence in falling priority:

- `:`
- `#`
- Unary `+` `-` `bnot` `not`

- `/ * div rem band and`
- `+ - bor bxor bsl bsr or xor`
- `++ -`
- `== /= =< < >= > := /=`
- `andalso`
- `orelse`
- `= !`
- `catch`

When evaluating an expression, the operator with the highest priority is evaluated first. Operators with the same priority are evaluated left to right. Example:

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

1.7 Macros

1.7.1 Defining and Using Macros

A macro is defined the following way:

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

A macro definition can be placed anywhere among the attributes and function declarations of a module, but the definition must come before any usage of the macro.

If a macro is used in several modules, it is recommended that the macro definition is placed in an include file.

A macro is used the following way:

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

Macros are expanded during compilation. A simple macro `?Const` will be replaced with `Replacement`. Example:

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

This will be expanded to:

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

A macro `?Func(Arg1, ..., ArgN)` will be replaced with `Replacement`, where all occurrences of a variable `Var` from the macro definition are replaced with the corresponding argument `Arg`. Example:

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

This will be expanded to:

```
bar(X) ->  
    {a,a,b,b},  
    {a,X,b,123}.
```

It is good programming practice, but not mandatory, to ensure that a macro definition is a valid Erlang syntactic form.

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`.

1.7.2 Predefined Macros

The following macros are predefined:

`?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'`.

1.7.3 Flow Control in Macros

The following macro directives are supplied:

`-undef(Macro)`. Causes the macro to behave as if it had never been defined.
`-ifdef(Macro)`. Evaluate the following lines only if `Macro` is defined.
`-ifndef(Macro)`. Evaluate the following lines only if `Macro` is not defined.
`-else`. Only allowed after an `ifdef` or `ifndef` directive. If that condition was false, the lines following `else` are evaluated instead.
`-endif`. Specifies the end of an `ifdef` or `ifndef` directive.

Example:

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

When trace output is desired, `debug` should be defined when the module `m` is compiled:

```
% erlc -Ddebug m.erl  
  
or  
  
1> c(m, {d, debug}).  
{ok,m}
```

`?LOG(Arg)` will then expand to a call to `io:format/2` and provide the user with some simple trace output.

1.7.4 Stringifying Macro Arguments

The construction `??Arg`, where `Arg` is a macro argument, will be expanded to a string containing the tokens of the argument. This is similar to the `#arg` stringifying construction in C.

The feature was added in Erlang 5.0/OTP R7.

Example:

```
-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.

1.8 Records

A record is a data structure for storing a fixed number of elements. It has named fields and is similar to a struct in C. Record expressions are translated to tuple expressions during compilation. Therefore, record expressions are not understood by the shell.

Read more about records in [Programming Examples](#)¹.

¹URL: [../examples/records](#)

1.8.1 Defining Records

A record definition consists of the name of the record, followed by the field names of the record. Record and field names must be atoms. Each field can be given an optional default value. If no default value is supplied, undefined will be used.

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

A record definition can be placed anywhere among the attributes and function declarations of a module, but the definition must come before any usage of the record.

If a record is used in several modules, it is recommended that the record definition is placed in an include file.

1.8.2 Creating Records

The following expression creates a new `Name` record where the value of each field `FieldI` is the value of evaluating the corresponding expression `ExprI`:

```
#Name{Field1=Expr1,...,FieldK=ExprK}
```

The fields may be in any order, not necessarily the same order as in the record definition, and fields can be omitted. Omitted fields will get their respective default value instead.

If several fields should be assigned the same value, the following construction can be used:

```
#Name{Field1=Expr1,...,FieldK=ExprK, _=ExprL}
```

Omitted fields will then get the value of evaluating `ExprL` instead of their default values. This feature was added in Erlang 5.1/OTP R8 and is primarily intended to be used to create patterns for ETS and Mnesia match functions. Example:

```
-record(person, {name, phone, address}).  
  
...  
  
lookup(Name, Tab) ->  
    ets:match_object(Tab, #person{name=Name, _='_'}).
```

1.8.3 Accessing Record Fields

`Expr#Name.Field`

Returns the value of the specified field. `Expr` should evaluate to a `Name` record.

The following expression returns the position of the specified field in the tuple representation of the record:

`#Name.Field`

Example:

```
-record(person, {name, phone, address}).
...

lookup(Name, List) ->
    lists:keysearch(Name, #person.name, List).
```

1.8.4 Updating Records

`Expr#Name{Field1=Expr1,...,FieldK=ExprK}`

`Expr` should evaluate to a `Name` record. Returns a copy of this record, with the value of each specified field `FieldI` changed to the value of evaluating the corresponding expression `ExprI`. All other fields retain their old values.

1.8.5 Records in Guards

Accessing record fields using `Expr#Name.Field` is allowed in guards. There is also a type test BIF `is_record(Var, Name)`. Example:

```
is_person(P) when record(P, person) ->
    true;
is_person(_P) ->
    false.
```

1.8.6 Records in Patterns

A pattern that will match a certain record is created the same way as a record is created:

`#Name{Field1=Expr1,...,FieldK=ExprK}`

In this case, one or more of `Expr1...ExprK` may be unbound variables.

1.8.7 Internal Representation of Records

Record expressions are translated to tuple expressions during compilation. A record defined as

```
-record(Name, {Field1,...,FieldN}).
```

is internally represented by the tuple

```
{Name,Value1,...,ValueN}
```

where each `ValueI` is the default value for `FieldI`.

To each module using records, a pseudo function is added during compilation to obtain information about records:

```
record_info(fields, Record) -> [Field]  
record_info(size, Record) -> Size
```

`Size` is the size of the tuple representation, that is one more than the number of fields.

1.9 Errors and Error Handling

1.9.1 Terminology

Errors can roughly be divided into three different types:

- Compile-time errors
- Logical errors
- Run-time errors

A compile-time error, for example a syntax error, should not cause much trouble as it is caught by the compiler.

A logical error is when a program does not behave as intended, but does not crash. An example could be that nothing happens when a button in a graphical user interface is clicked.

A run-time error is when a crash occurs. An example could be when an operator is applied to arguments of the wrong type. The Erlang programming language has built-in features for handling of run-time errors.

When a run-time error occurs in Erlang, execution for the process which evaluated the erroneous expression is stopped. This is referred to as a *failure*, that execution or evaluation *fails*, or that the process *fails*, *terminates* or *exits*. Note that a process may terminate/exit for other reasons than a failure.

A process that terminates will emit an *exit signal* with an *exit reason* that says something about which error has occurred. Normally, some information about the error will be printed to the terminal.

1.9.2 Handling of Run-Time Errors in Erlang

Error Handling Within Processes

It is possible to prevent run-time errors from causing the process to terminate by using `catch`, see the Expressions [page 25] chapter.

Error Handling Between Processes

Processes can monitor other processes and detect process terminations, see the Processes [page 37] chapter.

1.9.3 Exit Reasons

When a run-time error occurs, the exit reason is a tuple $\{\text{Reason}, \text{Stack}\}$. Reason is an atom indicating the type of error:

<i>Reason</i>	<i>Type of error</i>
<code>badarg</code>	Argument is of wrong type.
<code>badarith</code>	Argument is of wrong type in an arithmetic expression.
<code>badmatch</code>	Evaluation of a match expression failed.
<code>function_clause</code>	No matching function clause is found when evaluating a function call.
<code>case_clause</code>	No matching branch is found when evaluating a case expression.
<code>if_clause</code>	No true branch is found when evaluating an if expression.
<code>undef</code>	The function cannot be found when evaluating a function call.
<code>badfun</code>	There is something wrong with a fun.
<code>badarity</code>	A fun is applied to the wrong number of arguments.
<code>timeout_value</code>	The timeout value in a <code>receive..after</code> expression is evaluated to something else than an integer or infinity.
<code>noproc</code>	Trying to link to a non-existing process.
<code>nocatch</code>	Trying to evaluate a throw outside a catch.
<code>system_limit</code>	A system limit has been reached. See Efficiency Guide for information about system limits.

Table 1.7: Exit Reasons.

`Stack` is the stack of function calls being evaluated when the error occurred, given as a list of tuples $\{\text{Module}, \text{Name}, \text{Arity}\}$ with the most recent function call first.

1.10 Processes

1.10.1 Processes

This chapter is under construction.

1.10.2 Process Creation

A process is created by calling `spawn`:

```
spawn(Module, Name, Args) -> pid()
  Module = Name = atom()
  Args = [Arg1,...,ArgN]
  ArgI = term()
```

`spawn` creates a new process and returns the pid.

The new process will start executing in `Module:Name(Arg1,...,ArgN)` where the arguments is the elements of the (possible empty) `Args` argument list.

There exist a number of other spawn BIFs, for example `spawn/4` for spawning a process at another node. See `erlang(3)`.

1.10.3 Registered Processes

Besides addressing a process by using its pid, it are also BIFs for registering a process under a name. The name must be an atom and is automatically unregistered if the process terminates:

<code>register(Name, Pid)</code>	Associates the name <code>Name</code> , an atom, with the process <code>Pid</code> .
<code>registered()</code>	Returns a list of names which have been registered using <code>register/2</code> .
<code>whereis(Name)</code>	Returns the pid registered under <code>Name</code> , or <code>undefined</code> if the name is not registered.

Table 1.8: Name Registration BIFs.

1.10.4 Process Termination

When a process terminates, it always terminates with an `exit reason` set. The reason may be any term.

A process terminates *normally*, that is with exit reason `normal` when there is no more code to evaluate.

A process terminates with exit reason `{Reason, Stack}` when a run-time error occurs. See Error and Error Handling [page 35].

A process can terminate itself by calling one of the BIFs `exit(Reason)` or `fault(Reason)`. The process then terminates with reason `Reason` or `{Reason, Stack}`, respectively.

A process may also be terminated if it receives an *exit signal* with another reason than `normal`, see Error Handling [page 37] below.

1.10.5 Message Sending

Processes communicate by sending and receiving messages. Messages are sent by using the `send` operator `!` [page 19] and received by calling `receive` [page 19].

Message sending is asynchronous and safe, the message is guaranteed to eventually reach the recipient, provided that the recipient exists.

1.10.6 Error Handling

Erlang has a built-in feature for error handling between processes. Terminating processes will emit exit signals to all linked processes, which may terminate as well or handle the exit 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.

Refer to OTP Design Principles for more information about OTP supervision trees, which uses this feature.

Emitting Exit Signals

When a process terminates, it will terminate with an *exit reason* as explained in Process Termination [page 36] above. This exit reason is emitted in an *exit signal* to all linked processes.

A process can also call the function `exit(Pid, Reason)`. This will result in an exit signal with exit reason `Reason` being emitted to `Pid`, but does not affect the calling process.

Receiving Exit Signals

The default behaviour when a process receives an exit signal with an exit reason other than `normal`, is to terminate and in turn emit exit signals with the same exit reason to its linked processes. An exit signal with reason `normal` is ignored.

A process can be set to trap exit signals by calling:

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

When a process is trapping exits, it will not terminate when an exit signal is received. Instead, the signal is transformed into a message `{'EXIT', FromPid, Reason}` which is put into the mailbox of the process just like a regular message.

An exception to the above is if the exit reason is `kill`, that is if `exit(Pid, kill)` has been killed. This will unconditionally terminate the process, regardless of if it is trapping exit signals or not.

1.10.7 Monitoring Processes

Processes can monitor another process by calling `erlang:monitor/2`. Monitoring is unidirectional. If the monitored process terminates, the monitoring process will receive a 'DOWN' message.

1.10.8 Process Dictionary

Each process has its own process dictionary, accessed by calling the following BIFs:

```
put(Key, Value)
get(Key)
get()
get_keys(Value)
erase(Key)
erase()
```

1.11 Distributed Erlang

An Erlang runtime system is made into a distributed Erlang *node* by giving it a name. Nodes can be connected to each other and a process at one node can monitor other nodes. It is also possible to spawn processes at other nodes. Message passing and error handling between processes at different nodes is the same as between processes at the same node.

There exists a number of modules useful for a distributed Erlang system. One example is `global` which provides global name registration.

The distribution mechanism is implemented using TCP/IP sockets. How to implement an alternative carrier is described in ERTS User's Guide.

1.11.1 Distribution BIFs

Examples of BIFs used for distributed programming, see `erlang(3)` for more information:

<code>erlang:disconnect_node(Node)</code>	Forces the disconnection of a node.
<code>erlang:get_cookie()</code>	Returns the magic cookie of the current node.
<code>is_alive()</code>	Returns <code>true</code> if the runtime system is a node and can connect to other nodes, <code>false</code> otherwise.
<code>monitor_node(Node, true false)</code>	Monitor the status of <code>Node</code> . A message <code>{nodedown, Node}</code> is received if the connection to it is lost.
<code>node()</code>	Returns the name of the current node. Allowed in guards.
<code>node(Arg)</code>	Returns the node where <code>Arg</code> , a pid, reference, or port, is located.
<code>nodes()</code>	Returns a list of all visible nodes this node is connected to.
<code>nodes(Arg)</code>	Depending on <code>Arg</code> , this function can return a list not only of visible nodes, but also hidden nodes and previously known nodes, etc.
<code>set_cookie(Node, Cookie)</code>	Sets the magic cookie used when connecting to <code>Node</code> . If <code>Node</code> is the current node, <code>Cookie</code> will be used when connecting to all new nodes.
<code>spawn[_link _opt](Node, Fun)</code>	Creates a process at a remote node.
<code>spawn[_link opt](Node, Module, FunctionName, Args)</code>	Creates a process at a remote node.

Table 1.9: Distribution BIFs.

1.11.2 Distribution Command Line Flags

Examples of command line flags used for distributed programming, see `erl(1)` for more information:

<code>-connect_all false</code>	Only explicit connection set-ups will be used.
<code>-hidden</code>	Makes a node into a hidden node.
<code>-name Name</code>	Makes a runtime system into a node, using long node names.
<code>-setcookie Cookie</code>	Same as calling <code>erlang:set_cookie(node(), Cookie)</code> .
<code>-sname Name</code>	Makes a runtime system into a node, using short node names.

Table 1.10: Distribution Command Line Flags.

1.11.3 Distribution Modules

Examples of modules useful for distributed programming:

In Kernel:

<code>global</code>	A global name registration facility.
<code>global_group</code>	Grouping nodes to global name registration groups.
<code>net_adm</code>	Various Erlang net administration routines.
<code>net_kernel</code>	Erlang networking kernel.

Table 1.11: Kernel Modules Useful For Distribution.

In STDLIB:

<code>slave</code>	Start and control of slave nodes.
--------------------	-----------------------------------

Table 1.12: STDLIB Modules Useful For Distribution.

1.11.4 Nodes

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

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. Example:

```
% erl -name dilbert
(dilbert@uab.ericsson.se)1> node().
'dilbert@uab.ericsson.se'
```

```
% erl -sname dilbert
(dilbert@uab)1> node().
dilbert@uab
```

Note:

A node with a long node name cannot communicate with a node with a short node name.

1.11.5 Node Connections

The nodes in a distributed Erlang system are loosely connected. The first time the name of another node is used, for example if `spawn(Node,M,F,A)` or `net_adm:ping(Node)` is called, a connection attempt to that node will be made.

Connections are by default transitive. 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 by using the command line flag `-connect_all false`, see `erl(1)`.

If a node goes down, all connections to that node are removed. Calling `erlang:disconnect(Node)` will force disconnection of a node.

The list of (visible) nodes currently connected to is returned by `nodes()`.

1.11.6 Hidden Nodes

In a distributed Erlang system, it is sometimes useful to connect to a node without also connecting to all other nodes. An example could be some kind of O&M functionality used to inspect the status of a system without disturbing it. 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 are not transitive, they must be set up explicitly. Also, hidden nodes does not show up in the list of nodes returned by `nodes()`. Instead, `nodes(hidden)` or `nodes(connected)` must be used. This means, for example, that the hidden node will not be added to the set of nodes that `global` is keeping track of.

This feature was added in Erlang 5.0/OTP R7.

1.11.7 epmd

The Erlang Port Mapper Daemon *epmd* is automatically started at every host where an Erlang node is started. It is responsible for mapping the symbolic node names to machine addresses. See `epmd(1)`.

1.11.8 Security

Nodes are protected by a *magic cookie* system. When a connection attempt is made from a node A to a node B, node A provide the cookie that has been set for node B. If the magic cookie is not correct, the connection attempt is rejected.

If the node is started with the command line flag `-setcookie Cookie`, then the cookie will be assumed to be the atom `Cookie` for all other nodes.

If the node is started without the command line flag, then the node will create the cookie from the contents of the file `$HOME/.erlang.cookie`, where `$HOME` is the user's home directory. If the file does not exist, it will be created and contain a random character sequence.

Which magic cookie to use when connecting to another node can be changed by calling `erlang:set_cookie/2`.

1.12 Compilation and Code Loading

How code is compiled and loaded is not a language issue, but is system dependent. This chapter describes compilation and code loading in Erlang/OTP with pointers to relevant parts of the documentation.

1.12.1 Compilation

Erlang programs must be *compiled* to object code. The compiler can generate a new file which contains the object code. The current abstract machine which runs the object code is called BEAM, therefore the object files get the suffix `.beam`. The compiler can also generate a binary which can be loaded directly.

The compiler is located in the Kernel module `compile`, see `compile(3)`.

```
compile:file(Module)
compile:file(Module, Options)
```

The Erlang shell understands the command `c(Module)` which both compiles and loads `Module`.

There is also a module `make` which provides a set of functions similar to the UNIX type Make functions, see `make(3)`.

The compiler can also be accessed from the OS prompt, see `erl(1)`.

```
% erl -compile Module1...ModuleN
% erl -make
```

The `erlc` program provides an even better way to compile modules from the shell, see `erlc(1)`. It understands a number of flags that can be used to define macros, add search paths for include files, and more.

```
% erlc <flags> File1.erl...FileN.erl
```

1.12.2 Code Loading

The object code must be *loaded* into the Erlang runtime system. This is handled by the *code server*, see `code(3)`.

The runtime system can be started in either *embedded* or *interactive* mode. Which one is determined by the command line flag `-mode`. Default mode is *interactive*.

In *embedded* mode, all code is loaded during system start-up according to the boot file, see *OTP System Principles* and `script(4)`.

In *interactive* mode, the code server dynamically loads modules into the system on demand, which means the first time a function in a module is called. Initially, the search path consists of the current working directory and all object code directories under `$ROOT/lib`, where `$ROOT` is the installation directory of Erlang/OTP. The module `code` contains functions for modifying the search path.

1.12.3 Code Replacement

Erlang supports change of code in a running system. Code replacement is done on module level.

The code of a module can exist in two variants in a system: *current* and *old*. When a module is loaded into the system for the first time, the code becomes 'current'. If then a new instance of the module is loaded, the code of the previous instance becomes 'old' and the new instance becomes 'current'.

Both old and current code is valid, and may be evaluated concurrently. Fully qualified function calls always refer to current code. Old code may still be evaluated because of processes lingering in the old code.

If a third instance of the module is loaded, the code server will remove (purge) the old code and any processes lingering in it will be 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. Example:

```
-module(m).  
-export([loop/0]).  
  
loop() ->  
    receive  
        code_switch ->  
            m: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 `m:loop()` and change to current code. Note that `m:loop/0` must be exported.

List of Tables

1.1	Recognized Escape Sequences.	8
1.2	Term Comparison Operators.	21
1.3	Arithmetic Operators.	22
1.4	Logical Operators.	23
1.5	Type Test BIFs.	28
1.6	Other BIFs Allowed in Guard Expressions.	28
1.7	Exit Reasons.	35
1.8	Name Registration BIFs.	36
1.9	Distribution BIFs.	38
1.10	Distribution Command Line Flags.	39
1.11	Kernel Modules Useful For Distribution.	39
1.12	STDLIB Modules Useful For Distribution.	39