Erlang 4.7.3
Reference Manual

DRAFT (0.7)

Jonas Barklund
(barklund@hotmail.com)

Robert Virding
(rv@cslab.ericsson.se)

February 9, 1999

Contents

“Release notes” for draft 0.7

This draft has the following important differences from draft 0.6:

- Cleaned up meaning of ERLANG.
- Removed STANDARD ERLANG lexical rules from lexical summary. (Used Std test in ALL STANDARD rules)
- Switch expression of case fixed.
- LineTerminator is only LF in ERLANG 4.7.3. Warn about using whitespace after `$`.
- Fix recognizer BIFs in ERLANG 4.7.3 to be only guard recognizers.
- Re-did macro definition and application syntax. Hopefully correct.
- All numeric literals (Integer, Decimal, ExplicitRadix and Float) are now unsigned in the lexical structure.
- Made the logical operators have the same priorities as the additive and multiplicative operators, as they should.
- Atomic literals now defined in the expression chapter as this is where values are explained.

“Release notes” for draft 0.6

This draft has the following important differences from draft 0.5:

- There is now an index (laboriously compiled).
- There is a neat summary of all ERLANG expressions in §A.
- Function application is now described.
- Records are now described.
- Ports are now fully (?) described.
- Dynamic code loading is now fully (?) described.
- The external term format is now described.
- The portable (?) hash function is now described.
- Parse trees are now described.
- Added `try` *`E`* `end` to be almost equivalent with `catch` *`E`*.
- The description of atoms has changed (§4.2).
- All BIFs of ERLANG 4.7.3 are described (§13), except two: `set_node/2` and `set_node/3`.
- Exact equality and arithmetic equality (formerly called coerced equality) are now well-defined, I think (§4.11.1).
- The section about Unicode escapes is now written.
- Lowered minimum *maxint* to $2^{59}-1$ (it was $2^{63}-1$) so that on a 64-bit machine an implementation without bignums can store all numbers as scalars with four bits for a tag etc.
- Operators `//` and `mod` have been added (computing the same as `div` and `rem` except that they round towards negative infinity rather than zero).

- Extent of function calls and last call optimization has been defined (§6.7.3).
- Dropped type and rule declarations completely from the specification.
- Scheduling and waiting in `receive` expressions is described more precisely.

I am considering:

- Moving §6.6.3 to an appendix.

Note:

- In the version for STANDARD ERLANG, I have assumed that the proposal about new BIFs will be accepted, hence there are quite a lot of references to such BIFs. However, they are not described yet, hence there are also quite a lot of unresolved cross-references in the STANDARD ERLANG version. If the BIF proposal is not accepted, then I will change to use the current BIFs also in that version, of course.

The following is what remains:

- Checking release notes for R4.
- The BIFs `set_node/2` (§13.10.11) and `set_node/3` (§13.10.12).
- Adding appendix about EPMD and ERLANG to ERLANG node communication.
- If remaining proposals are accepted, incorporating them (the new BIFs is a hog, the rest is fairly easy).
- Checking that monitoring of nodes is described properly.

PLEASE, DRAFT READERS: THERE ARE PLACES IN THE TEXT WHERE I REQUEST INFORMATION, ESPECIALLY ABOUT VARIOUS PARAMETERS OF ERLANG 4.7. IF YOU KNOW THESE THINGS, EMAIL ME!

“Release notes” for draft 0.5

This draft has the following important differences from draft 0.4:

- It has been added to the introduction how the specification relates to the ERLANG language and the implementations ERLANG 4.7.3 and ERLANG 5.0.
- Some notions have been added to the glossary: expression, operator, macro and term.
- Macro `?VERSION` added. Comments?
- The description of equality tests is not quite accurate, will rewrite.
- The evaluation order has been updated to be strictly left-to-right.
- The section about guards (§6.20) has been rewritten on considerably simpler form.
- The directionality when relating input and output environments has been made more clear.
- It should now be even clearer that clauses are tried left to right and that in `receive` expressions, all clauses are tried for a term in the message queue before the next term is tried.
- The terminology for abrupt completion has been made more uniform.
- New chapter (§5) about arithmetics (LIA-1 based).
- New (but not yet complete) chapter (§12) about ports.
- The format for BIF descriptions has evolved and many BIFs described.
- The section about scheduling (§10.7) has been improved.
- New (short) sections about lifetime of terms (§4.12) and about memory management (§4.13).

Some things that are planned to be improved (and thus do not need to be commented upon in the 0.5 draft if you agree):

- I will try to reduce the number of forward references further.

Known unclear issues:

- How should shadowing of BIFs work for those that are guard BIFs?

I have not gone through all comments from Torbjörn Keisu. They should be incorporated in release 0.5.1.

"Release notes" for draft 0.4

This draft has the following important differences from draft 0.3.1:

- '`.`' and '`?`' are now listed as separators.
- Full stops are detected properly (before white space and comments have been thrown away).
- The structure of a module is now described as consisting of
  - First a module declaration `-module(`*`ModuleName`*`)`.
  - Then any *header forms*, i.e., `export`, `import` and `compile` attributes, together with `type` and `rule` declarations.
  - Then the *program forms*, i.e., function declarations mixed with `file` and wild attributes, record declarations and macro definitions.
- The compilation of ERLANG programs is now described in detail in §7.
- A *list* is now either `[]` or a cons in which the right part is a list.
- The description of coercion has been improved (§4.10.1).

I have taken (most of) the written comments on 0.3 and/or 0.3.1 by HÂkan and Per Mildner into account.

Some things that are planned to be improved (and thus do not need to be commented upon in the 0.4 draft if you agree):

- Express better the directionality when relating input and output environments of expressions.
- Express arithmetics in terms of LIA-1.
- Change the writing so that becomes clear that clauses (in `case` etc) are tried left to right.
- Make the writing for `receive` expressions *even* clearer concerning the fact that for each message, each clause is tried before the next message is considered.

- Use a more consistent terminology for abrupt completion and its causes.
- The BIF Chapter (§13) is both incomplete and requiring changes.
- Assuming that the revised proposal about evaluation order is accepted, the evaluation order will be changed to strict left-to-right everywere.

“Release notes” for draft 0.3.1

This draft has the following important differences from draft 0.3:

- Various typos have been fixed (in particular the ones in §6.10 and §6.11 about match and send expressions and in §8.5.2 about `module_info/1` for `imports`).
- The operator `and` is no longer also listed as a keyword (§3.8).
- The proposed new escape sequence `\s` (for space) is described (§3.10).
- The blurb about process communcation has been removed from the introduction of binaries (§4.5).
- Records are now in the term order (between tuples and lists) described in §4.11.2.
- The `unregistered_name` error handler has been removed (§6.11), on request from Klacke and Tony.
- The description of the list difference operator `--` has been fixed so now it may be correct (§6.13.2).
- The description of `receive` expressions has been somewhat adjusted to accord better with reality and with the intentions (§6.19.9).
- The descriptions of how the clauses in `if`, `case`, `receive`, `try` and `fun` expressions are checked have been improved to increase clarity (§6.19.7, §6.19.8, §6.19.9, §6.19.10).
- The description of `receive` expressions has been significantly improved and may be correct (§6.19.10).
- The missing Section 6.20 about guards is no longer missing.

- §10 has been almost completely rewritten and uses a new concept of *signals* that comprises messages and exit signals, as well as link/unlink requests, etc. (This is actually how it is implemented in the '5.0' system and the presentation becomes more precise.)
- The priority `high` has been added (§10).

I have taken the written comments on 0.3 by Klacke and Tony into account. The comments by Per Mildner remain (I forgot to bring them to Stockholm).

Chapter 1

Introduction

This document is a specification of the ERLANG implementation called ERLANG 4.7.3, developed at Ericsson Telecom AB.

ERLANG was originally designed by Joe Armstrong, Robert Virding, Claes Wikström and Mike Williams at the Computer Science Laboratory of Ericsson Telecommunications Systems Laboratories.

This specification is primarily designed to be useful for ERLANG programmers and implementors of ERLANG by providing clear although mostly informally expressed semantics for all language constructs. It should be of use also for those developing analysis tools for ERLANG although for these purposes the semantics may have to be reformulated as a formal system.

The specification should thus be able to function well as a reference manual for the language. As such it should be a good companion to the book *Concurrent Programming in ERLANG, Second Edition* [3], which is more of a tutorial, text book and evangel than a language reference. That edition of the book, however, uses an earlier version of the language (the implementation ERLANG 4.3).

It is also intended that the specification should be useful as a basis for a future international standardization of the language.

Chapter 2

Notation and glossary

In this chapter we define the notation used in the remainder of the specificaition, including that used for grammars. Notation that is mostly local to a single chapter is described in that chapter. There is also a glossary that explains various terms that are used in the specification (some of them specific to ERLANG*, some of them not).*

2.1 Symbols

We use typewriter style for ERLANG tokens, e.g., '`foo(X) -> bar(X)`'. The symbol '`X`' is thus an ERLANG variable.

We use slanted typewriter style for metavariables, e.g., for the '*`T`*' in '`throw(`*`T`*`)`'. A metavariable always consists of letters but may have an index as in 'E_2' or 'v_1'. A metavariable stands for some unspecified sequence of ERLANG tokens. The letter (and case of the letter) chosen for the main symbol of the metavariable is an indication of the token sequences over which the metavariable usually ranges (but the actual range is always given explicitly in the text).

- An '*`E`*' indicates an arbitrary ERLANG expression.
- A '*`T`*', '*`t`*', '*`v`*' or '*`w`*' indicates an ERLANG term. (The two latter symbols are typically used for terms obtained as the value of some expression, rather than a term occurring as part of the program text.)
- A '*`V`*' indicates an ERLANG variable.
- An '*`A`*' indicates an ERLANG atom.
- A '*`B`*' indicates an ERLANG Boolean atom or an ERLANG binary.
- An '*`O`*' indicates an ERLANG operator.
- An '*`I`*' indicates an ERLANG integer numeral.

- An '`F`' indicates an ERLANG floating point numeral.
- A '`P`' indicates an arbitrary ERLANG pattern in some contexts and an ERLANG PID in others.
- An '`R`' indicates an ERLANG port or an ERLANG ref.
- An '`M`' indicates an ERLANG module (i.e., a binary that is the representation of a module).

We use italic letters in non-ERLANG expressions to stand for numbers or ERLANG terms.

2.2 Sets

In set expressions, $x \cup y$ and $x \cap y$ mean the union and intersection of the sets x and y, respectively. $\overline{x}$ means the complement of a set x. $x \supseteq y$ means that x contains every element of y.

2.3 Mappings

A *mapping* is a set of pairs such that each pair has a distinct left half. We write such a pair of v and t as $v \mapsto t$. A mapping ϵ can be *applied* to a value v, which is written $\epsilon(v)$. If there is a pair $v \mapsto t$ in ϵ, then the application denotes t. Otherwise the denotation is undefined.

We define the *domain* of a mapping ϵ to be the set of values occurring in the left half of a pair in ϵ and denote it by $\mathrm{dom}(\epsilon)$.

The *restriction* of a mapping ϵ to a set of values d means the mapping which is the largest subset of ϵ such that its domain is contained in d and is written $\epsilon|d$.

We say that that a mapping ϵ' *extends* a mapping ϵ if $\mathrm{dom}(\epsilon') \supseteq \mathrm{dom}(\epsilon)$ and for every value v in $\mathrm{dom}(\epsilon)$, $\epsilon'(v) = \epsilon(v)$. That is, ϵ is a subset of ϵ'. Note that the extension can be trivial — ϵ extends ϵ.

If ϵ' extends ϵ then the *difference* between ϵ' and ϵ is the mapping consisting of the pairs that occur in ϵ' but not in ϵ and is denoted by $\epsilon' \setminus \epsilon$. In other words, $\epsilon' \setminus \epsilon = \epsilon'|\overline{\mathrm{dom}(\epsilon)}$.

With the usual set notation we may denote a finite mapping ϵ with domain $\{v_1, \ldots, v_n\}$ by $\{v_1 \mapsto \epsilon(v_1), \ldots, v_n \mapsto \epsilon(v_n)\}$. (All mappings that will be discussed in this specification are finite.)

Let ϵ and δ be mappings. The result of *extending* ϵ with δ is a mapping that contains all pairs of δ and those pairs of ϵ having left halves not in the domain of δ and we denote it by $\epsilon \oplus \delta$. In other words, $\epsilon \oplus \delta = \epsilon|\overline{\mathrm{dom}(\delta)} \cup \delta$.

We say that an mapping is *defined* for some value if the value is in the domain of that mapping.

An *empty* mapping has an empty set as its domain.

Most of this terminology is introduced for describing environments in the evaluation of expressions (§6).

2.4 Tables

A *table* is an object with a state representing a mapping between ERLANG terms. We use the table metaphor because it is suggestive to use a terminology of adding and removing rows.

A table consists of a finite number of rows where each row contains a *key* and a *value*, both of which are typically ERLANG terms. The keys of all rows in a table are distinct. An empty table has no rows. If t is a table having a row with key k, then we may write $t(k)$ for the value of that row.

Suppose that a table at a particular time contains n rows having the keys $k_1, \ldots, k_n$ and values $v_1, \ldots, v_n$. An association list (§4.9.2) representing the contents of the table contains n 2-tuples $\{k_i, v_i\}$, $1 \leq i \leq n$, in some order. A list representing the keys of the table contains n terms k_i, $1 \leq i \leq n$, in some order.

2.5 Types

When describing BIFs (§13) we include information about their types. This information imposes constraints on which should be the types of the evaluated arguments and what the type of the result can be. The *signature* of a BIF is written in accordance with that used for the experimental type checker for ERLANG [17, 18]. The type expressions listed in Table 2.1 are used in this description: The type of a function *F/n* is expressed on the form

$F(T_{1,1},\ldots,T_{1,n})$ `-> T_1 ;`
$F(T_{2,1},\ldots,T_{2,n})$ `-> T_2 ;`
...
$F(T_{k,1},\ldots,T_{k,n})$ `-> T_k`

and says that

- If *F/n* is applied to n arguments that are of the types $T_{1,1}, \ldots, T_{1,n}$, then the result is of type T_1.
- Otherwise, if ...
- Otherwise, if *F/n* is applied to n arguments that are of the types $T_{k,1}, \ldots, T_{k,n}$, then the result is of type T_k.

This should cover all valid combinations of argument types for *F/n*.

Type expression	Type
AtomicLiteral	the denoted term
`atom()`	atom
`bool()`	Boolean
`char()`	character
`int()`	integer
`float()`	float
`num()`	integer or float
`ref()`	ref
`bin()`	binary
`pid()`	PID
`port()`	port
`function()`	function
`tuple()`	tuple
`{`T_1`,...,`T_k`}`	k-tuple with elements of types T_1, ... , T_k
`[`T`]`	list with elements of type T
`string()`	string
`term()`	any ERLANG term

Table 2.1: Types in ERLANG.

2.6 Grammar

2.6.1 Grammar notation

The purpose of a grammar is to define sets of well-formed sequences of terminals, called *syntactic categories*. Some examples of syntactic categories in this specification are *AtomicLiteral* and *Expr*, which consist of the atomic literals and the expressions of ERLANG, respectively.

The *terminals* can be, e.g., individual characters of some alphabet, or elements of some other set of tokens. The set of tokens may be infinite but it must be possible to partition them into a finite number of categories. The grammar will only be concerned with the category to which a token belongs.

We use a standard grammar formalism as explained, e.g., by Aho & Ullman [1] (although we use a somewhat different syntax than in their exposition). A grammar consists of a set of *productions*. Each production consists of a *head*, which is a syntactic category, and a *body*, which is a finite (and possibly empty) sequence of terminals and syntactic categories.

A production is written as in

RecordField:
    AtomLiteral = *Expr*

where *RecordField* is the head and *AtomLiteral* = *Expr* is the body consisting of a syntactic category *AtomLiteral*, a terminal `=` and a syntactic category *Expr*.

We use two shorthands for productions. First, we may write a production having one head but k bodies, where $k > 1$, with each body on a separate line. This is shorthand for k productions, all with the same head. For example, we write the single production

CompareExpr:
  ListConcExpr RelationalOp ListConcExpr
  ListConcExpr EqualityOp ListConcExpr
  ListConcExpr

instead of the three productions

CompareExpr:
  ListConcExpr RelationalOp ListConcExpr

CompareExpr:
  ListConcExpr EqualityOp ListConcExpr

CompareExpr:
  ListConcExpr

Second, we may attach a subscript *opt* (for 'optional') to a body element (as a matter of fact we only do so for elements that are syntactic categories). A production having such an annotation in the body is shorthand for two productions, one where the body element is present (but without the subscript) and one where it is not. For example, the production

TupleSkeleton:
  `{` $Exprs_{opt}$ `}`

is short for the two productions

TupleSkeleton:
  `{` *Exprs* `}`

TupleSkeleton:
  `{ }`

If more than one body element of a production has an *opt* subscript, then this expansion can be repeated and will produce 2^k productions, where k is the number of subscripted elements. For example, the production

ModuleDeclaration:
  ModuleAttribute $HeaderForms_{opt}$ $ProgramForms_{opt}$

is short for the two productions

ModuleDeclaration:
  ModuleAttribute HeaderForms ProgramForms$_{opt}$

ModuleDeclaration:
  ModuleAttribute ProgramForms$_{opt}$

which are short for the four productions

ModuleDeclaration:
  ModuleAttribute HeaderForms ProgramForms

ModuleDeclaration:
  ModuleAttribute HeaderForms

ModuleDeclaration:
  ModuleAttribute ProgramForms

ModuleDeclaration:
  ModuleAttribute

In §3.11 an additional ad-hoc shorthand is used where a single production

Digit[*k*]: one of the first *k* of
  `0 1 2 3 4 5 6 7 8 9 Aa Bb Cc Dd Ee Ff`

is used to summarize sixteen productions on the form

Digit2:
  `0`
  `1`

⋮

Digit16:
  `0`
  `1`
  ⋮
  `E`
  `e`
  `F`
  `f`

2.6.2 The lexical grammar

The productions in in §3 can be viewed as constituting a grammar on their own. That grammar has individual ASCII characters as terminals. It defines two syntactic categories *Token* (§3.4) and *FullStop* (§3.18) from which the terminals of the main grammar (§2.6.3) are drawn.

The subgrammar for *Token* actually consists of regular expressions and can therefore be coded in a system such as the `lex` utility program, which translates the grammar to a finite automaton that can be implemented very efficiently.

The lexical grammar is summarized in §E.1.

2.6.3 The main grammar

In the main grammar presented in §6 and §8, the terminals are the syntactic categories of the lexical grammar and anything in typewriter style (all of which belong to *Token* as described in §3).

That grammar is almost, but not quite, a LALR(1) grammar. The problem is that match expressions (§6.10) and generators (§6.19.5) begin with a pattern, which is typically indistinguishable from an expression using a lookahead of only one token. One way to make the grammar a LALR(1) grammar is to change the productions

MatchExpr:
    Pattern `=` *SendExpr*
    ⋮

Generator:
    Pattern `<-` *Expr*

to

MatchExpr:
    ApplicationExpr `=` *SendExpr*
    ⋮

Generator:
    ApplicationExpr `<-` *Expr*

and add *UniversalPattern* as a *PrimaryExpr*.

Then it must be verified after parsing that the left-hand operand of `=` and `<-` are indeed patterns and that no *UniversalPattern* appears in an expression.

(Note that this problem would not have appeared if the syntax of match expressions and generators had included a keyword before the pattern.)

The main grammar is summarized in §E.2.

2.7 Glossary

The purpose of this section is to explain some terms that are used throughout the specification and either do not have a natural point of definition or that need to be used before their point of definition. In the latter case, this glossary is intended to give a summarical explanation that will be sufficient until the full description is reached. Most concepts explained here are thus described in more detail elsewhere. When a word in an explanation is in bold face, it is explained separately.

Abrupt completion
The evaluation of an expression completes abruptly either because a problem has been encountered that makes it impossible or meaningless to continue evaluation (cf. **exit**), or because it has been requested to **throw** a value. Abrupt completion always has an associated reason which is an ERLANG term. Abrupt completion can be trapped if it occurs in the body of a `catch` expression (§6.9) and the reason can then be accessed. Abrupt completion for expressions is described in more detail in §6.4.

A process may also complete abruptly, e.g., because the evaluation of its original function application completes abruptly or it receives an untrapped exit signal. Abrupt completion for processes is described in more detail in §10.4.1.

Application
We use this word with two overloaded meanings. We can mean a function application, i.e., the application $F(E_1, \ldots, E_k)$ of an ERLANG function F to a sequence of **arguments** $E_1, \ldots, E_k$, or we can mean an application in the sense of **OTP**, i.e., a collection of related modules.

Argument
A function **application** expression has two main parts: a function to be applied and a sequence of *arguments*, or actual parameters, of the function application. The arguments are evaluated before the function call begins, so the called function can only observe the values of the arguments.

Arity The *arity* of a function is the number of arguments to which it expects should be applied. Each function has a specific arity (so the clauses defining it must expect the same number of arguments) but there may be functions having the same **function symbol** but different arities.

ASCII
ASCII is the popular acronym for the 7-bit code for representation of characters properly called ANSI X3.4 [2].

BIF A BIF is a built-in function of ERLANG.

Being built-in does not imply any particular form of implementation: a BIF could be implemented, for example, through a virtual machine instruction, a procedure in another language, or an Erlang function. A BIF must not be redefined during the lifetime of a node so the Erlang compiler and loader are permitted to use all information in the description of the BIF to make execution efficient.

Some BIFs have unqualified names (e.g., `length/1`) while others must be referred to using a qualified name (e.g., `lists:map/2`) unless they have been explicitly imported (§8.2.2).

The operators of Erlang (§3.9) are not BIFs, but with the exception of match expressions (with the `=` operator), binary operator expressions behave exactly like applications of binary functions.

Erlang constructs such as `if` expressions, `catch` expressions, etc., are not even functions and thus not BIFs.

The BIFs of Erlang are described in detail in §13.

Big-endian

When a sequence of **bytes** $b_1, \dots, b_k$ is interpreted as a *big-endian* numeral, the significance decreases monotonically. If the numeral has base 256 and is interpreted as unsigned, its value is

$$256(256(\dots 256b_1 + b_2 \dots) + b_{k-1}) + b_k = \sum_{i=1}^{k} 256^{k-i} b_i,$$

and we denote this by $\mathit{BigEndianValue}(\langle b_1, \dots, b_k \rangle)$.

For example, the four bytes `22 188 72 209` when interpreted as a big-endian numeral represent 381,438,161 because $256^3 \cdot 22 + 256^2 \cdot 188 + 256^1 \cdot 72 + 256^0 \cdot 209 = 16777216 \cdot 22 + 65536 \cdot 188 + 256 \cdot 72 + 1 \cdot 209 = 369098752 + 12320768 + 18432 + 209 = 381438161$.

If the numeral is interpreted as signed, then its value is

$$(\mathit{BigEndianValue}(\langle b_1, \dots, b_k) + 2^{8k-1}\rangle) \bmod 2^{8k} - 2^{8k-1}$$

and we denote this by $\mathit{BigEndianSignedValue}(\langle b_1, \dots, b_k \rangle)$.

Bignum

A bignum is an Erlang integer that is not a fixnum. Bignums are represented in such a way that integers with very large magnitudes can be represented. However, arithmetic operations on bignums, even comparatively small ones, should be expected to be much more expensive than the corresponding operations on **fixnums**. §4.5.

Binary

When used as a noun, a binary is an Erlang term that represents a finite sequence of **bytes**. Binaries are described in more detail in §4.5.

Body A body (syntactic category *Body*) is a nonempty sequence of expressions. Evaluating a body means to evaluate the constituent expressions in order. The value of the body is the value of the last expression.

Byte A byte is an integer in the range $[0, 255]$.

Call To *call* a **function** (e.g., a **BIF**) means to evaluate an application of the function. However, we often mean only the part of the evaluation that follows argument evaluation.

Clause

In expressions where there is a choice between several alternatives, each alternative is specified through a clause. Every clause has a **body** that will be evaluated if the clause is chosen. What the other parts are depends on the surrounding expression. In function declarations and in `fun`, `case`, `receive` and `try` expressions, each clause has a **pattern** and a corresponding **guard**. In `if` expressions, the clauses have only a **guard**.

Compile time

When we write that something is carried out at compile time or that a compile time error should be raised, we mean that it should happen as part of the process of transforming an Erlang module definition into a loadable binary.

However, one must also consider compilers that are applied to source code that has already been loaded, e.g., in order to be used in an interpreter. In that case, enough processing must have taken place as the source code was loaded so that all errors that should be detected at compile-time according to this specification were detected at load time.

Compilation of Erlang modules is described in more detail in §7.

Context

A context is a set of variables. Each expression that occurs as part of an Erlang program (i.e., a module declaration) has an *input context*, which is the set of variables that will have bindings at **run time**, i.e., when the expression is evaluated. (In other words: the input context is the domain of the **environment** in which the expression will be evaluated.) It also has an *output context*, which is the input context extended with any variables for which the expression will provide bindings. Input and output contexts are described in more detail in §6.2.

Effect Evaluation of an ERLANG expression by an ERLANG process may produce an *effect*, regardless of whether it completes normally or abruptly. An effect is one of:

- changing the state of a **process**;
- changing the state of a **port**;
- changing the state of a **node**.

Typical effects are sending a **message** (which adds to the message queue of some process) or receiving a message (which removes from the message queue of the own process).

Effects that are observable externally are those that change the state of a port, and in some cases those that change the state of a node. A typical example of such an effect is sending a message to a port in order to produce output.

Environment

An expression is evaluated in an environment, which is a mapping from variables to terms. Environments are described in more detail in §6.2.

Erlang

In this specification, the unqualified name ERLANG refers to ERLANG 4.7.3. ERLANG 4.7.3 may be written explicitly when referring to different behaviour of different versions.

Error A **compile-time** error is a violation of a syntactic or semantic rule that governs ERLANG programs and that can be verified in the process of compiling an ERLANG module (§7). If a compile-time error occurred while compiling a module, the result of loading or using it is undefined.

A **run-time** error is a violation of a required condition in the semantic rules that govern ERLANG programs and that is verified during evaluation. If the description states that the evaluation should **exit** with some reason, then this behaviour is required. (The exit reason is then a term describing the error.) If a stated precondition is violated and the description does not state an exit reason, then computation may proceed but the result and effects are undefined.

If a process completes abruptly due to an error that was encountered while evaluating its function application, the **exit signal** provides information about the error.

Exit When we say that the evaluation of an expression *exits with reason* `T`, this is short for saying that the evaluation of the expression completes abruptly with an associated reason `{'EXIT',T}`.

An exit indicates that an exceptional situation such as a **run-time error** has occurred.

Exit signal
An exit signal is an ERLANG term communicated from one process to another, typically to inform the latter that the former process has completed (or to simulate that this has happened). Exit signals are described in more detail in §10.4.

Expression
An ERLANG *expression* is a well-formed sequence of tokens (syntactic category *Expr*) that can be evaluated. Either the evaluation completes normally (cf. **normal completion**), in which case the result is the value of the expression, or it completes abruptly, in which case there is an associated reason for the **abrupt completion**.

Fixnum
A fixnum is an ERLANG integer in a range that can be represented efficiently on the platform hosting the implementation. Arithmetic operations on fixnums should be expected to be fast. The largest and smallest fixnums are given by the implementation parameters *maxfixnum* and *minfixnum*. §4.5

Float *Float* is a synonym for floating-point number. Floats are described in more detail in §5.4.

Free variable
A *free variable* in a **context** (cf. §6.2) is a variable that does not belong to that context.

Function
In ERLANG a *function* may compute a function in the mathematical sense but does not necessarily do so. This is because it may have **effects**, such as sending or receiving messages or modifying the state of the process (§10.9.2), the node (§11.7.2) or some port (§12.10.2), and it may also depend on these states and what is received.

All ERLANG functions (including the **BIFs**) are *strict*, i.e., in a function **application** all **arguments** are fully evaluated before any part of the function body is evaluated.

Function symbol
A named function is declared through a *FunctionDeclaration*, in which case its name consists of a function symbol that is an atom and an **arity** which is a nonnegative integer.

Garbage collection
Garbage collection is the activity of automatically reclaiming memory that can no longer be referenced. In ERLANG this is done fully

automatically. It is commonly believed that such automatic memory management reduces the number of severe programming errors and thus shortens development time and product quality, possibly at a modest expense of execution time. (One of the differences between the newer language Java [9] over the older language C [13] is that Java provides garbage collection.)

Guard

A *guard* (syntactic category *Guard*) is a sequence of guard tests (syntactic category *GuardTest*), which are Boolean-valued expressions. They are used in **clauses** (such as those of function declarations and `receive` expressions). When a clause is considered for selection, all guard tests of the guard must evaluate to `true`. (In contexts where the guard is optional, an omitted guard is equivalent to the trivially true guard `true`.)

The guard tests and their constituent guard expressions (syntactic category *GuardExpr*) have been chosen so that they are independent of the state (i.e., their result depends exclusively on their arguments), have no side effect and (with a few exceptions) take $O(1)$ time (with respect to the size of their arguments).

Implementation

Although we referred to Erlang 4.7.3 as an Erlang implementation, there are actually implementations on various platforms that may differ slightly, for example, due to different operating systems. When we write that something is implementation-defined, no portable Erlang 4.7.3 program can depend on the choice made for a particular platform.

Latin-1

Latin-1 is the popular name for the 8-bit code for representation of characters properly called ISO/IEC 8859-1 [12].

List A list is a sequence of terms that either is empty, or consists of a first element (the head) and a remaining list (the tail). Lists are described in more detail in §4.9.

Literal

A literal is an expression for which the value is considered obvious so no evaluation is necessary. Numerals are obvious examples of literals, other examples from Erlang are string literals and atom literals (§4)

All Erlang literals are **terms** but there are **terms** for which there are no literals, e.g., **PIDs** and **refs**.

Little-endian

When a sequence of **bytes** $b_1, \ldots, b_k$ is interpreted as a *little-endian* numeral, the significance increases monotonically. If the numeral has base 256 and is interpreted as unsigned, its value is

$$b_1 + 256(b_2 + 256(\ldots b_{k-1} + 256 b_k \ldots)) = \sum_{i=1}^{k} 256^{i-1} b_i$$

and we denote this by $\mathit{LittleEndianValue}(\langle b_1, \ldots, b_k \rangle)$.

For example, the four bytes `22 188 72 209` when interpreted as a little-endian numeral represent 3,511,204,886 because $256^0 \cdot 22 + 256^1 \cdot 188 + 256^2 \cdot 72 + 256^3 \cdot 209 = 1 \cdot 22 + 256 \cdot 188 + 65536 \cdot 72 + 16777216 \cdot 209 = 22 + 48128 + 4718592 + 3506438144 = 3511204886$.

Macro

A *macro* is a token abstraction and each macro application is replaced as part of the **preprocessing** by a sequence of tokens. Macros are described in more detail in §7.2.

Message

A message is an Erlang term communicated from one process to another. Each message is queued at the receiving process, which can subsequently read a message by evaluating a `receive` expression (§6.19.9). Messages are described in more detail in §10.5.

Module

A module is a named unit of executable Erlang code. Its external interface is a mapping from (exported) function names to functions. A module is loaded onto a **node**. Modules are described in more detail in §8. Dynamic replacement of modules is described in §9.

Node A node hosts Erlang **processes** and **ports**. There could be more than one node on a computer and a node could run on a multiprocessor computer. All processes on a node share certain resources. Nodes are described in more detail in §11.

Normal completion

When the evaluation of an expression completes normally, a result has been computed that is the value of the expression.

Operator

A *binary operator* is a symbol that is written between two expressions, called its *operands*. For every Erlang binary operator (§3.9) except `=`, such an operator expression is evaluated exactly like an application of a binary function.

A *prefix operator* is a symbol that is written before an expression, called its *operand*. For every ERLANG prefix operator (§3.9), such an operator expression is evaluated exactly like an application of a unary function.

OTP OTP stands for Open Telecom Platform [6] and is a programming environment for telecom applications developed at Ericsson Telecom AB. The ERLANG language is a core component of OTP.

Pattern
A pattern expresses the structure of a term and its syntax (syntactic category *Pattern*) resembles that of a term. It can be matched (§6.6) against a term in an input **context**, which will either fail or succeed with an output context that contains bindings for any **variables** in the pattern that were not in the input context. A *universal pattern* (syntactic category *UniversalPattern*) is a wild card, i.e., it will match any term.

PID A PID stands for a Process IDentifier and is an ERLANG term that uniquely identifies a **process** that exists or has existed. PIDs are (ideally, cf. §4.6) never 'reused' so spawning a new process always yields a new PID. PIDs are described in more detail in §4.6.

Port A port is an ERLANG term that uniquely identifies an external resource. Communication with a port has been designed to be similar to communication with a **process**. Ports are described in more detail in §12.

Preprocessing
A **module** declaration is preprocessed as part of compilation (cf. **compile time**). This involves macro expansion, elimination of the record abstraction and conditional compilation. Preprocessing is described in more detail in §7.1.

Process
A process is an entity that carries out the evaluation of an 'original' function application and can send and receive signals (§10.6), e.g., **messages** and **exit signals**. Processes are described in more detail in §10.

Ref A *ref* is an ERLANG term. The interesting property of refs is that each call of the BIF `make_ref/0` (§13.13.3) is guaranteed to return a universally unique ref and that there is no other way to obtain a ref (e.g., there are no ref literals). Refs are discussed in more detail in §4.4. ("Ref" is obviously short for "reference", but we write simply "ref" here as "reference" has a rather different meaning in some programming languages.)

Run time

When we write that something is carried out at run time, we mean that it should happen at the time an expression is being evaluated, etc. For example, testing the type of **arguments** to **BIFs** is carried out at run time in ERLANG (i.e., the types of the values of the arguments are tested).

Skeleton

A skeleton is an ERLANG expression that reveals the structure of a data structure but for which the individual parts are given by arbitrary expressions.

A skeleton is a **literal** if, and only if, all its subexpressions are **literals**.

Syntactic sugar

When we write that an expression `E` is *syntactic sugar* for some expression `E`$'$, it means that evaluation of `E` should behave exactly like evaluation of `E`$'$. The compiler could implement `E` by replacing it with `E`$'$ or some expression equivalent to `E`$'$.

Term An ERLANG term is an ERLANG expression for which it is obvious without any evaluation what value it denotes. ERLANG terms are described in more detail in §4.

In mathematics, a term is usually a syntactic entity. That is, it can be written using some formal language. This is not the case for all ERLANG terms.

All ERLANG terms except refs, ports, pids, functions and binaries have **literals** denoting them. All occurrences of a literal denote the same term.

Throw

When we say that the evaluation of an expression *throws* `T`, this is short for saying that the evaluation of the expression completes abruptly with an associated reason `{'THROW',T}`.

This is always caused by evaluation of an application `throw(T)` (§13.13.5) and indicates a programmer-controlled nonlocal exit.

Token When we discuss the main grammar of ERLANG (§2.6.3), tokens are its terminals; the tokens themselves are defined by the lexical grammar (§2.6.2).

Tuple An ERLANG k-tuple (where $k \geq 0$) is a mapping from the integers 1, ... , k to terms $\texttt{t}_1$, ... , $\texttt{t}_k$. A view of k-tuples that is closer to the implementation is that they consist of k enumerated *fields*, each of which contains an ERLANG term.

Type The terms of ERLANG are partitioned into a collection of *types* and many operations are only meaningful for terms of a certain type. ERLANG is a *dynamically typed* language, which means that the type of a term is always obvious but that the type of any other kind of expressions, e.g., a variable, is not stated explicitly and generally cannot be inferred at compile time.

The types and terms of ERLANG are described in more detail in §4.

In the descriptions of the **BIFs** of ERLANG, we describe what are the expected types of their arguments and what is the type of their result. The notation used for these types is described in §2.5.

Unicode

The *Unicode* standard, version 2.0, [5] is a 16-bit code for representation of multilingual text. It contains symbols for most scripts used in the world today. It includes the 7-bit ASCII character set as its first 128 characters and the 8-bit Latin-1 character set as its first 256 characters.

Variable

A *variable* (syntactic category *Variable*, cf. §3.16) stands for a term. It can be part of an **expression** or of a **pattern**. In the evaluation of an expression, the value must be found in the **environment** (§6.19.1). In pattern matching (§6.6), the value may already be in the environment, or a value will be added to the environment when the variable is first encountered.

In other words, a variable will be bound locally to a certain term. Unlike in conventional (imperative) programming languages, there is no concept of updating the value of a variable.

Chapter 3

Lexical structure

This chapter describes the lexical structure of ERLANG programs and how a sequence of ASCII characters is translated to a sequence of tokens and full stops.

3.1 Lexical translation

A sequence of ASCII characters is translated into a sequence of ERLANG tokens by applying the following four steps, in order:

1. The sequence of characters is translated to a sequence of input characters and line terminators (§3.3). If the sequence does not end with a line terminator, one is added at the end.
2. The sequence of input characters and line terminators is translated to a sequence of ERLANG input elements (§3.4).
3. The sequence of ERLANG input elements is translated to a sequence of tokens and full stops by discarding white space (§3.5) and comments (§3.6) and detecting full stops (§3.18). The resulting sequence of tokens and full stops is the result of the lexical processing.

3.2 Character classes

ErlangUppercase:
    the capital ASCII letters `A`–`Z` (`\101` to `\132`)

ErlangLowercase:
    the small ASCII letters `a`–`z` (`\141` to `\172`)

ErlangLetter:
    ErlangLowercase
    ErlangUppercase

ErlangDigit:
  the ASCII decimal digits `0`–`9` (`\060` to `\071`)

3.3 Line terminators

Line terminators need to be recognized uniformly across platforms so Erlang compilers and tools can report line numbers and determine the end of comments coherently.

LineTerminator:
  the LF character (“linefeed” or “newline”)

InputCharacter:
  AsciiInputCharacter but not LF

3.4 Input elements

The sequence of input characters and line terminators is translated to a sequence of input elements, i.e., white space (§3.5), comments (§3.6) and tokens.

There are six kinds of tokens: separators (§3.7), keywords (§3.8), operators (§3.9), integer literals (§3.11), float literals (§3.12), character literals (§3.13), string literals (§3.14), atom literals (§3.15), variables (§3.16), universal patterns (§3.17).

Note that white space and comments always will separate tokens.

Input:
  InputElements$_{opt}$

InputElements:
  InputElement
  InputElements InputElement

InputElement:
  WhiteSpace
  Comment
  Token

Token:
  Separator
  Keyword
  Operator
  IntegerLiteral
  FloatLiteral
  CharLiteral
  StringLiteral
  AtomLiteral Variable
  UniversalPattern

3.5 White space

White space is defined as the characters with ASCII codes less than or equal to that of ASCII space, i.e., the control characters and space. The line terminators are composed of control characters but have been identified in a preceding step and are therefore included explicitly below.

WhiteSpace:
  LineTerminator
  ControlCharacter
  the ASCII SP character, also known as "space"

ControlCharacter:
  any ASCII control character (`\000` to `\037`)

3.6 Comments

A comment begins with an ASCII `%` character and extends up to and including the next line terminator.

Comment:
  `%` *InputCharacters*$_{opt}$ *LineTerminator*

InputCharacters:
  InputCharacter
  InputCharacters InputCharacter

Examples of comments:

```
        %A space after the percent is not obligatory but...
% The comment can begin a line.
     %% There can be additional %s in the comment.
```

3.7 Separators

The following 15 tokens are *separators* in ERLANG:

Separator: one of

```
    (    )    {    }    [    ]    .         :
    |    ||   ;    ,    ?    ->   #
```

3.8 Keywords

The following 13 tokens are the *keywords* of ERLANG:

Keyword: one of

```
    after     cond      let       when
    begin     end       of
    case      fun       query
    catch     if        receive
```

Thus they cannot be used as atom literals (§3.15).

The keywords `all_true`, `cond`, `let` and `some_true` are not defined in ERLANG 4.7.3 but are reserved for possible future extensions of the language.

3.9 Operators

The following 29 tokens are the *operators* of ERLANG:

Operator: one of

```
    +    -    *    /    div  rem
    or   xor  bor  bxor bsl  bsr  and  band
    ==   /=   =:=  =/=  <    =<   >    >=
    not  bnot ++   --   =    !    <-
```

Thus they cannot be used as atom literals (§3.15).

3.10 Escape sequences

These are the escape sequences for character literals, string literals and quoted atom literals. All escape sequences begin with a backslash (\).

EscapeSequence:

`\ b`	`%` `\008:` backspace BS
`\ d`	`%` `\177:` delete DEL
`\ e`	`%` `\033:` escape ESC
`\ f`	`%` `\014:` form feed FF
`\ n`	`%` `\012:` linefeed LF
`\ r`	`%` `\015:` carriage return CR
`\ s`	`%` `\040:` space SPC
`\ t`	`%` `\011:` horizontal tab HT
`\ v`	`%` `\013:` vertical tab VT
`\ \`	`%` `\008:` backslash `\`
ControlEscape	`%` `\000 to \037:` 64 less than the char
`\ '`	`%` `\047:` single quote `'`
`\ "`	`%` `\042:` double quote `"`
OctalEscape	`%` `\000 to \777:` from octal value

ControlEscape:
  `\ ^` *ControlName*

ControlName:
  any character between `\100` and `\137`

OctalEscape:
  `\` *OctalDigit*
  `\` *OctalDigit OctalDigit*
  `\` *OctalDigit OctalDigit OctalDigit*

OctalDigit: one of
  `0 1 2 3 4 5 6 7`

In the case of a control escape, it denotes the character which has a code that is 64 less than that of the *ControlName* following `\` and `^`.

In the case of an octal escape, it denotes the integer denoted by the octal numeral. Note that only the octal escapes `\000` to `\177` denote characters. The octal escapes `\200` to `\377` denote noncharacter bytes while `\400` to `\777` denote larger integers.

3.11 Integer literals

An integer literal consists of an optional sign, an optional radix specifier and a sequence of digits in the specified radix (which is decimal if omitted).

IntegerLiteral:
  DecimalLiteral
  ExplicitRadixLiteral

DecimalLiteral:
  Digits[10]

ExplicitRadixLiteral:
  `2 #` *Digits*[2]
  `3 #` *Digits*[3]
  `4 #` *Digits*[4]
  `5 #` *Digits*[5]
  `6 #` *Digits*[6]
  `7 #` *Digits*[7]
  `8 #` *Digits*[8]
  `9 #` *Digits*[9]
  `1 0 #` *Digits*[10]
  `1 1 #` *Digits*[11]
  `1 2 #` *Digits*[12]
  `1 3 #` *Digits*[13]
  `1 4 #` *Digits*[14]
  `1 5 #` *Digits*[15]
  `1 6 #` *Digits*[16]

Digits[k]:
  Digit[k]
  Digits[k] *Digit*[k]

Digit[k]: one of the first k of
  `0 1 2 3 4 5 6 7 8 9 Aa Bb Cc Dd Ee Ff`

An integer literal without an explicit radix should thus be composed of decimal digits and will be interpreted arithmetically as a decimal numeral as described in §5.9.2.

If an explicit radix is given, it consists of a decimal numeral between `2` and `16` followed by a `#` character. Suppose that that the decimal interpretation of this numeral is r. The characters following it should then be drawn from the first r "extended digits" `0`, … , `9`, `A`, … , `F`, where the letters may alternatively be in lower case. Its interpretation as a numeral in radix r is given in §5.9.3.

A compile-time error occurs if an integer literal is too large or too small to be representable as an Erlang integer.

Examples of integer literals:

```
0   1499   -54   2#0010010   -8#377
10#1499   16#fa66   16#FA66
```

(Their values are 0, 1499, -54, 18, -255, 1499, 64 102, 64 102 and 1 234 567, respectively.)

3.12 Float literals

A float literal has five parts: an optional sign, a whole number part, a decimal point, a fractional part and an optional exponent part. The exponent part, if present, is indicated by `E` or `e` and is followed by the exponent as a decimal numeral with an optional sign.

FloatLiteral:
  DecimalLiteral `.` *DecimalLiteral* *ExponentPart*$_{opt}$

ExponentPart:
  ExponentIndicator *Sign*$_{opt}$ *DecimalLiteral*

Sign: one of
  `+ -`

ExponentIndicator: one of
  `E e`

(The rule for *DecimalLiteral* appears in §3.11.) A compile-time error occurs if the magnitude of a float literal is too large or too small to be representable as an Erlang float.

The interpretation of a float literal as a float is given in §5.9.5.

Examples of float literals:

```
1.0   -5.1e6   5.1e+6   0.346E-20
```

3.13 Character literals

A character literal is indicated by `$` followed either by a single character or an escape sequence.

CharLiteral:
  `$` *CharLiteralChar*

CharLiteralChar:
  InputCharacter
  EscapeSequence

The escape sequences are described in §3.10.

Note that `$` should not be followed by white space (§3.5)[1] which should instead be expressed through an escape sequence.

Examples of character literals:

[1]The reasons that `$` should not be followed by whitespace are (i) it is impossible to determine safely from the printed representation of a program which character `$` followed by whitespace and/or a line break actually denotes, and (ii) whitespace (especially at the end of a line) often gets transformed by text processing or text transmission tools.

```
$x   $R   $$   $\n   $\s   $\\   $\^T   $\125
```

The values of these literals are the characters 'x', 'R', dollar, newline, space, backslash, control-T and 'U' (which has ASCII code eightyfive, which is denoted by the octal numeral `125`).

3.14 String literals

A string literal is enclosed in `"` characters.

StringLiteral:
    `"` *StringCharacters*$_{opt}$ `"`

StringCharacters:
    StringCharacter
    StringCharacters StringCharacter

StringCharacter:
    InputCharacter but not *ControlCharacter* or `\` or `"`
    EscapeSequence

The escape sequences are described in §3.10.
Examples of string literals:

```
"Fred"   ""   "\n"   "\e42n"   "Ludwig Van Beethoven"
```

The second literal denotes an empty string.

3.15 Atom literals

An *atom literal*, which we will also refer to simply as an *atom*, is on one of two forms:

- An *unquoted* atom is a nonempty sequence of ERLANG letters, ERLANG digits and the ASCII character '`@`', where the first character must be a lowercase ERLANG letter. Such an atom cannot consist of the same sequence of characters as a keyword (§3.8) or an operator (§3.9).

- A *quoted* atom is a sequence of input characters and escape sequences enclosed in single quotes (`'`). As for string literals, line terminators and control characters cannot appear "naked" in a quoted atom.

AtomLiteral:
    AtomLiteralChars but not a *Keyword* or *Operator*
    `'` *QuotedCharacters*$_{opt}$ `'`

AtomLiteralChars:
  ErlangLowercase $NameChars_{opt}$

NameChars:
  NameChar
  NameChars NameChar

NameChar:
  ErlangLetter
  ErlangDigit
  `@`
  `_`

QuotedCharacters:
  QuotedCharacter
  QuotedCharacters QuotedCharacter

QuotedCharacter:
  InputCharacter but not *ControlCharacter* or \ or '
  EscapeSequence

The escape sequences are described in §3.10.
Examples of atoms:

```
friday   tv@lf@rs@lj@re   one_2_three   '!%r@(\'.\ $\\[[#'
```

The fourth example shows several escape sequences and nonletters.
We say that the *printname* of an atom is

- the atom literal as such, in the case of an unquoted atom;
- the sequence of ASCII characters resulting from decoding of escape sequences and removal of the surrounding quotes, in the case of a quoted atom.

Two atoms are the same if and only if they have the same printname. For example, the atoms `foo` and `'foo'` are the same and so are the atoms `'foo bar'`and `'foo\040bar'`.

3.16 Variables

A *variable* is a nonempty sequence of ERLANG letters, ERLANG digits and the character '`@`', where the first character must be an uppercase ERLANG letter. Since no keyword, operator or literal begins with an uppercase ERLANG letter, there can be no ambiguity between them.

Variable:
  _ *NameChars*
  ErlangUppercase NameChars$_{opt}$

Note that a *single* underscore is not a *Variable* but a *UniversalPattern* (cf. §3.17).

Examples of variables:

```
MostSignificantDigit   X   Best_guess   _Rest   LOUD
```

It is recommended that compilers warn about a variable that has only a binding occurrence (and thus no applied occurrences) unless the variable begins with an underscore.

3.17 Universal pattern

A *universal pattern* consists of a single underscore.

UniversalPattern:
  _

It is thus syntactically similar to a variable but may only appear in patterns (cf. §6.6) where occurrences of the universal pattern are like binding occurrences of distinct variables that have no other occurrences. Matching against the universal pattern thus always succeeds.

3.18 Separated token sequences

In the final step of lexical processing, white space and comments are discarded. At the same time, each occurrence of a single period token (i.e., '`.`', which is a separator) that is followed by white space or a comment is replaced by a *FullStop*. The final result is a sequence of tokens and full stops where each full stop should be thought of as terminating the preceding sequence of tokens.

TerminatedTokens:
  TokenSequences$_{opt}$

TokenSequences:
  TokenSequence
  TokenSequences TokenSequence

TokenSequence:
  Tokens FullStop

Tokens:
    Token
    Tokens Token

FullStop:
    . *WhiteSpace*
    . *Comment*

Chapter 4

Types and terms

4.1 Types in Erlang

ERLANG is a *dynamically typed* language, which means that the type of a variable or expression generally cannot be determined at compile time. The dynamic typing offers a high degree of flexibility in that a variable can take on, for example, an integer in one invocation but a list in another invocation. This corresponds to having union types in a statically typed language. Some of the advantages of polymorphic static typing can be achieved also for well-structured ERLANG programs by adding type declarations and type analysis [17].

Every ERLANG term belongs to exactly one of the types below.

The types in ERLANG can be divided into *elementary types* and *compound types*. A term of an elementary type never properly contains an arbitrary ERLANG term and is said to be an *elementary term*. A term of a compound type is said to be a *compound term* and has a number of *immediate subterms*.

The elementary types in ERLANG are:

- Atoms (§4.2).
- Numbers (integers and floats) (§4.3).
- Refs (§4.4).
- Binaries (§4.5).
- Process identifiers (§4.6).
- Ports (§4.7).

The compound types in ERLANG are:

- Tuples (§4.8).

- Lists and conses (§4.9).

The BIFs for recognizing terms of a certain type are described in §§13.1.

4.2 Atoms

The only distinguishing property of an atom is its *printname* (cf. §3.15). Two atoms are equal if and only if they have the same printname. The printname of an atom must have at most *maxatomlength* characters. In ERLANG 4.7.3, *maxatomlength* is 255.

The atoms `true` and `false` are called *Boolean*. Thus, when we say that an expression is Boolean, we mean that its value is a Boolean atom. The Boolean atoms are distinguished in that ERLANG provides four operators acting only on them:

- The logical complement operator `not` (§6.16.4).
- The logical operators `and` (§6.15.6), `or` (§6.14.5) and `xor` (§6.14.6).

Boolean atoms are also distinguished in the filters of list comprehensions (§6.19.5).

Comparison for equality between two atoms is $O(1)$. This is accomplished by "interning", i.e., making sure that each occurrence of an atom is represented internally by the same value through a hashtable that maps printnames to atoms and is used each time an atom is to be obtained from its printname. Because such a table grows each time a new atom is created, many consider it bad programming style to write programs that may add new atoms at runtime, e.g., through the BIFs `list_to_atom/1` and `binary_to_term/1`. Strings [§4.9.1] can often be used instead when the $O(1)$ comparison for equality is not needed.

Atoms are recognized by the BIF `atom/1` (§13.1).

There are no operators acting specifically on atoms (but note the Boolean operators above).

Atom literals are described in §3.15.

ERLANG BIFs relating to atoms are described in §13.2.

4.3 Numbers

ERLANG has two numeric types: *integers* and *floats*. The arithmetic operations permit arbitrary combinations of integer and float operands, when meaningful. We therefore describe both types together.

ERLANG integers and floats are described in detail in §5.2 and §5.4, respectively.

Integer literals are described in §3.11. Float literals are described in §3.12.

Numbers, integers and floats are recognized by the BIFs `number/1`, `integer/1` and `float/1` , respectively (§13.1).

ERLANG provides the following operators acting on numeric terms:

- The unary plus and minus operators `+`, `-` (§6.16.1, §6.16.2).
- The multiplicative operators `*` (§6.15.1), `/` (§6.15.2), `div` (§6.15.3) and `rem` (§6.15.4).
- The addition operators `+` and `-` (§6.14.1)
- The signed shift operators `bsl` and `bsr` (§6.14).
- The unary bitwise complement operator `bnot` (§6.16.3).
- The integer bitwise operators `band` (§6.15.5), `bor` (§6.14.2) and `bxor` (§6.14.2).

ERLANG BIFs relating to numbers are described in §13.3.

4.3.1 Characters

A character is an integer having a value in the range $[0, 255]$ and is thus a byte.

There are no operators acting specifically on characters.

The characters literals are a subset of the integer literals[1], plus the character literals described in §3.13.

4.4 Refs

Refs are terms for which the only meaningful operations are obtaining a new ref and comparing two refs for equality.

When we describe operations (such as transformation to the external term format, cf. §D) we shall assume that the internal representation of a ref *R* consists of three parts:

- `node[`*R*`]`, which is the node on which *R* was created, represented by an atom;
- `creation[`*R*`]`, a nonnegative integer that is the value of `creation[`*N*`]` for the node *N* on which *R* was created;
- `ID[`*R*`]`, a nonnegative integer which is a “serial number” for *R* on the node on which it was created.

[1]It is possible, but *not* recommended, to use an integer literal to denote a character.

Two refs are equal if all these parts are equal. In Erlang 4.7.3 `ID[R]` is limited to XXX. Thus the BIF `make_ref/0` may eventually produce duplicate values.

There are no ref literals.

Refs are recognized by the BIF `reference/1` (§13.1).

There are no operators acting specifically on refs.

Erlang BIFs relating to refs are described in §13.13.

4.5 Binaries

A *binary* is a sequence of bytes, i.e., a sequence of integers between 0 and 255.

There are no binary literals.

Binaries are recognized by the BIF `binary/1`(§13.1).

There are no operators acting specifically on binaries.

Erlang BIFs relating to binaries are described in §13.4.

Note

The Erlang 4.7.3 implementation has disjoint address spaces for its processes and thus copy terms sent as messages (§10.5). However, in typical applications binaries may be very large and copying them would therefore be expensive. Therefore the Erlang 4.7.3 implementation has a single memory area for all binaries residing on a node and uses indirect addressing. That is, a binary would be represented in the memory of a process by a pointer into the common binary area together with information about length. When sending a binary in a message, only the local information is copied, not the elements of the binary. This also implies that a binary can be split (cf. the BIF `split_binary/2`, §13.4.7) in constant time as no copying of the elements is necessary. Of course this arrangement complicates memory management, as the binary area must be maintained separately.

4.6 Process identifiers

An Erlang *process* is an entity that carries out the evaluation of an application. That particular evaluation is identified by a distinct *process identifier*, usually called simply a *PID*. The PID must be used in order to send messages to the process and when manipulating it (e.g., linking with it or attempting to kill it).

PIDs are elementary terms and a PID can be created only by spawning a process. Spawning a process always yields a PID that is distinct from all accessible PIDs. (PIDs and refs are obviously similar and an inefficient implementation of refs could indeed be obtained by letting `make_ref/0` spawn a new process and use its PID.)

When a process completes, its PID is still a PID but it no longer refers to a process so BIFs cannot use it. The result or effect when a BIF is given the PID of a completed process varies, cf. Section13.8.

Processes are further described in Chapter 10.

When we describe operations (such as transformation to the external term format, cf. §D) we shall assume that the internal representation of a PID `P` consists of three parts:

- `node[P]`, which is the node on which `P` was spawned, represented by an atom;
- `creation[P]`, a nonnegative integer that is the value of `creation[N]` for the node `N` on which `P` was spawned;
- `ID[P]`, a nonnegative integer which is a "serial number" for `P` on the node on which it was spawned.

Two PIDs are equal if all these parts are equal. In ERLANG 4.7.3 `ID[P]` is limited to XXX. Thus the BIFs `spawn/3`, etc., may eventually produce duplicate values.

There are no PID literals.

PIDs are recognized by the BIF `pid/1`(§13.1).

There are no operators acting specifically on processes or PIDs.

ERLANG BIFs relating to processes are described in §13.8.

4.7 Ports

An ERLANG node communicates with resources in the outside world (including the rest of the computer on which it resides) through *ports*. Examples of such external resources are files and non-ERLANG programs running on the same host. An external resource behaves much like an ERLANG process, although interaction with it causes or is caused by events in the outside world.

Each external resource is identified by an ERLANG term that is referred to as a port. When a port is created, it is connected externally to an entity, which is either

- a recently spawned external process or recently opened driver (§12.2);
- a file.

Internally the port is connected to a process, which is originally the process that opened the port.

A process communicates with an external resource through messages sent to a port. Any process can send messages to an external resource. The process connected with a port will receive messages from the resource.

When the external resource is depleted (i.e., the end of the file has been reached, the external process has completed or the driver is closed), the port is closed, corresponding to the termination of a process. A process can be linked to a port and it will then be notified when the port is closed.

Ports are obviously similar to PIDs but do not allow all operations that PIDs allow.

Ports are further described in §12.

When we describe operations (such as transformation to the external term format, cf. §D) we shall assume that the internal representation of a port *Q* consists of three parts:

- `node[Q]`, which is the node on which *Q* was opened, represented by an atom;
- `creation[Q]`, a nonnegative integer that is the value of `creation[N]` for the node *N* on which *Q* was opened;
- `ID[Q]`, a nonnegative integer which is a "serial number" for *Q* on the node on which it was opened.

Two ports are equal if all these parts are equal. In Erlang 4.7.3 `ID[Q]` is limited to 256. Thus the BIF `open_port/2` may eventually produce duplicate values. However, `open_port/2` will never return a duplicate open port and the number of simultaneously open ports is limited to 256.

There are no port literals.

Ports are recognized by the BIF `port/1`(§13.1).

There are no operators acting specifically on ports.

Erlang BIFs relating to ports are described in §13.12.

4.8 Tuples

A k-tuple, where $k \geq 0$, is a mapping from the integers $1, \ldots, k$ to Erlang terms, which are its immediate subterms, or elements. (There is exactly one 0-tuple, which is a void mapping.) We say that the *size* of such a tuple is k. The types of the k terms are independent. A k-tuple can be used as a sequence of k terms where each term can be accessed through its index.

A tuple must have at most *maxtuplesize* elements. In Erlang 4.7.3, *maxtuplesize* is 65535.

Tuple skeletons are described in §6.19.3. A tuple literal is a tuple skeleton where all subexpressions are themselves literals.

The time for accessing a tuple element given the tuple and an index (i.e., what is computed by the BIF `element/2`) is $O(1)$, i.e., a constant-time operation. The element update operation — obtaining a tuple that differs from a given one in exactly one element (i.e., what is computed by the BIF `setelement/3`) — is $O(n)$, where n is the number of elements of the tuple. (A future version of ERLANG may have a different trade-off between element access and element update. For example, reducing the time for element update to $O(\log n)$ may justify increasing the time for element access to $O(\log n)$.)

Tuples are recognized by the BIF `tuple/1`(§13.1).

There are no operators acting specifically on tuples.

ERLANG BIFs relating to tuples are described in §13.5.

4.8.1 Records

A record type `R` has a number of *field names*. A term of record type `R` has a value for each of these fields.

A term of record type `R` is a tuple which has one more element than the number of fields of `R` and having the atom `R` as its first element.

Terms of a record type `R` are recognized by record guard tests (§6.20.1).

There are no operators acting specifically on records.

Record declarations are described in §8.4 and record expressions are described in §6.17.

4.8.2 Functions

A *function* of arity n is a term that can be *applied* to a sequence of n terms. Evaluating the application may cause certain effects and may either never complete, complete abruptly with some associated reason or complete normally with a result.

There are no function literals. However, `fun` expressions, having functions as their values, are described in §6.19.10. Function declarations, described in §8.3, associate a function name with a function in a certain module.

In ERLANG 4.7.3 a function (i.e., the value of a `fun` expression) is represented by a tuple, hence the recognizer `tuple/1` returns `true` for a function. It is not recommended to exploit this representation.

ERLANG BIFs relating to functions are described in §13.8.

4.9 Lists and conses

ERLANG has a constant `[]`, which is called *nil.* ERLANG also has a term-forming binary operator `[`$\cdots$`|`$\cdots$`]` called *cons.* The operands of cons are usually referred to as the *head* and the *tail* of the resulting term and are its immediate subterms.

The arguments of cons can be any terms but the intended use of cons is for forming *lists.* (In any use of cons as a general pairing operator, a 2-tuple [§4.8] could be used instead.)

Let us define which terms are *lists.*

- Nil is an *empty list* (thus having zero elements).
- Cons applied to an arbitrary term and a list (with k elements) is a list (with $k+1$ elements).
- There are no other lists than those constructed by a finite number of applications of the two preceding rules.

A list thus represents a finite sequence. As suggested by the use of the cons operator, the properties of a linked representation should be assumed. Computing the cons operator takes $O(1)$ time and so does obtaining the head and/or the tail of a consed term. However, obtaining an element at an arbitrary position of a list takes $O(n)$ time, where n is the index of the element to retrieve.

In addition to the nil constant and the cons operator, there are additional list skeletons, described in §6.19.4, although for every list skeleton, there is an equal term that is a composition of cons operators and nil constants. A list literal is a list skeleton in which all subexpressions are themselves literals.

Nil and conses are both recognized by the BIF `list/1` (§13.1), although the name is highly misleading.

Erlang provides the following operators acting on lists and conses:

- The list addition operator `++` (§6.13.1).
- The list subtraction operator `--` (§6.13.2).

Erlang BIFs relating to lists and conses are described in §13.6.

4.9.1 Strings

A *string* is a list of characters (§4.3.1) and can be seen as representing a text. Note that a list is a string only if all its elements are characters. It follows that a cons is a string only if its head is a character and its tail is a string.

String literals are described in §3.14 (but note that list literals with character literals also denote strings).

There are no operators acting specifically on strings (but note the list operators above).

Erlang BIFs converting from and to strings are described in various sections of §13.

As strings are lists, note that a string can be used anywhere a list is expected (for example, as operand of a list operator or as argument of a list BIF).

4.9.2 Association lists

An *association list* is a list of 2-tuples. For each 2-tuple we say that the first element is the key and the second element is the value.

Let `lst` be an association list

$$\texttt{[\{}k_1\texttt{,}v_1\texttt{\},\{}k_2\texttt{,}T_v\texttt{\},...,\{}k_n\texttt{,}v_n\texttt{\}]}$$

and let K be the set of keys in `lst`. `lst` represents a mapping which for each key $k \in K$ contains a pair (k, v_j) such that $k = k_j$ and for all i, $1 \leq i < j$, $k \neq k_i$.

When we write that a BIF returns an association list, the first element of each 2-tuple in the returned list is always distinct.

4.10 Relational and equality operators on terms

Erlang provides the following relational and equality operators, acting on a pair of terms, each of any type.

- The comparison operators `<`, `=<`, `>` and `>=` (§6.12.1).
- The (exact) equality operators `=:=` and `=/=` (§6.12.2).
- The arithmetic equality operators `==` and `/=` (§6.12.3).

4.10.1 Coercion

Coercion is applied when computing some arithmetic operators (including the arithmetic equality operators).

The function *toFloat* maps a number to a float as follows:

$$\begin{aligned} \mathit{toFloat}(a) &= a && \text{if } a \text{ is a float;} \\ &= \mathit{cvt}_{\texttt{integer}\rightarrow\texttt{float}}(a) && \text{if } a \text{ is an integer.} \end{aligned}$$

($\mathit{cvt}_{\texttt{integer}\rightarrow\texttt{float}}$ is as defined in LIA-1 [14].)

The function *coerce* maps a pair of terms to a pair of terms as follows:

$$\begin{aligned} \mathit{coerce}(a,b) &= (a,b) && \text{if } a \text{ or } b \text{ is not a number;} \\ &= (\mathit{toFloat}(a), \mathit{toFloat}(b)) && \text{if } a \text{ or } b \text{ is a float;} \\ &= (a,b) && \text{otherwise.} \end{aligned}$$

4.11 Size of data structures

The function *size* gives a measure of the size of an ERLANG term T as an integer. The memory needed for representing T in ERLANG 4.7.3 is $O(\mathit{size}(T))$ (this excludes shared information such as the printname of an atom). The measure is also used in this document for expressing the rate of growth of operations such as comparisons.

- If T is an atom, a fixnum (§5.2), a float, a ref, a PID or a port, then
$$\mathit{size}(T) = O(1).$$
- If T is a bignum (§5.2), then
$$\mathit{size}(T) = O(\log \Re^{-1}[T]).$$
- If T is a binary of k bytes, then
$$\mathit{size}(T) = O(k).$$
- If T is a cons with head T_h and tail T_t, then
$$\mathit{size}(T) = O(1) + \mathit{size}(T_h) + \mathit{size}(T_h).$$
- If T is a tuple with elements $T_1, \ldots, T_k$, or a function with values $T_1, \ldots, T_k$ for the free variables, then
$$\mathit{size}(T) = O(1) + O(k) + \sum_{i=1}^{k} \mathit{size}(T_i).$$
- If T is a function with values $T_1, \ldots, T_k$ for free variables, then
$$\mathit{size}(T) = O(1) + O(k) + \sum_{i=1}^{k} \mathit{size}(T_i).$$

We will allow ourselves to apply *size* also to sets of terms and sets of pairs of terms.

- If t is a set of items $t_1, \ldots, t_k$, then
$$\mathit{size}(t) = O(1) + O(k) + \sum_{i=1}^{k} \mathit{size}(t_i).$$
- If t is a pair of items a and b, then
$$\mathit{size}(t) = O(1) + \mathit{size}(a) + \mathit{size}(b).$$

4.11.1 Equality between terms

(Exact) equality between ERLANG terms a and b is defined as follows:

- If a and b were the result of the same evaluation of an expression, then they are equal.
- Otherwise, if a and b are of different type, then they are not equal.
- Otherwise, equality of a and b depends on the type of a and b:
  - **Atom**: a and b are equal if and only if they have the same printname.
  - **Integer**: a and b are equal if and only if $\Re^{-1}[a] = \Re^{-1}[b]$.
  - **Float**: a and b are equal if and only if $\Re^{-1}[a] = \Re^{-1}[b]$. (As in all programming languages, it is unwise to trust equality for floats, as imprecision due to rounding may lead to unexpected inequalities.)
  - **Ref**: a and b are not equal.
  - **Binary**: a and b are equal if and only if they consist of identical sequences of bytes.
  - **PID**: a and b are not equal.
  - **Port**: a and b are not equal.
  - **Cons**:
    - If a and b are both empty, then they are equal.
    - Otherwise, if a and b are both nonempty, then they are equal if and only if the heads of a and b are equal and the tails of a and b are equal.
    - Otherwise, a and b are not equal.
  - **Tuple**:
    - If a and b have different size, then they are not equal.
    - Otherwise, if all corresponding elements of a and b are pairwise equal, then a and b are equal.
    - Otherwise, a and b are not equal.

  It follows that two records are equal if, and only if, they are of the same type and all corresponding elements are pairwise equal.

  For tuples that represent functions the representation is such that if a and b were the results of (different occurrences of) identical expressions, then equality is not defined; otherwise they are not equal.

Due to how the representation of floats in the external term format (§D), equality is not at all defined for floats that have been transformed to the external format and back again, or have been sent as messages.

The time required for determining exact equality of two terms T_1 and T_2 should be $\min(\mathit{size}(T_1), \mathit{size}(T_2))$.

Arithmetic equality between ERLANG terms a and b is defined as follows:

- If a and b are numbers, then they are arithmetically equal if and only if a' is (exactly) equal to b', where $(a', b') = \mathit{coerce}(a, b)$ (§4.10.1).
- Otherwise, if a and b are both of elementary types, then they are arithmetically equal if and only if they are (exactly) equal.
- Otherwise, if a and b are both lists, then:
  - If a and b are both empty, then they are arithmetically equal.
  - Otherwise, if a and b are both nonempty, then they are arithmetically equal if and only if the heads of a and b are arithmetically equal and the tails of a and b are arithmetically equal.
  - Otherwise, a and b are not arithmetically equal.
- Otherwise, if a and b are both tuples of the same size, then a and b are arithmetically equal if and only if all corresponding elements of a and b are pairwise arithmetically equal.
- Otherwise a and b are not arithmetically equal.

4.11.2 The term order

The *term order* of ERLANG terms, which we will write here as `<`, is a order relation that satisfies the following criteria:

- It is transitive, i.e., if t_1 `<` t_2 and t_2 `<` t_3, then it must be the case that t_1 `<` t_3.
- It is asymmetric, i.e., there can be no terms t_1 and t_2 such that t_1 `<` t_2 and t_2 `<` t_1. (This implies that it is irreflexive, i.e., that there can be no term t such that t `<` t.)
- It is an arithmetic total order relation, i.e., if t_1 is not arithmetically equal to t_2, then exactly one of t_1 `<` t_2 and t_2 `<` t_1 holds, unless t_1 and t_2 are functions.
- The terms are primarily ordered according to their type, in the following order: numbers `<` atoms `<` refs `<` ports `<` PIDs `<` tuples `<` empty list `<` conses `<` binaries.

- Numbers are ordered arithmetically (so there is no distinction between integers and floats in this ordering). For example, `4.5 < 5 < 5.3`.
- Atoms are ordered lexicographically according to the codes of the characters in the printnames. For example, `'' < a < aaa < ab < b`.
- If t_1 and t_2 are both refs, both PIDs or both ports, then t_1 precedes t_2 if and only if either
  * `node(`t_1`)` precedes `node(`t_2`)`, or
  * `node(`t_1`)` equals `node(`t_2`)` and t_1 was created before t_2.
- Tuples are ordered first by their size, then according to their elements lexicographically. For example, `{} < {a} < {aaa} < {xxx} < {aaa,xxx} < {xxx,aaa}`. (It follows that records are ordered first by their number of elements, then according to their type, then according to their elements with fields compared in the order given by $\mathit{record\_field}_R^{-1}$, where R is the record type.) Functions are not ordered, except for equality as described above.
- An empty list precedes a cons (and thus a nonempty list) and conses are ordered first by their heads, then by their tails. (Thus a longer list may precede a shorter list even though a shorter tuple always precedes a longer tuple.) For example, `[] < [a|2] < [a|b] < [a] < [a,a] < [b]`.
- Binaries are ordered first by their size, then according to their elements lexicographically. (That is, the same as the order between tuples of integers.)

4.12 Lifetime of data structures

We say that a term *has identity* if it is

- a ref,
- a PID,
- a port, or
- a compound term (i.e., a tuple or a list) in which some immediate subterm has identity.

An elementary term with identity is created by evaluating an expression on a certain form. (For example, a ref is created by evaluating an application of the BIF `make_ref/0`). In this case, each evaluation of such an expression creates a new term that can be distinguished from all other such terms. If an elementary term with identity is embedded in a compound term, that compound term also has identity. ERLANG 4.7.3 will not share or copy terms with identity except when such sharing or copying is implied by the language semantics.

For a term with identity there is thus a definite moment of creation. There is, however, no corresponding moment of destruction: the lifetime of a term with identity is unbounded.

A term that does not have identity is said to be *generic*.

For generic terms, the concept of lifetime is not meaningful at all as there is no way in which two equal specimen of a generic term could be distinguished. If the value of an expression is a generic term, it would be impossible to tell from two such terms whether they were the result of the same evaluation or two separate evaluations. (Evaluation of the expression might have side effects that would be different for one or more evaluations but that is not the point here.) An implementation is permitted to share or copy generic terms.

For example, when a literal `{1,2,3}` is evaluated more than once, an implementation may let all evaluations return references to the same (generic) tuple or let each evaluation return a new specimen of such a term.

4.13 Memory management

In the previous section we noted that for generic terms, there is no concept of lifetime and for terms with identity, the lifetime is unbounded.

However, ERLANG 4.7.3 will keep track of all references to terms (both generic terms and terms with identity) and when no references to a specimen of a term remain, the memory occupied by the specimen must eventually be reused. There are no "memory leaks," i.e., memory that is not part of the representation of terms that can be referenced but is never reclaimed.

Reclamation of memory in a process (garbage collection) could occur incrementally or in batches but will not cause violation of the scheduling policies described in §10.7.

Chapter 5

Arithmetics

We define the mathematical functions in terms of which the arithmetics of Erlang *are defined. This chapter depends significantly on the international standard document ISO/IEC 10967-1 [14], referenced in this text as LIA-1.*

5.1 Notation

Let $\mathcal{Z}$ be the set of mathematical integers, $\mathcal{R}$ the set of real numbers and $\mathcal{B}$ the set of Booleans, denoted by **true** and **false**.

There are four exceptional values that are not numbers but may be the results of the LIA-1 functions defined in this chapter: **integer_overflow**, **floating_overflow**, **underflow** and **undefined**.

The following definitions are restated from LIA-1. For $x \in \mathcal{R}$, the notation $\lfloor x \rfloor$ stands for the largest integer not greater than x:

$$\lfloor x \rfloor \in \mathcal{Z} \quad \text{and} \quad x - 1 < \lfloor x \rfloor \leq x$$

and $tr(x)$ stands for the integer part of x (truncated towards 0):

$$\begin{aligned} tr(x) &= \lfloor x \rfloor && \text{if } x \geq 0; \\ &= -\lfloor -x \rfloor && \text{if } x < 0. \end{aligned}$$

The following definitions are restated from the 1995 working draft of the international standard document ISO/IEC 10967-2 [15], referenced in this text as LIA-2.

Let S be a subset of $\mathcal{R}$, closed under (arithmetic) negation. The following are four rounding functions for mapping values of $\mathcal{R}$ into S. Given any $x \in \mathcal{R}$,

$$\lfloor x \rfloor_S = \max\{\, z \in S \mid z \leq x \,\}$$

$$\lceil x \rceil_S = \min\{\, z \in S \mid z \geq x \,\}$$

$$\begin{aligned}
truncate_S(x) &= \lfloor x \rfloor_S && \text{if } x \geq 0;\\
&= \lceil x \rceil_S && \text{if } x < 0.\\
nearest_S(x) &= \lfloor x \rfloor_S && \text{if } |\lfloor x \rfloor_S - x| < |x - \lceil x \rceil_S|;\\
&= \lceil x \rceil_S && \text{if } |\lfloor x \rfloor_S - x| > |x - \lceil x \rceil_S|;\\
&= \lfloor x \rfloor_S \text{ or } \lceil x \rceil_S && \text{if } |\lfloor x \rfloor_S - x| = |x - \lceil x \rceil_S|.
\end{aligned}$$

In addition it must hold that $nearest_S(-x) = -nearest_S(x)$.

When the subscript S is omitted, $\mathcal{Z}$ is assumed.

We may write $floor_S(x)$ for $\lfloor x \rfloor_S$ and $ceiling_S(x)$ for $\lceil x \rceil_S$.

Note that

- $floor_S(x)$ rounds x towards negative infinity,
- $ceiling_S(x)$ rounds x towards positive infinity,
- $truncate_S(x)$ rounds x towards zero and
- $nearest_S(x)$ rounds x to the nearest value in S.

When we write $[i, j]$, where i and j are integers, we mean the set $\{ x \in \mathcal{Z} \mid i \leq x \leq j \}$. When we write $[i, j)$, where i and j are integers, we mean the set $\{ x \in \mathcal{Z} \mid i \leq x < j \}$.

5.2 The integer type

The set of numbers that can be represented by the integer type is called I and is a subset of $\mathcal{Z}$. LIA-1 requires I to be characterized by four parameters:

$bounded \in \mathcal{B}$	(whether the set I is finite)
$modulo \in \mathcal{B}$	(whether out-of-bounds results "wrap")
$minint \in I$	(the smallest integer in I)
$maxint \in I$	(the largest integer in I)

For the integer type of Erlang, *modulo* and *bounded* are **false**. As *bounded* is **false**, $I = \mathcal{Z}$ and the values of *minint* and *maxint* are not meaningful. Erlang has three additional parameters:

$fixnum \in \mathcal{B}$	(whether there are "fixnums")
$minfixnum \in I$	(the smallest fixnum in I)
$maxfixnum \in I$	(the largest fixnum in I)

fixnum is **true**, *minfixnum* is -2^{27} and *maxfixnum* is $2^{27} - 1$.

Let $I_f = \{ x \in I \mid minfixnum \leq x \leq maxfixnum \}$. I_f is the set of "fixnums", the representation of which can utilize the most efficient representation of integers in the machine, typically occupying one word of memory.

Let $I_b = I \setminus I_f$. I_b is the set of "bignums", the representation of which may require arbitrary amounts of memory. An implementation for which $I_b \neq \emptyset$, such as Erlang, is said to have bignums.

5.3 Integer operations

Elsewhere in this specification we express the integer arithmetic operations of ERLANG in terms of the following functions from LIA-1:

$add_I : I \times I \to I \cup \{\textbf{integer\_overflow}\}$
$(x, y) \mapsto$ the sum of x and y

$sub_I : I \times I \to I \cup \{\textbf{integer\_overflow}\}$
$(x, y) \mapsto$ the difference of x and y

$mul_I : I \times I \to I \cup \{\textbf{integer\_overflow}\}$
$(x, y) \mapsto$ the product of x and y

$div_I : I \times I \to I \cup \{\textbf{integer\_overflow}, \textbf{undefined}\}$
$(x, y) \mapsto$ the quotient of x and y

$rem_I : I \times I \to I \cup \{\textbf{undefined}\}$	$(x, y) \mapsto$ the remainder of x and y
$mod_I : I \times I \to I \cup \{\textbf{undefined}\}$	$(x, y) \mapsto x$ modulo y
$neg_I : I \to I \cup \{\textbf{integer\_overflow}\}$	$(x) \mapsto$ the (arithmetic) negation of x
$abs_I : I \to I \cup \{\textbf{integer\_overflow}\}$	$(x) \mapsto$ absolute value of x
$eq_I : I \times I \to \mathcal{B}$	$(x, y) \mapsto x$ equals y
$neq_I : I \times I \to \mathcal{B}$	$(x, y) \mapsto x$ does not equal y
$lss_I : I \times I \to \mathcal{B}$	$(x, y) \mapsto x$ is less than y
$leq_I : I \times I \to \mathcal{B}$	$(x, y) \mapsto x$ is not greater than y
$gtr_I : I \times I \to \mathcal{B}$	$(x, y) \mapsto x$ is greater than y
$geq_I : I \times I \to \mathcal{B}$	$(x, y) \mapsto x$ is not less than y

For each function, LIA-1 states a number of axioms. In ERLANG 4.7.3, *modulo* = **false** (§5.2), *bounded* = **false** and $mod_I = mod_I^a$. For *div* and *rem* the pair div_I^t/rem_I^t is provided.

For convenience we reproduce the strengthened axioms of Section 5.1.3 of LIA-1 here:

$$\begin{aligned}
add_I(x, y) &= x + y && \text{if } x + y \in I;\\
&= \textbf{integer\_overflow} && \text{if } x + y \notin I.\\
sub_I(x, y) &= x - y && \text{if } x - y \in I;\\
&= \textbf{integer\_overflow} && \text{if } x - y \notin I.\\
mul_I(x, y) &= x * y && \text{if } x * y \in I;\\
&= \textbf{integer\_overflow} && \text{if } x * y \notin I.
\end{aligned}$$

$$\begin{aligned}
div^f_I(x,y) &= \lfloor x/y \rfloor && \text{if } y \neq 0 \text{ and } \lfloor x/y \rfloor \in I;\\
&= \textbf{integer\_overflow} && \text{if } y \neq 0 \text{ and } \lfloor x/y \rfloor \notin I;\\
&= \textbf{undefined} && \text{if } y = 0.\\[1ex]
rem^f_I(x,y) &= x - (\lfloor x/y \rfloor * y) && \text{if } y \neq 0;\\
&= \textbf{undefined} && \text{if } y = 0.\\[1ex]
div^t_I(x,y) &= tr(x/y) && \text{if } y \neq 0 \text{ and } tr(x/y) \in I;\\
&= \textbf{integer\_overflow} && \text{if } y \neq 0 \text{ and } tr(x/y) \notin I;\\
&= \textbf{undefined} && \text{if } y = 0.\\[1ex]
rem^t_I(x,y) &= x - (tr(x/y) * y) && \text{if } y \neq 0;\\
&= \textbf{undefined} && \text{if } y = 0.\\[1ex]
mod^a_I(x,y) &= x - (\lfloor x/y \rfloor * y) && \text{if } y \neq 0;\\
&= \textbf{undefined} && \text{if } y = 0.\\[1ex]
neg_I(x) &= -x && \text{if } -x \in I;\\
&= \textbf{integer\_overflow} && \text{if } -x \notin I.\\[1ex]
abs_I(x) &= |x| && \text{if } |x| \in I;\\
&= \textbf{integer\_overflow} && \text{if } |x| \notin I.\\[1ex]
eq_I(x,y) &= \textbf{true} && \text{if } x = y;\\
&= \textbf{false} && \text{if } x \neq y.\\[1ex]
neq_I(x,y) &= \textbf{true} && \text{if } x \neq y;\\
&= \textbf{false} && \text{if } x = y.\\[1ex]
lss_I(x,y) &= \textbf{true} && \text{if } x < y;\\
&= \textbf{false} && \text{if } x \geq y.\\[1ex]
leq_I(x,y) &= \textbf{true} && \text{if } x \leq y;\\
&= \textbf{false} && \text{if } x > y.\\[1ex]
gtr_I(x,y) &= \textbf{true} && \text{if } x > y;\\
&= \textbf{false} && \text{if } x \leq y.\\[1ex]
geq_I(x,y) &= \textbf{true} && \text{if } x \geq y;\\
&= \textbf{false} && \text{if } x < y.
\end{aligned}$$

5.4 The floating-point type

The set of numbers that can be represented by the float type is called F and is a finite subset of $\mathcal{R}$. F may contain both normalized and denormalized values (cf. Section 5.2 of LIA-1); F_N stands for the set of normalized values in F.

LIA-1 requires F to be characterized by five parameters:

$p \in \mathcal{Z}$	(the precision of F)
$r \in \mathcal{Z}$	(the radix of F)
$emin \in \mathcal{Z}$	(the smallest exponent of F)
$emax \in \mathcal{Z}$	(the largest exponent of F)
$denorm \in \mathcal{B}$	(whether F contains denormalized values)

ERLANG 4.7.3 directly uses the float representation of the underlying processor so these parameters are not defined. It is guaranteed, however, that the size of a float is at least 64 bits.

5.5 Floating-point operations

Elsewhere in this specification we express the floating-point arithmetic operations of ERLANG in terms of the following functions from LIA-1:

$add_F : F \times F \to F \cup \{\textbf{floating\_overflow}, \textbf{underflow}\}$
$(x, y) \mapsto$ the sum of x and y

$sub_F : F \times F \to F \cup \{\textbf{floating\_overflow}, \textbf{underflow}\}$
$(x, y) \mapsto$ the difference of x and y

$mul_F : F \times F \to F \cup \{\textbf{floating\_overflow}, \textbf{underflow}\}$
$(x, y) \mapsto$ the product of x and y

$div_F : F \times F \to F \cup \{\textbf{floating\_overflow}, \textbf{underflow}, \textbf{undefined}\}$
$(x, y) \mapsto$ the quotient of x and y

$neg_F : F \to F$ $(x) \mapsto$ the (arithmetic) negation of x

$abs_F : F \to F$ $(x) \mapsto$ absolute value of x

$sign_F : F \to I$ $(x) \mapsto$ the sign of x

$exponent_F : F \to F \cup \{\textbf{undefined}\}$
$(x) \mapsto$ the exponent of x

$fraction_F : F \to F$ $(x) \mapsto x$ scaled by a power of r to the range $[1/r, 1)$

$scale_F : F \times I \times F \to F \cup \{\textbf{floating\_overflow}, \textbf{underflow}\}$
$(x, n) \mapsto$ the product of x and r^n

$succ_F : F \to F \cup \{\textbf{floating\_overflow}\}$
$(x) \mapsto$ the least float greater than x

$pred_F : F \to F \cup \{\textbf{floating\_overflow}\}$
$(x) \mapsto$ the greatest float less than x

$ulp_F : F \to F \cup \{\textbf{underflow}, \textbf{undefined}\}$	
	$(x) \mapsto$ the value of one unit in the last place of x
$trunc_F : F \times I \to F$	$(x) \mapsto x$ with the low $p - n$ digits zeroed
$round_F : F \times I \to F \cup \{\textbf{floating\_overflow}\}$	
	$(x) \mapsto x$ rounded to n significant digits
$intpart_F : F \to F$	$(x) \mapsto$ the integer part of x
$fractpart_F : F \to F$	$(x) \mapsto x$ minus the integer part of x
$eq_F : F \times F \to \mathcal{B}$	$(x, y) \mapsto x$ equals y
$neq_F : F \times F \to \mathcal{B}$	$(x, y) \mapsto x$ does not equal y
$lss_F : F \times F \to \mathcal{B}$	$(x, y) \mapsto x$ is less than y
$leq_F : F \times F \to \mathcal{B}$	$(x, y) \mapsto x$ is not greater than y
$gtr_F : F \times F \to \mathcal{B}$	$(x, y) \mapsto x$ is greater than y
$geq_F : F \times F \to \mathcal{B}$	$(x, y) \mapsto x$ is not less than y

For each function, LIA-1 states a number of axioms. For convenience we reproduce the axioms of Section 5.2.7 of LIA-1 here:

$$\begin{aligned}
add_F(x, y) &= result_F(add^*_F(x + y), rnd_F) \\
sub_F(x, y) &= add_F(x, -y) \\
mul_F(x, y) &= result_F(x * y, rnd_F) \\
div_F(x, y) &= result_F(x/y, rnd_F) && \text{if } y \neq 0; \\
&= \textbf{undefined} && \text{if } y = 0. \\
neg_F(x) &= -x \\
abs_F(x) &= |x| \\
sign_F(x) &= 1 && \text{if } x > 0; \\
&= 0 && \text{if } x = 0; \\
&= -1 && \text{if } x < 0. \\
exponent_F(x) &= \lfloor (\log_r |x| \rfloor + 1 && \text{if } x \neq 0; \\
&= \textbf{undefined} && \text{if } x = 0. \\
fraction_F(x) &= x / r^{exponent_F(x)} && \text{if } x \neq 0; \\
&= \textbf{undefined} && \text{if } x = 0. \\
scale_F(x, n) &= result_F(x * r^n, rnd_F) \\
succ_F(x) &= \min\{\, z \in F \mid z > x \,\} && \text{if } x \neq fmax; \\
&= \textbf{floating\_overflow} && \text{if } x = fmax.
\end{aligned}$$

$$
\begin{aligned}
pred_F(x) &= \max\{\, z \in F \mid z < x \,\} && \text{if } x \neq -fmax;\\
&= \textbf{floating\_overflow} && \text{if } x = -fmax.\\
ulp_F(x) &= r^{e_F(x)-p} && \text{if } x \neq 0 \text{ and } r^{e_F(x)-p} \in F;\\
&= \textbf{underflow} && \text{if } x \neq 0 \text{ and } r^{e_F(x)-p} \notin F;\\
&= \textbf{undefined} && \text{if } x = 0.\\
trunc_F(x) &= \lfloor x/r^{e_F(x)-n} \rfloor * r^{e_F(x)-n} && \text{if } x \geq 0;\\
&= -trunc_F(-x, n) && \text{if } x < 0.\\
round_F(x) &= rn_F(x, n) && \text{if } |rn_F(x,n)| \leq fmax;\\
&= \textbf{floating\_overflow} && \text{if } |rn_F(x,n)| > fmax.\\
intpart_F(x) &= sign_F(x) * \lfloor |x| \rfloor\\
fractpart_F(x) &= x - intpart_F(x)\\
eq_F(x,y) &= \textbf{true} && \text{if } x = y;\\
&= \textbf{false} && \text{if } x \neq y.\\
neq_F(x,y) &= \textbf{true} && \text{if } x \neq y;\\
&= \textbf{false} && \text{if } x = y.\\
lss_F(x,y) &= \textbf{true} && \text{if } x < y;\\
&= \textbf{false} && \text{if } x \geq y.\\
leq_F(x,y) &= \textbf{true} && \text{if } x \leq y;\\
&= \textbf{false} && \text{if } x > y.\\
gtr_F(x,y) &= \textbf{true} && \text{if } x > y;\\
&= \textbf{false} && \text{if } x \leq y.\\
geq_F(x,y) &= \textbf{true} && \text{if } x \geq y;\\
&= \textbf{false} && \text{if } x < y.
\end{aligned}
$$

The functions are expressed in terms of a number of helper functions and sets:

- The set F^* is F extended with all numbers having the same precision as numbers in F_N but larger magnitude.
- The approximate addition function $add_F^* : F \times F \to \mathcal{R}$ is as described in Section 5.2.4 of LIA-1, ideally but not necessarily such that $add_F^*(x, y) = x + y$.
- The functions $e_F : \mathcal{R} \to \mathcal{Z}$ and $rn_F : F \times \mathcal{Z} \to F^*$ are as described in Section 5.2.7 of LIA-1, i.e., they are defined such that

$$
\begin{aligned}
e_F(x) &= \lfloor \log_r |x| \rfloor + 1 && \text{if } |x| \geq fmin_N;\\
&= emin && \text{if } |x| < fmin_N.
\end{aligned}
$$

and

$$rn_F(x,n) = sign_F(x) * \lfloor |x|/r^{e_F(x)-n} + 1/2 \rfloor * r^{e_F(x)-n}$$

- $rnd_F : \mathcal{R} \to F^*$ is the rounding function used when taking an exact result in $\mathcal{R}$ to a p-digit approximation. It must satisfy the requirements stated in Sections 5.2.5 and 5.2.8 of LIA-1. There are two derived constants characterizing rnd_F:

  * *rnd_error* $\in \mathcal{R}$ is the maximum rounding error in ulps;
  * *rnd_style* $\in \{\textbf{nearest}, \textbf{truncate}, \textbf{other}\}$ is the rounding style.

  For Erlang 4.7.3, *rnd_error* is XXX and *rnd_style* is XXX.

- $result_F : \mathcal{R} \times (\mathcal{R} \to F^*) \to F \cup \{\textbf{floating\_overflow}, \textbf{underflow}\}$ is the function described in Section 5.2.6 of LIA-1. The value of $result_F(x, rnd)$, where $x \in \mathcal{R}$ and rnd is a rounding function in $\mathcal{R} \to F^*$, is the result of applying the rounding function to x, provided that the result is in F. If $|x|$ is greater than zero but less than *fmin*, then XXX???

5.6 Conversions

Let $nearest_{I\to F} : I \to F \cup \{\textbf{floating\_overflow}\}$ be defined as

$$nearest_{I\to F}(x) = result_F(x, nearest_F),$$

where $result_F$ is as in §5.5 (cf. Section 5.2.6 of LIA-1) and $nearest_F$ is a rounding-to-nearest function for F (§5.1).

Define the following four functions:

$$\begin{aligned}
floor_{F\to I}(x) &= floor_Z(x) && \text{if } floor_Z(x) \in I;\\
&= \textbf{integer\_overflow} && \text{if } floor_Z(x) \notin I.\\
ceiling_{F\to I}(x) &= ceiling_Z(x) && \text{if } ceiling_Z(x) \in I;\\
&= \textbf{integer\_overflow} && \text{if } ceiling_Z(x) \notin I.\\
truncate_{F\to I}(x) &= truncate_Z(x) && \text{if } truncate_Z(x) \in I;\\
&= \textbf{integer\_overflow} && \text{if } truncate_Z(x) \notin I.\\
nearest_{F\to I}(x) &= nearest_Z(x) && \text{if } nearest_Z(x) \in I;\\
&= \textbf{integer\_overflow} && \text{if } nearest_Z(x) \notin I.
\end{aligned}$$

Note that the four functions $floor_Z$, $ceiling_Z$, $truncate_Z$ and $nearest_Z$ meet the requirements in Section 5.3 of LIA-1 for being used as the rounding function $rnd_{F\to I}$ in a conversion function $cvt_{F\to I}$.

5.7 Representation and evaluation

The purpose of this section is to define notation and terminology that is used in the subsequent chapters.

- If $i \in I$, then $\Re[i]$ is the ERLANG integer representing i.
- If $f \in F$, then $\Re[f]$ is the ERLANG float representing f.
- If $b \in \mathcal{B}$, i.e., **true** or **false**, then $\Re[b]$ is the ERLANG Boolean atom representing b. That is, $\Re[\textbf{true}] = \texttt{true}$ and $\Re[\textbf{false}] = \texttt{false}$.
- If x is one of **integer_overflow**, **floating_overflow**, **underflow** and **undefined**, then $\Re[x]$ is the ERLANG atom `badarith`.

Similarly,

- If $\mathtt{I}$ is an Erlang integer, then $\Re^{-1}[\mathtt{I}] \in I$ is the integer it represents.
- If $\mathtt{F}$ is an Erlang float, then $\Re^{-1}[\mathtt{F}] \in F$ is the real number it represents.
- If $\mathtt{B}$ is an Erlang Boolean atom, then $\Re^{-1}[\mathtt{B}] \in \mathcal{B}$ is the Boolean it represents. That is, $\Re^{-1}[\texttt{true}] = \textbf{true}$ and $\Re^{-1}[\texttt{false}] = \textbf{false}$.

We have a notation for writing the result of evaluating an expression:

- When we write $\mathtt{E} \Rightarrow \mathtt{T}$ we state that evaluating the expression $\mathtt{E}$ completes normally and that its value is the term $\mathtt{T}$. (If the environment is relevant, it is stated elsewhere.)
- When we write $\mathtt{E} \rightsquigarrow \mathtt{R}$ we state that evaluating the expression $\mathtt{E}$ exits with reason $\mathtt{R}$.

5.8 Notification

Whenever the evaluation of the translated ERLANG expressions causes one of the functions defined in the preceding sections of this chapter to return an exceptional value, the evaluation of the translated ERLANG expression exits with reason `badarith`.

A `catch` expression (§6.9) can be used for handling the exception in accordance with Section 6.1.1 of LIA-1.

The usual mechanisms for handling of abnormal completion ensure that in absence of a `catch` expression that catches the arithmetic exception, the process will complete abruptly; any exit signals sent to linked processes will propagate information about the arithmetic exception (§10.4).

5.9 Conversion to and from numerals

We will define conversions from I and F to canonical decimal numerals. Below we will only discuss decimal numerals and thus omit "decimal". We will also define conversions from decimal numerals to I or F.

5.9.1 Integer to decimal numeral

Given an integer $i \in I$, the canonical numeral is defined recursively as follows.

- If $0 \leq i < 10$, then the canonical numeral for i is the decimal digit with value i.
- If $i < 0$, then the canonical numeral for i is a minus sign ('$-$') followed by the canonical numeral for $-i$.
- If $i \geq 10$, then the canonical numeral for i is the canonical numeral for $\lfloor i/10 \rfloor$ followed by the decimal digit with value $i \bmod 10$.

5.9.2 Decimal numeral to integer

Given a sequence of characters, its interpretation as a decimal integer numeral (if any) is defined as follows:

- If the sequence consists of a minus sign followed by decimal digits d_1, … , d_k, then it denotes $-i$, where i is the integer denoted by the digits d_1, … , d_k.
- If the sequence consists of a plus sign followed by decimal digits d_1, … , d_k, then it denotes the same integer as that denoted by the digits d_1, … , d_k.
- If the sequence consists only of decimal digits d_1, … , d_k, then it denotes the integer $\sum_{j=1}^{k} d_j \cdot 10^{k-j}$.
- Otherwise, it does not denote any integer.

Note that this also defines the meaning of a *DecimalLiteral*: if the sequence of characters that it constitutes denotes $i \in I$, then the *DecimalLiteral* denotes $\Re[i]$.

5.9.3 Numeral with radix to integer

In this context, 'A' and 'a' are digits with value 10, 'B' and 'b' are digits with value 11, etc., up to 'F' and 'f' which are digits with value 15. Given a radix r and a sequence of characters, its interpretation as an integer numeral in radix r (if any) is defined as follows:

- If the sequence consists of a minus sign followed by digits $d_1, \ldots, d_k$, then it denotes $-i$, where i is the integer denoted by the digits $d_1, \ldots, d_k$.
- If the sequence consists of a plus sign followed by decimal digits $d_1, \ldots, d_k$, then it denotes the same integer as that denoted by the digits $d_1, \ldots, d_k$.
- If the sequence consists only of digits $d_1, \ldots, d_k$ where each digit d_j, $1 \leq j \leq k$, has a value that is less than r, then it denotes the integer $\sum_{j=1}^{k} d_j \cdot r^{k-j}$.
- Otherwise, it does not denote any integer.

Note that this also defines the meaning of a *ExplicitRadixLiteral*: consider the sequence of characters that it constitutes. Let r be the integer denoted by the digits before the '#' character. If the concatenation of the sign (if any) with the digits following the '#' character denotes $i \in I$ in radix r, then the *ExplicitRadixLiteral* denotes $\Re[i]$.

5.9.4 Float to numeral

The canonical numeral for a float $f \in F$ is defined recursively as follows.

- If $f < 0$, then the canonical numeral for f is a minus sign ('$-$') followed by the canonical numeral for $-f$.
- If $f = 0$, then the canonical numeral for f is the digit '0' followed by a decimal point ('.'), the digit '0', the letter 'e' and the digit '0'.
- If $f > 0$, then let w and e be the unique integers such that $f = w \cdot 10^e$ and $w \bmod 10 \neq 0$. The canonical numeral for f is the canonical numeral for w with a decimal point ('.') inserted after the first digit, followed by the letter 'e', followed by the canonical numeral for $e + \lfloor \log_{10} w \rfloor$. (Obviously $w > 0$ and the canonical numeral for w thus begins with a digit.)

5.9.5 Numeral to float

Given a sequence of characters, the number it denotes (if any) is defined as follows. The sequence of characters should consist of

- a (possibly signed) decimal numeral, which we will call the whole number part;
- a decimal point;
- an unsigned decimal numeral, which we will call the fractional part;

- optionally an 'E' or 'e' followed by a (possibly signed) decimal numeral, which we will call the exponent.

If it does not, then the sequence of characters does not denote any number.

Let e' be the number of digits in the fractional part, let w be the integer denoted by the concatenation of the whole number part and the fractional part, and let e be the integer denoted by the exponent, or zero if there was no exponent part (§5.9.2).

The number in F denoted by the sequence of characters is then

$$\mathit{result}_F(w \cdot 10^{e-e'}, \mathit{rnd}_F).$$

Note that this also defines the meaning of a *FloatLiteral*: if the sequence of characters that it constitutes denotes $f \in F$, then the *FloatLiteral* denotes $\Re[f]$.

Chapter 6

Expressions and Evaluation

ERLANG is on one hand a functional programming language and on the other hand a language with concurrency.

That ERLANG is a functional language means that the central syntactic concept is that of an *expression* which is *evaluated* in order to obtain its *value*, which is the result of the evaluation.

That ERLANG has explicit concurrency means that there is the concept of a *process* and *communication* between processes as an action. A process is a dynamic entity with state that carries out the evaluation of an expression. During its lifetime, it can exchange messages with other processes and create new processes.

Communication is commanded by evaluating an expression (of the form `P ! E`), which means that there are expressions for which evaluation has a *side effect*.[1]

The presence of (side) effects means that some expression do not have a unique value. Two evaluations in the same context might produce different results. Good programmers avoid confusing use of such possibilities.

6.1 Environments

A *binding* is a pair of a variable and a term. An *environment* is a mapping (§2.3) from variables to terms, i.e., a set of bindings such that no two bindings have the same variable in their left halves.

6.2 Binding, effect and result

ERLANG is different from most other programming languages — including other functional programming languages — in that expressions constitute

[1]As achieving the effect is often the sole reason for evaluating the expression, calling it a "side" effect is sometimes misleading.

the *only* major syntactical category. It is customary to make no distinction between expressions (evaluated for their result) and commands (executed for their effect) — cf. C [13], Scheme [20], Standard ML [19], etc. — but all those languages have declarations as a separate category.

An ERLANG expression is always evaluated in an environment, which we refer to as the *input environment* of the expression. The expression may provide bindings for variables not in its input environment. The *output environment* of the expression is then the extension of the input environment with the variable bindings it provides.

Each occurrence of an ERLANG expression has a lexical location. It may be evaluated several times during the execution of a program (for example, if it is located in the body of a function) and the environments in which it will be evaluated may differ. However, the domains of all these environments will be the same. The *input context* of the expression is the set of variables that is the common domain of these input environments; similarly the *output context* is the common domain of the output environments. Note that the output context of an expression always contains the input context; there is no shadowing of variables.

ERLANG has been designed so that the domain of the output environment for an expression is a function of the input domain. This allows the input and output context of every expression occurrence in an ERLANG program to be determined at compile-time. An applied occurrence of a variable that does not belong to the input context where it occurs — usually called an *unbound variable* — can therefore be detected at compile-time and a compiler must do so and give a compile-time error when an unbound variable is detected.

In ERLANG an occurrence of an expression in some environment thus has three roles:

- It provides a (possibly trivial) extension of the environment to other subexpressions of the expression or body in which it occurs. Given the input context of the expression, it is possible to determine at compile time its output context.

- Its evaluation produces effects, i.e., it may cause the process evaluating it to send or receive messages (which could be either interprocess communication or I/O through ports).

- Its evaluation has a result, which is the value of the expression, provided that the evaluation of the expression completes normally. How this value is used depends on the surrounding expression.

In this chapter we go through all ERLANG expressions and explain their syntax, their effects, their results and how they extend the environment at run time (which implies how they extend the context at compile time).

6.3 Variables and their scope

For each occurrence of a variable there will always be an occurrence of the same variable that is its *binding occurrence*. However, in general it cannot be determined until run time which is the binding occurrence. This is due to the lack of a defined order of evaluation. If a variable occurrence is not a binding occurrence, then it is called an *applied occurrence*.

If a variable is in the output context of an expression occurrence but not in its input context, then the variable must have a binding occurrence inside the expression occurrence. As we will see, the binding occurrence of a variable will always be in a pattern but a pattern may also contain applied occurrences of variables.

6.4 Normal and Abrupt Completion of Evaluation

For any expression there is a *normal mode* of evaluation in which the execution is carried out according to the rules laid out in the following sections. If the evaluation of an expression is carried out according to these rules until the computation is finished and the result available, then the expression is said to *complete normally*.

The evaluation of an expression may alternatively *complete abruptly*, always with an associated *reason* which is an Erlang term. The abrupt completion will have one of the following causes:

- The BIF `throw/1` has been applied to a term *T*. The reason for the abrupt completion is then the term `{'THROW',`*T*`}`.
- The BIF `exit/1` has been applied to a term *T*. The reason for the abrupt completion is then the term `{'EXIT',`*T*`}`.
- A run-time error has occurred (for example, in the evaluation of a BIF application), which is then described by a term *T*. The reason for the abrupt completion is then the term `{'EXIT',`*T*`}`. When we write that *evaluation exits with reason* *R*, this is short for writing that *evaluation completes abruptly with reason* `{'EXIT',`*R*`}`.

It follows that abrupt completion due to an error in a BIF is indistinguishable from abrupt completion due to evaluation of the BIF `exit/1`. Indeed the BIF `exit/1` is intended to be used to signal that an error has occurred.

When the evaluation of an expression has completed abruptly, the steps of the normal mode of evaluation of the expression are no longer followed and there is no output environment. Abrupt completion is discussed separately for each kind of expression but in general, abrupt completion of a subexpression causes abrupt completion of the whole expression with the same reason. The exceptions are `catch` expressions, which are intended to be used for catching an abrupt completion and go back to normal mode of evaluation (§6.9).

The BIF `throw/1` is intended for abrupt completion as a form of non-local control and abrupt completion caused by its evaluation should always be caught.

6.5 Order of evaluation

The order in which the subexpressions of an expression are evaluated is not defined, with one exception:

In a body, the expressions are evaluated strictly from left to right.

In order to simplify the presentation of the expressions of ERLANG, we will adopt a convention: When we say that a sequence of expressions E_1, E_2, ..., E_k, $k \geq 0$, is evaluated *in some order* in an environment ϵ, we mean that:

- In the normal mode of evaluation, all expressions are evaluated in some order, say $E_{o_1}, \ldots, E_{o_k}$.
- If the evaluation of some expression E_{o_i}, where $1 \leq i \leq k$, completes abruptly with some reason, the expressions $E_{o_{i+1}}, \ldots, E_{o_k}$ are not evaluated and evaluation of the whole sequence completes abruptly with the same reason.
- ϵ is the input environment of expression E_{o_1}.
- For each i, $1 < i \leq k$, the output environment of expression $E_{o_{i-1}}$. is the input environment of expression E_{o_i}.
- The output environment of the sequence is the output environment of expression E_{o_k}.

Note that sequence of expressions being evaluated in some order does not imply anything about how the values of these expressions are used. This will be described separately for each kind of expression.

The uncertainty about the order of evaluation together with the requirement of §6.2 that the compiler must give a compile-time error for an applied occurrence of an unbound variable implies that the compiler must only accept a program if for any evaluation order, there will not be an applied occurrence of an unbound variable. The effect of this is that when a sequence of expressions will be evaluated in some order, the compiler should assume for each expression that it will be the first to be evaluated, so its input context will be ϵ.

For example, in a context where `X` is unbound, the expression `(X=8) + X` should give a compile-time error. If the left operand of `+` is evaluated first, then the occurrence of `X` in the right operand will have the value `8`. However, if the right operand is evaluated first, then the occurrence of `X` in it will be unbound.

In the same context, the expression `(X=8) + (X=9)` should be accepted by the compiler, because regardless of the order in which the operands are evaluated, the applied occurrence of `X` will be bound. (However, there will be a run-time error because either `X` will be bound to `8` and then matched against `9`, or it will be bound to `9` and then matched against `8`.)

6.6 Pattern matching

Pattern matching occurs as part of the evaluation of several ERLANG language constructs so we describe it separately.

6.6.1 Patterns

Pattern:

AtomicLiteral	(§6.19.2)
Variable	(§3.16)
UniversalPattern	(§3.17)
TuplePattern	
RecordPattern	
ListPattern	

TuplePattern:
  `{` $Patterns_{opt}$ `}`

ListPattern:
  `[ ]`
  `[` *Patterns* $ListPatternTail_{opt}$ `]`

ListPatternTail:
  `|` *Pattern*

Patterns:
  Pattern
  Patterns `,` *Pattern*

RecordPattern:
  `#` *RecordType* *RecordPatternTuple*

RecordType:
  AtomLiteral

RecordPatternTuple:
  `{` $RecordFieldPatterns_{opt}$ `}`

RecordFieldPatterns:
  RecordFieldPattern
  RecordFieldPatterns , *RecordFieldPattern*

RecordFieldPattern:
  RecordFieldName = *Pattern*

RecordFieldName:
  AtomLiteral

(Strictly speaking "cons pattern" would be a more appropriate name for what we call a list pattern.)

We say that two patterns are equal (and thus exchangeable) if they match exactly the same terms resulting in exactly the same bindings.

Part of the idea with pattern matching is to verify that a term has a certain (nested) structure with respect to lists or tuples. It is then obvious that:

- $[P_1]$ equals $[P_1 | []]$.
- $[P_1, P_2, \ldots, P_k]$, where $k > 1$, equals $[P_1 | [P_2, \ldots, P_k]]$.
- $[P_1, P_2, \ldots, P_k | P_{k+1}]$, where $k > 1$, equals $[P_1 | [P_2, \ldots, P_k | P_{k+1}]]$.

We can therefore describe pattern matching as if each *ListPattern* is either [] or [*Pattern* | *Pattern*].

In the scope of a record declaration (§8.4) that establishes R as a record type with n fields, a record pattern $\#R\{F_1=P_1, \ldots, F_k=P_k\}$, where $F_1, \ldots, F_k$ are distinct names of fields in R, is syntactic sugar for a tuple pattern $\{R, Q_2, \ldots, Q_{n+1}\}$ where for each i, $2 \leq i \leq n+1$,

- If there is an integer j, $1 \leq j \leq k$, such that $\mathit{record\_field}_R(F_j) = i$, then Q_i is P_j.
- Otherwise, Q_i is _.

It is a compile-time error if a record pattern is not in the scope of an appropriate record declaration. As record patterns are syntactic sugar, we can describe pattern matching as if they did not exist.

6.6.2 Definition of the pattern matching problem

A pattern matching problem takes as input a *pattern* P, a term T and an (input) environment ϵ and results in either *failure* or *success*, in the latter case together with an (output) environment ϵ' that extends ϵ.

The domain of ϵ' must include the domain of ϵ and all variables occurring in P. We say that ϵ' is minimal if its domain is exactly that.

Informally we can say that pattern matching succeeds if the structure of the pattern `P` is the same as that of the term `T` and there exists an environment ϵ' extending ϵ such that for each occurrence of a variable in `P`, its value in ϵ' is the term in the corresponding position of `T`. ϵ' is then the output environment if it is minimal.

More precisely, the pattern matching succeeds with an output environment ϵ' if ϵ' is a minimal extension of ϵ and `P` matches `T`, which means that exactly one of the following hold:

- `P` is an atomic literal which denotes `T`;
- `P` is a variable which ϵ' maps to `T`;
- `P` is a universal pattern;
- there exists a $k \geq 0$ such that `P` is a tuple pattern `{`P_1`, ... ,` P_k`}`, `T` is a tuple with size k and elements $T_1, \ldots, T_k$, and for each i, $1 \leq i \leq k$, P_i matches T_i;
- `P` is a list pattern `[]` and `T` is an empty list;
- `P` is a list pattern `[`P_h`|`P_t`]`, `T` is a cons with head T_h and tail T_t, P_h matches T_h and P_t matches T_t.

If pattern matching succeeds with an environment ϵ', then ϵ' is unique (as can be shown).

6.6.3 Coding pattern matching

As will be obvious below where pattern matching is used, the pattern is available at compile-time and so is the context, as was noted in §6.2. Therefore the pattern matching can be computed by code that traverses the term and verifies that the structure is the same as in the pattern, filling in values for variables not in the input context as they are encountered in the pattern. When a variable not in the input context of the pattern occurs more than once in the pattern, any occurrence can be proclaimed the binding occurrence. It should be the one actually visited first by the pattern matching algorithm being used.

Here is an example of how code for matching `P` against `T` could be generated. The generated code examines the term `T`, provided at run time. We assume that the representation of environments is such that for each variable there is a *location* with undefined initial contents that can be written once with a value for the variable. We assume that when the code is run, ϵ' has been obtained by extending ϵ with locations for the variables that occur in `P` but not in ϵ. If execution passes through all the code, the pattern matching has succeeded and all locations in ϵ' have been written with values.

We describe recursively the code generation for a pattern p with `P` as initial value. At run time, t should be the term against which p is to be matched. The initial value of t will be `T`.

- If p is an atomic literal, generate code that finishes the matching with failure if t is not exactly that literal.
- If p is the binding occurrence of a variable, generate code that writes t in its location in ϵ'.
- If p is a variable but not the binding occurrence, generate code that finishes the matching with failure if the contents of its location in ϵ' is not (exactly) equal to t.
- If p is the universal pattern, generate no code.
- If p is a tuple pattern `{`p_1`,`...`,`p_k`}` (where $k \geq 0$), generate code that:
  - If t is not a tuple of size k, complete with failure.
  - Match p_1 against element 1 of t.

    ...
  - Match p_k against element k of t.
- If p is a list pattern `[]`, generate code that finishes the matching with failure if t is not an empty list.
- If p is a list pattern `[`p_h`|`p_t`]` (or some list pattern that is equal to such a pattern), generate code that:
  - If t is not a cons, finish the matching with failure.
  - Match p_h against the head of t.
  - Match p_t against the tail of t.

As there are no loops in the generated code (and assuming that testing for equality always completes) the matching must either finish with failure or reach the end and thus finish successfully.

6.7 Functions, function applications and calls

Function application is part of the evaluation of several kinds of ERLANG expressions, so we describe it once and for all here. The syntax of these expressions is described elsewhere (§6.18, §13.8), as is the syntax of the expressions that name or denote functions (§6.19.10, §8.3).

Evaluation of a function application consists of two parts: evaluation of the arguments of the function and a *function call*. How arguments are evaluated is described separately for each form of function application and the function call never begins until all arguments have been evaluated so here will be described only how the actual function call is evaluated.

The input to a function call is some specification of which function is to be applied and the values of the arguments as a sequence of terms v_1, ... , v_n, for some $n \geq 0$.

The function to be applied is always specified in one of the following four ways:

1. A *remote application*: two atoms `Mod` and `Fun`. Let `P` be the process evaluating the application.

   - If a row with key (`Mod`, `Fun`, `n`) is in `entry_points[node[P]]` (§11.7.2), then the function to be applied is `Fun/n` in the module named *Mod* and the value of the row is a pointer to the executable code (§9.3).
   - Otherwise, if there is a row with key (`E`, `undefined_function`, 3) in `entry_points[node[P]]` (§10.9.2, §11.7.2), where `E` is the value of `error_handler[P]`, then the result of the function application is obtained by instead evaluating an application

     `E:undefined_function(Mod,Fun,[`v_1`,...,`v_n`]).`

     The initial value of `error_handler[P]` is `error_handler` (cf. below).
   - Otherwise, the result of the function application is obtained by evaluating an application

     `error_handler:undefined_function(Mod,Fun,[`v_1`,...,`v_n`])`

     The exported function `undefined_function/3` in the preloaded module `error_handler` exits with `{undef,{Mod,Fun,[`v_1`,...,`v_n`]}}`.

2. An atom `Fun`. The only kind of expression that specifies the function in this way is an *ApplicationExpr* on the form *AtomLiteral* `(` $Exprs_{opt}$ `)`. There are three possibilities:

   - If there is an import attribute (§8.2.2)

     `-import(Mod,[...,F/n,...])`

     then the function to be applied is to be obtained exactly as in case 1 from the atoms `Mod` and `Fun`. (It is thus a remote application.)
   - Otherwise, if there is a definition of a function named `Fun/n` in the lexically enclosing module (which we may assume to be named `Mod`), then the function to be applied is the one so defined. This is called a *local application*.

  - Otherwise, if there is a BIF with an unqualified name *F*/*n*, then it is the function to be applied.
  - Otherwise it is a compile-time error.

3. An *implicit* **fun** *application*: a function term that is the value of an expression `fun` *Fun*/*k*. Let *Mod* be the module in which the `fun` expression lexically occurred. It is a compile-time error if a module declaration contains an expression `fun` *Fun*/*k* but there is no definition of a function named *Fun*/*k* in the declaration of module *Mod*. The function to be applied is thus the function *Mod*:*Fun*/*k* (which needs not be exported).

4. An *explicit* **fun** *application*: a function term that is the value of an explicit `fun` expression (§6.19.10). Let *Mod* be the module in which the `fun` expression lexically occurred.

In cases 3 and 4 above, it may be that the arity of the function is not the same as the number of arguments to which it is being applied. In this case, evaluation of the function application exits with `{badarity,{`*Mod*`,`*Fun*`,[`v_1`, ... ,` v_n`]}}`.

From the four cases above we see that the function to be applied is either one named *Fun*/*n* defined through a *FunctionDeclaration* (§8.3) in some module named *Mod*, or it is denoted by an explicit `fun` expression (§6.19.10). We now describe how the evaluation proceeds in these two cases.

6.7.1 Call of a named function

We shall describe evaluation of a call by a process *P* of the function named *Fun*/*n* in module *Mod* to the values v_1, ... , v_n of the arguments. First, `current_function[`*P*`]` (§10.9.2) should be set to `{`*Mod*`,`*Fun*`,[`v_1`, ... ,`v_n`]}`.[2] Next, let the *FunctionDeclaration* defining *Fun*/*n* in `Mod` be

$$F(P_{1,1}, \ldots, P_{1,n})\ [\texttt{when}\ G_1]\ \texttt{->}\ B_1\ ;$$
$$\vdots\ ;$$
$$F(P_{k,1}, \ldots, P_{k,n})\ [\texttt{when}\ G_k]\ \texttt{->}\ B_k\,.$$

where k is a natural number, each $P_{i,j}$ ($1 \leq i \leq k$ and $1 \leq j \leq n$) is a *Pattern*, each (optional) G_i is a *Guard* and each B_i is a *Body*.

Carry out the following step for each function clause i, where i goes from 1 to k in that order, until a clause s is found for which both pattern matching and guard evaluation succeeds or all clauses have been tried.

- Match the terms v_1, ... , v_n against the patterns $P_{i,1}$, ... , $P_{i,n}$ in an empty environment. If the matching succeeds, evaluate the guard G_i

[2]This is necessary only in order to support the function `process_info/2` (§13.8.12).

in the output environment of the pattern matching (an omitted guard trivially succeeds).

If there is no clause for which both pattern matching and guard evaluation succeeds, then the evaluation of the function application exits with `{function_clause,{`*Mod*`,`*Fun*`,[`v_1`,...,`v_n`]}}`. Otherwise, the evaluation of the function application continues by evaluating the body B_s in the output environment of guard G_s.

6.7.2 Call of an unnamed function

We shall describe evaluation of a call of a function term to the values v_1, ... , v_n of the arguments. Suppose that the function term was obtained by evaluating in an environment ϵ a *FunExpr*

```
fun (P_{1,1},...,P_{1,n}) [when G_1] -> B_1 ;
    ⋮ ;
    (P_{k,1},...,P_{k,n}) [when G_k] -> B_k
end
```

where k is a natural number, each $P_{i,j}$ ($1 \leq i \leq k$ and $1 \leq j \leq n$) is a *Pattern*, each (optional) G_i is a *Guard* and each B_i is a *Body*. Suppose also that the `fun` expression occurred lexically in the module named *Mod*.

Carry out the following step for each function clause i, where i goes from 1 to k in that order, until a clause s is found for which both pattern matching and guard evaluation succeeds or all clauses have been tried.

- Match the terms v_1, ... , v_n against the patterns $P_{i,1}$, ... , $P_{i,n}$ in an empty environment (*not* ϵ). If the matching succeeds, let ϵ'_i be the output environment. Evaluate the guard G_i in $\epsilon \oplus \epsilon'_i$ (an omitted guard trivially succeeds).

If there is no clause for which both pattern matching and guard evaluation succeeds, then the evaluation of the function application exits with `{lambda_clause,`*Mod*`}`. Otherwise, evaluation of the function application continues by evaluating the body B_s in the output environment of guard G_s.

Note that matching the formal parameters (i.e., the patterns of the clauses) against actual parameters (i.e., the terms v_1, ... , v_n) in an empty environment implies that variables in the patterns of a clause shadow variables in the input environment of the `fun` expression. It is recommended that the compiler issues a warning when such shadowing takes place (i.e., when there is a variable in a pattern of a `fun` clause that is bound in the input environment of the `fun` expression).

6.7.3 Extent of function calls and last call optimization

We shall state precisely when a function call begins and ends. Consider a function application where a function is specified in either of the four ways described in §6.7. The function call begins when matching of the patterns of the function clauses (either as identified through a module name, a function symbol and an arity, or as given in an explicit `fun` expression) against the values of the arguments begins. (In the actual implementation there is an entry point of the code for the function, cf. `entry_points[N]` of a node `N`, and the beginning of the function call corresponds to the moment when execution reaches that entry point.)

Note that argument evaluation in the evaluation of a function application thus always occurs before the function call begins.

Note also that in the case of a remote function application (i.e., case 1 of §6.7) when there is no exported function with the given module name, function symbol and arity, there is no function call, so there is no beginning, nor an end.

In order to state when the function call ends we must consider several cases (cf. §6.7.1):

- If there is no clause of the function declaration for which both pattern matching and guard evaluation succeeds, then the function call ends when the evaluation of the function application completes (abruptly).
- Otherwise, let `B` be the body of the selected clause and let `E'` be the final expression in the evaluation of `B` as defined below.
    - ∗ If `E'` is a function application, and evaluation of `B` does not complete abruptly before `E'` is evaluated, then the original function call ends when the function call in `E'` begins. The function call in `E'` is said to be the *last call* of `B`.
    - ∗ Otherwise, the function call ends when the evaluation of the original function application completes (normally or abruptly).

We shall define the final expression of a body and of an expression through mutual recursion. This is well-defined only when evaluation of the body or the expression completes normally.

- The final expression in the evaluation of a body `E`$_1$`, ..., E`$_k$, where $k \geq 1$, is the expression `E`$_k$.
- The final expression in the evaluation of an expression `E` is defined case by case:
    - ∗ If `E` is a block expression `begin B end`, then the final expression of `E` is the final expression of the body `B`.

- ∗ If `E` is an `if` or `case` expression (§6.19.7, §6.19.8), then the final expression of `E` is the final expression of the body of the selected clause of `E`.
- ∗ If `E` is a `receive` expression (§6.19.9), then:
  - ∗ If the expiry time was reached, then the final expression of `E` is the final expression of the expiry body of `E`.
  - ∗ Otherwise, the final expression of `E` is the final expression of the body of the selected clause of `E`.
- ∗ If `E` is a parenthesized expression `(`E'`)`, then the final expression of `E` is E'.
- ∗ Otherwise, the final expression of `E` is `E` itself.

When a function call ends, ERLANG ensures that any resources that are not recycled through garbage collection have been restored. In particular this means that if memory for function calls is allocated on a stack, the size of the stack is the same when a function call begins and when it ends.

A consequence of this requirement and the definition of when a function call ends is that an ERLANG implementation provides *last call optimization* when the final expression in the body of a called function is a function application.

Consider a process `P` that is evaluating an application of the exported function ***`Fun/n`*** in the module named `M`. Let `B` be the binary that contains the compiled code (§7.7) for the version of `M` that is current when the function call begins (§9.1). Process `P` is then *using* function ***`Fun/n`*** in `B` from the time that the function call begins until the function call ends.

6.8 Bodies

A *body* is a nonempty sequence of expressions.

Syntax

Body:
    Exprs

Exprs:
    Expr
    Exprs **,** *Expr*

Evaluation

Evaluation of a body E_1`,` ..., E_k, where $k \geq 1$, with an input environment ϵ is carried out as follows:

- First E_1 is evaluated, then E_2, and so on, until finally E_k is evaluated. The values of expressions E_1, ..., E_{k-1} are completely ignored. If the evaluations of all these expressions complete normally, then the evaluation of the body also completes normally and its value is the value of expression E_k.
- If the evaluation of some expression E_i, where $1 \leq i \leq k$, completes abruptly with some reason R, the expressions E_{i+1}, ..., E_k are not evaluated and evaluation of the body completes abruptly with reason R.

(Recall that in §6.5 we stated that this is the *perceivable* evaluation order. If advantageous from an efficiency point of view, an implementation can often change the order of evaluation of expressions that have no side effects.)

Output environment

- ϵ is used as input environment of expression E_1.
- For each i, $1 < i \leq k$, the output environment of expression E_{i-1} is used as input environment of expression E_i.
- The output environment of expression E_k is used as output environment of the body.

6.9 `catch` expressions

A `catch` expression is used for restoring normal mode of evaluation.

Syntax

Expr:
  `catch` *Expr*
  MatchExpr

Evaluation

Evaluating an expression `catch E` begins by evaluating `E`.

- If evaluation of `E` completes normally and its result is `v`, then evaluation of `catch E` also completes normally with result `v`.
- If evaluation of `E` completes abruptly with reason `{'THROW',T}`, for some term `T`, then evaluation of `catch E` completes normally with result `T`.

- If evaluation of `E` completes abruptly with reason `{'EXIT',T}`, for some term `T`, then evaluation of `catch E` completes normally with result `{'EXIT',T}`.

Output environment

- The input environment of `catch E` is used as input environment of `E`.
- The output environment of `E` is not used;[3] the output environment of `catch E` is the same as its input environment.

6.10 Match expressions

A match expression consists of a pattern and an expression. Its purpose is to match the pattern against the value of the expression, providing bindings for variables having their binding occurrence in the pattern.

Syntax

MatchExpr:
    Pattern = *MatchExpr*
    SendExpr

Evaluation

The evaluation of an expression `P = E`, where `P` is a pattern and `E` is an expression, begins with evaluating `E`.

- If the evaluation of `E` completes abruptly with reason `R`, then the evaluation of the match expression also completes abruptly with reason `R`.
- If the evaluation of `E` completes normally with the term `T` as result, then what remains is matching `P` against `T`.
    - ∗ If the matching succeeds, then the computation of the match expression completes normally with result `T`.
    - ∗ If the matching fails, the computation of the match expression exits with reason `{badmatch,T'}`, where `T'` is some term that is a (not necessarily strict) subterm of `T` such that its top level does not match the corresponding subpattern of `P`.

[3]If evaluation of `E` completes abruptly, then the values of some variables with binding occurrences in `E` may not have been computed. Therefore any bindings in `E` must not be visible outside it.

Output environment

- The input environment of the match expression is used as input environment of `E`.
- The output environment of `E` is used as input environment of the pattern matching.
- The output environment of the pattern matching is used as output environment of the match expression.

6.11 Send expressions

A send expression has two operands. The value of the leftmost operand should identify a process or port to which the value of the rightmost operand will be sent.

Syntax

SendExpr:
    CompareExpr `!` *SendExpr*
    CompareExpr

Evaluation

The evaluation of a send expression E_1 `!` E_2 begins with evaluating the operands E_1 and E_2 in some order in the input environment of the send expression.

Let v_1 and v_2 be the values of E_1 and E_2 respectively.

- If v_1 is a PID or a port, then v_2 is dispatched as a message to v_1 (§10.5, §12.6).
- If v_1 is an atom, then v_1 is looked up in `registry[`N`]`, where N is the node on which the current process is executing.
    - ∗ If there is a process with some PID P registered under the name v_1 on node N (§11.5), then v_2 is dispatched as a message to P.
    - ∗ If there is no process registered under the name v_1 on node N, then evaluation of the send expression exits with `badarg`.
- If v_1 is a 2-tuple of atoms A and N, then A is looked up in `registry[`N`]` (although this lookup is performed on node N).
    - ∗ If there is a process with some PID P registered under the name A on node N, then v_2 is dispatched as a message to P (§10.5).

  * If there is no process registered under the name `A` on node `N`, then the send expression has no effect.

- If v_1 is not a PID, neither a port, nor an atom, nor a 2-tuple of atoms, then evaluation of the send expression exits with `badarg`.

If evaluation completes normally, the value of the send expression is v_2.

Output environment

The output environment of the operands is used as output environment of the send expression.

6.12 Relational and equational operators

The relational and equality operators do not associate neither to left, nor to the right.

These operators can be applied to any pair of values and will always return `true` or `false`.

Syntax

CompareExpr:
  ListConcExpr RelationalOp ListConcExpr
  ListConcExpr EqualityOp ListConcExpr
  ListConcExpr

RelationalOp: one of
```
<    =<    >    >=
```

EqualityOp: one of
```
=:=  =/=   ==   /=
```

Evaluation

Evaluation of an expression E_1 `O` E_2, where `O` is one of the eight relational and equality operators, begins with evaluating the operands E_1 and E_2 in some order in the input environment of E_1 `O` E_2. Let the values of the operands be v_1 and v_2, respectively. See the following sections for how the result is computed for each operator.

Output environment

The output environment of the operands is used as output environment of E_1 `O` E_2.

6.12.1 Relational operators `<`, `=<`, `>`, and `>=`

The terms are compared according to the term order (§4.11.2) and equality (§4.11.1). If the comparison succeeds, the value of the expression is `true`; otherwise it is `false`.

- The operator `<` succeeds if v_1 precedes v_2 in the term order.
- The operator `=<` succeeds if v_1 precedes v_2 in the term order, or v_1 is equal to v_2.
- The operator `>` succeeds if v_2 precedes v_1 in the term order.
- The operator `>=` succeeds if v_2 precedes v_1 in the term order, or v_1 is equal to v_2.

6.12.2 Exact equational operators `=:=`, `=/=`

If the operator is `=:=`, then the value of the expression is `true` if v_1 is (exactly) equal to v_2 and `false` otherwise.

If the operator is `=/=`, then the value of the expression is `false` if v_1 is (exactly) equal to v_2 and `true` otherwise.

6.12.3 Arithmetic equational operators `==`, `/=`

If the operator is `==`, then the value of the expression is `true` if v_1 is arithmetically equal to v_2 and `false` otherwise.

If the operator is `/=`, then the value of the expression is `false` if v_1 is arithmetically equal to v_2 and `true` otherwise.

6.13 List concatenation operators

The list concatenation operators associate to the right. This results in the most efficient computation of expressions on the form E_1 `++` E_2 `++` E_3. However, note that an expression E_1 `--` E_2 `--` E_3 is equivalent to E_1 `--` `(`E_2 `--` E_3`)`, which may be counterintuitive.

Syntax

ListConcExpr:
  AdditionShiftExpr ListConcOp ListConcExpr
  AdditionShiftExpr

ListConcOp: one of
  `++` `--`

Evaluation

Evaluation of an expression `E1 ++ E2` or `E1 -- E2` begins by evaluating the operands E_1 and E_2 in some order in the input environment of the whole expression. Let the values of the operands be v_1 and v_2, respectively. See the following sections for how the result is computed for each operator.

Output environment

The output environment of the operands is used as output environment of the list concatenation expression.

6.13.1 List addition operator `++`

- If v_1 is not a list, the evaluation of `E1 ++ E2` exits with reason `badarg`.
- Otherwise, suppose that the list v_1 has the k elements $x_1, \dots, x_k$. The value of `E1 ++ E2` is then a list with $x_1, \dots, x_k$ as its first k elements and v_2 as its kth tail. This means that if v_2 is a list with the l elements $y_1, \dots, y_l$, then the value of `E1 ++ E2` is a list with the $k+l$ elements $x_1, \dots, x_k, y_1, \dots, y_l$. (If v_2 is not a list, then neither is the value of `E1 ++ E2`.)

The time required for computing the result from v_1 and v_2 should be $O(k)$ (where k is the number of elements in v_1).

Informally, this operator computes the result of concatenating two lists.

6.13.2 List difference operator `--`

- If v_1 or v_2 is not a list, the evaluation of `E1 -- E2` exits with reason `badarg`.
- Otherwise, suppose that v_2 has the l elements $y_1, \dots, y_l$. Let us define inductively a sequence $s_0, \dots, s_l$, each of which is a list. Let s_0 be v_1 and let s_{i+1} (for $0 \leq i < l$) be:

  * If y_{i+1} is an element in s_i, then let s_{i+1} be s_i without the first occurrence of y_{i+1}.
  * Otherwise, let s_{i+1} be s_i.

  The value of `E1 -- E2` is then s_l.

The time required for computing the result from v_1 and v_2 should be $O(kl)$ (where k and l are the number of elements in v_1 and v_2, respectively).

Informally, this operator computes the result of removing the elements of one list from another, a form of "list difference".

6.14 Additive and shift operators

The additive and shift operators associate to the left.

Syntax

AdditionShiftExpr:
  AdditionShiftExpr AdditionOp MultiplicationExpr
  AdditionShiftExpr ShiftOp MultiplicationExpr
  AdditionShiftExpr `or` *MultiplicationExpr*
  AdditionShiftExpr `xor` *MultiplicationExpr*
  MultiplicationExpr

AdditionOp: one of
  `+` `-`
  `bor` `bxor`

ShiftOp: one of
  `bsl` `bsr`

Evaluation

Evaluation of an expression E_1 O E_2, where O is one of the additive operators `+`, `-`, `bor` and `bxor`, shift operators `bsl` and `bsr` or logical operators `or` and `xor` begins with evaluating the operands E_1 and E_2 in some order in the input environment of the whole expression. Let the values of the operands be v_1 and v_2, respectively. See the following sections for how the result is computed for each operator.

Output environment

The output environment of the operands is used as output environment of E_1 O E_2.

6.14.1 Numeric addition operators + and -

- If v_1 or v_2 is not a number, then evaluation of E_1 O E_2 exits with `badarg`.

- Otherwise, if both v_1 and v_2 are integers, $add_I(\Re^{-1}[v_1], \Re^{-1}[v_2])$ (if O is `+`) or $sub_I(\Re^{-1}[v_1], \Re^{-1}[v_2])$ (if O is `-`) is computed; let the result be r.

- Otherwise, the terms v_1 and v_2 are coerced to floats (§4.10.1); let $(w_1, w_2) = coerce(v_1, v_2)$. Next $add_F(\Re^{-1}[w_1], \Re^{-1}[w_2])$ (if O is `+`) or $sub_F(\Re^{-1}[w_1], \Re^{-1}[w_2])$ (if O is `-`) is computed; let the result be r.

If r is a number, the value of E_1 `O` E_2 is $\Re[r]$; otherwise, evaluation of E_1 `O` E_2 exits with $\Re[r]$.

6.14.2 Integer bitwise operator `bor`

- If v_1 or v_2 is not an integer, then evaluation of E_1 `bor` E_2 exits with `badarg`.
- Otherwise, the value of E_1 `or` E_2 is the integer that is the bitwise OR of v_1 and v_2, i.e., the integer that in binary two's-complement representation has a zero in those positions where the binary two's-complement representations of both v_1 and v_2 have a zero, and a one in the other positions.

6.14.3 Integer bitwise operator `bxor`

- If v_1 or v_2 is not an integer, then evaluation of E_1 `bxor` E_2 exits with `badarg`.
- Otherwise, the value of E_1 `xor` E_2 is the integer that is the bitwise XOR of v_1 and v_2, i.e., the integer that in binary two's-complement representation has a one in those positions where the binary two's-complement representation of exactly one of v_1 and v_2 has a one, and a zero in the other positions.

6.14.4 Shift operators `bsl` and `bsr`

These operators compute bitwise shifts in the binary two's-complement representation of integers. The left-hand operator gives the integer to be shifted and the right-hand operator gives the number of positions to shift.

The `bsl` operator computes bitwise shift to the left with zeroes in the lowest-order bits of the result. The `bsr` operator computes arithmetic shift to the right, i.e., with ones in the highest-order bits of the result if the left-hand operand was negative and zeros otherwise.

- If v_1 or v_2 is not an integer, evaluation of the shift expression exits with `badarg`. ERLANG 4.7.3 also exits with `badarg` if v_2 is not a fixnum (§5.2).
- Otherwise, if the operator is `bsl`:
  - ∗ If v_2 is negative, the value of the shift expression is the same as `bsr` for v_1 and $-v_2$.

  - ∗ Otherwise, compute $\Re^{-1}[v_1] \cdot 2^{\Re^{-1}[v_2]}$ and let the result be r. (This is the result of extending the lowest-order bits in the binary two's-complement representation of v_1 with v_2 zeroes.) If $r \in I$, then the value of E_1 `bsl` E_2 is $\Re[r]$; otherwise evaluation of the shift expression exits with $\Re$[**integer_overflow**].

- Otherwise, the operator is `bsr`:
  - ∗ If v_2 is negative, the value of the shift expression is the same as `bsl` for v_1 and $-v_2$.
  - ∗ Otherwise, compute $\lfloor \Re^{-1}[v_1] \cdot 2^{-\Re^{-1}[v_2]} \rfloor$ and let the result be r. (This is the result of removing the v_2 lowest-order bits in the binary two's-complement representation of v_1.) The result is always in I when v_1 is in I; the value of E_1 `bsr` E_2 is $\Re[r]$.

6.14.5 Disjunction operator `or`

- If v_1 or v_2 is not a Boolean, then evaluation of E_1 `or` E_2 exits with `badarg`.
- Otherwise, if at least one of v_1 and v_2 is `true`, the result is `true`.
- Otherwise, the result is `false`.

6.14.6 Exclusion operator `xor`

- If v_1 or v_2 is not a Boolean, then evaluation of E_1 `xor` E_2 exits with `badarg`.
- Otherwise, if exactly one of v_1 and v_2 is `true`, the result is `true`.
- Otherwise, the result is `false`.

6.15 Multiplicative operators

The multiplicative operators associate to the left.

Syntax

MultiplicationExpr:
  MultiplicationExpr MultiplicationOp PrefixOpExpr
  MultiplicationExpr `and` *PrefixOpExpr*
  PrefixOpExpr

MultiplicationOp: one of

```
    *    /
    div  rem
    band
```

Evaluation

Evaluation of an expression `E`$_1$ `O` `E`$_2$, where `O` is one of the five multiplicative operators `*`, `/`, `div`, `rem` and `band` or logical operator `and` begins with evaluating the operands `E`$_1$ and `E`$_2$ in some order in the input environment of `E`$_1$ `O` `E`$_2$. Let the values of the operands be `v`$_1$ and `v`$_2$, respectively. See the following sections for how the result is computed for each operator.

Output environment

The output environment of the operands is the output environment of `E`$_1$ `O` `E`$_2$.

6.15.1 Numeric multiplication operator `*`

- If `v`$_1$ or `v`$_2$ is not a number, then evaluation of `E`$_1$ `*` `E`$_2$ exits with `badarg`.
- Otherwise, if both `v`$_1$ and `v`$_2$ are integers, $mul_I(\Re^{-1}[v_1], \Re^{-1}[v_2])$ is computed; let the result be r.
- Otherwise, the terms `v`$_1$ and `v`$_2$ are coerced to floats (§4.10.1); let $(w_1, w_2) = \mathit{coerce}(v_1, v_2)$. Next $mul_F(\Re^{-1}[w_1], \Re^{-1}[w_2])$ is computed; let the result be r.

If r is a number, the value of `E`$_1$ `*` `E`$_2$ is $\Re[r]$; otherwise, evaluation of `E`$_1$ `*` `E`$_2$ exits with $\Re[r]$.

6.15.2 Float division operator `/`

- If `v`$_1$ or `v`$_2$ is not a number, then evaluation of `E`$_1$ `/` `E`$_2$ exits with `badarg`.
- Otherwise, the terms `v`$_1$ and `v`$_2$ are both coerced to floats (§4.10.1); let $(w_1, w_2) = (\mathit{toFloat}(v_1), \mathit{toFloat}(v_2))$. Next $div_F(\Re^{-1}[w_1], \Re^{-1}[w_2])$ is computed; let the result be r.

If r is a number, the value of `E`$_1$ `/` `E`$_2$ is $\Re[r]$; otherwise, evaluation of `E`$_1$ `/` `E`$_2$ exits with $\Re[r]$.

6.15.3 Integer division operator `div`

- If v_1 or v_2 is not an integer, then evaluation of E_1 `div` E_2 exits with `badarg`.
- Otherwise, $div_I^t(\Re^{-1}[v_1], \Re^{-1}[v_2])$ is computed; let the result be r.

If r is a number, the value of E_1 `div` E_2 is $\Re[r]$; otherwise, evaluation of E_1 `div` E_2 exits with $\Re[r]$.

6.15.4 Integer remainder operator `rem`

- If v_1 or v_2 is not an integer, then evaluation of E_1 `rem` E_2 exits with `badarg`.
- Otherwise, $rem_I^t(\Re^{-1}[v_1], \Re^{-1}[v_2])$ is computed; let the result be r.

If r is a number, the value of E_1 `rem` E_2 is $\Re[r]$; otherwise, evaluation of E_1 `rem` E_2 exits with $\Re[r]$.

6.15.5 Integer bitwise operator `band`

- If v_1 or v_2 is not an integer, then evaluation of E_1 `band` E_2 exits with `badarg`.
- Otherwise, the value of E_1 `band` E_2 is the integer that is the bitwise AND of v_1 and v_2, i.e., the integer that in binary two's-complement representation has a one in those positions where the binary two's-complement representations of both v_1 and v_2 have a one, and a zero in the other positions.

6.15.6 Conjunction operator `and`

- If v_1 or v_2 is not a Boolean, then evaluation of E_1 `and` E_2 exits with `badarg`.
- Otherwise, if both v_1 and v_2 are `true`, the result is `true`.
- Otherwise, the result is `false`.

6.16 Unary operators

The unary operators are `+`, `-`, `bnot` and `not`. The unary operators do not associate. For example, the parentheses are necessary in `bnot(-X)`.

Syntax

PrefixOpExpr:
    PrefixOp RecordExpr
    RecordExpr

PrefixOp: one of
    `+` `-`
    `bnot` `not`

Evaluation

Evaluation of an expression `O E`, where `O` is one of the four unary operators `+`, `-`, `bnot` and `not`, in an environment ϵ begins with evaluating the operand `E` in ϵ. Let its value be `v`. See the following sections for how the result is computed for each operator.

Output environment

The output environment of `E` is used as output environment of `O E`.

6.16.1 Unary plus operator `+`

`v` is returned.[4]

6.16.2 Unary minus operator `-`

The type of the result depends on the type of `v`.

- If `v` is not a number, evaluation of `- E` exits with `badarith`.
- Otherwise, if `v` is an integer, $neg_I(\Re^{-1}[v])$ is computed; let the result be r.
- Otherwise `v` is a float, $neg_F(\Re^{-1}[v])$ is computed; let the result be r.

If r is a number, then the value of `- E` is $\Re[r]$; otherwise, evaluation of `- E` exits with $\Re[r]$.

6.16.3 Bitwise complement operator `bnot`

- If `v` is an integer, then the value of `bnot E` is $(-v) - 1$ (Note that in two's-complement representation this numeral is the bitwise complement of `v` and that $(-v) - 1$ may be representable even when $-v$ is not)
- Otherwise, the evaluation of `bnot E` exits with `badarg`.

[4]It is not intended that the unary `+` operator should be applied to anything but numbers.

6.16.4 Boolean complement operator `not`

- If *v* is the atom `true`, then the value of `not` *E* is `false`.
- If *v* is the atom `false`, then the value of `not` *E* is `true`.
- Otherwise, the evaluation of `not` *E* exits with `badarg`.

6.17 Record expressions

A record declaration (§8.4)

`-record(`*R*`,{`F_1`[=`E_1`],...,`F_n`[=`E_n`]})`

establishes *R* as a record type with n named fields $F_1, \ldots, F_n$. The compiler decides an invertible mapping for *R* from the field names $F_1, \ldots, F_n$ to the integers $2, \ldots, n+1$. Let us call this mapping $\mathit{record\_field}_R$ and its inverse $\mathit{record\_field}_R^{-1}$. A record term of type *R* is then represented by a tuple with $n+1$ elements where the first element is the atom *R* and element i contains the value for field $\mathit{record\_field}_R^{-1}(i)$, where $2 \leq i \leq n+1$.

The value of $\mathit{record\_field}_R(F)$ is available as `#`*R*`.`*F*. For most purposes it is sufficient and appropriate to use this mapping only indirectly through record element access and record update expressions (see below).

Syntax

RecordExpr:
  RecordExpr$_{opt}$ `#` *RecordType* `.` *RecordFieldName*
  RecordExpr$_{opt}$ `#` *RecordType* *RecordUpdateTuple*
  ApplicationExpr

RecordUpdateTuple:
  `{` *RecordFieldUpdates*$_{opt}$ `}`

RecordFieldUpdates:
  RecordFieldUpdate
  RecordFieldUpdates `,` *RecordFieldUpdate*

RecordFieldUpdate:
  RecordFieldName *RecordFieldValue*

RecordFieldValue:
  `=` *Expr*

(The rules for *RecordType* and *RecordFieldName* appear in §6.6.)

There are four kinds of record expression and we will describe them one by one.

6.17.1 Record field index

A record field index expression gives the position of a field in the tuple that is the record.

Evaluation

The value of an expression `#R.F`, where `R` is a record name and `F` is the name of a field in `R`, is the integer $\mathit{record\_field}_{R}(F)$.

It is a compile-time error if the expression is not in the scope of a declaration of a record type `R` having a field `F`.

Output environment

The output environment is the same as the input environment.

Note

ERLANG 4.7.3 treats a record field index expression `#R.F` as syntactic sugar for an integer literal $\mathit{record\_field}_{R}(F)$.

6.17.2 Record field access

A record field access expression extracts one field of a record.

Evaluation

In the scope of a record declaration that establishes `R` as a record type with n fields, the evaluation of an expression `E#R.F`, where `F` is the name of a field in `R`, begins with evaluating the expression `E`; let its value be `v`.

- If `v` is a tuple with $n+1$ elements and element 1 of `v` is `R`, the value of the whole expression is element $\mathit{record\_field}_{R}(F)$ of `v`.
- Otherwise, evaluation of the whole expression exits with `badarg`.

It is a compile-time error if the expression is not in the scope of a declaration of a record type `R` having a field `F`.

Output environment

The input environment of the whole expression is used as input environment of `E` and the output environment of `E` is used as output environment of the whole expression.

Note

ERLANG 4.7.3 treats `E#R.F` as syntactic sugar for a function application `element(`$\mathit{record\_field}_{R}(F)$`,E)`.

6.17.3 Record creation

A record creation expression creates a new record with field values as specified or according to the record declaration.

Evaluation

In the scope of a record declaration that establishes R as a record type with n fields, an expression $\#R\{F_1=E_1, \ldots, F_k=E_k\}$, where R is a record name and $F_1, \ldots, F_k$ are distinct names of fields in R, is evaluated exactly as if it were a tuple skeleton

$\{R, E'_1, \ldots, E'_n\}$

where each expression E'_i, $1 \leq i \leq n$, is as follows:

- If there is an integer j, $1 \leq j \leq k$, such that $\mathit{record\_field}_R(F_j) = i$, then E'_i is E_j.
- Otherwise, if there is a default initializer expression for the field named $\mathit{record\_field}_R^{-1}(i)$, then E'_i is that expression.
- Otherwise, E'_i is the atom literal `undefined`.

It is a compile-time error if the expression is not in the scope of a declaration of a record type R having at least the fields $F_1, \ldots, F_k$, or if $F_1, \ldots, F_k$ are not distinct.

Expressions $E'_1, \ldots, E'_n$ are evaluated in some order, let their values be $v'_1, \ldots, v'_n$. The value of the record creation expression is then a $n+1$-tuple with the atom R as its first element and $v'_1, \ldots, v'_n$ as the remaining elements.

Output environment

The input environment of the whole expression is used as input environment of $E_1, \ldots, E_k$ and the output environment of $E_1, \ldots, E_k$ is used as output environment of the whole expression. However, any default initializer expressions must be evaluated in an empty environment and must have an empty output environment.

Note

ERLANG 4.7.3 treats $\#R\{F_1=E_1, \ldots, F_k=E_k\}$ as syntactic sugar for the tuple skeleton described above.

6.17.4 Record update

A record update expression creates a new record with field values as specified or according to a given record.

Evaluation

In the scope of a record declaration that establishes R as a record type with n fields, the evaluation of an expression E_0#R\{F_1=E_1,...,F_k=E_k\}, where R is a record name and F_1, ... , F_k are distinct names of fields in R, begins with evaluating the expressions E_0, E_1, ... , E_k in some order. Let their values be v_0, v_1, ... , v_k.

- If v_0 is not a tuple with $n+1$ elements, or its first element is not the atom R, then evaluation of the record update expression exits with `badarg`.
- Otherwise, the value of the whole expression is an $n+1$-tuple where element 1 is the atom R and where for each i, $2 \leq i \leq n+1$, element i is obtained as follows:
  - ∗ If there is an integer j, $1 \leq j \leq k$, such that $\mathit{record\_field}_R(F_j) = i$, then element i is v_j.
  - ∗ Otherwise, element i is element i of v_0.

It is a compile-time error if the expression is not in the scope of a declaration of a record type R having at least the fields F_1, ... , F_k, or if F_1, ... , F_k are not distinct.

Output environment

The input environment of the whole expression is used as input environment of E_0, E_1, ... , E_k and the output environment of E_0, E_1, ... , E_k is used as output environment of the whole expression.

Note

ERLANG 4.7.3 treats E_0#R\{F_1=E_1,...,F_k=E_k\} as syntactic sugar for an expression such as

```
case E0 of
    {R,V1,...,Vn} ->
        {R,E'1,...,E'n}
    _ -> exit(badarg)
end
```

where each V_i and E'_i, $1 \leq i \leq n$, is as follows:

- If there is an integer j, $1 \leq j \leq k$, such that $\mathit{record\_field}_R(F_j) = i$, then V_i is a universal pattern and E'_i is E_j.
- Otherwise, V_i is a new variable and E'_i is V_i.

A *new variable* is a variable not in the input domain. Moreover, all new variables are distinct.

6.18 Function application expressions

There are three forms of function application expressions: for applying a named function in the same module, for applying a named and exported function in a different module, and for applying the value of a function expression. There are also BIFs for function application, cf. §13.8. Syntactically the first form is included in the third form.

Syntax

ApplicationExpr:
  PrimaryExpr (*Exprs*$_{opt}$)
  PrimaryExpr : *PrimaryExpr* (*Exprs*$_{opt}$)
  PrimaryExpr

Evaluation

There are three forms of application expressions and we will describe them one by one.

- An expression on the form `Fun(`E_1`,...,`E_k`)`, where `Fun` is a function name, is a *local function application*. It is evaluated by evaluating the expressions E_1, ... , E_k in some order and then as described in case 2 of §6.7. It is a compile-time error if there is no function named `Fun/k` in the lexically enclosing module declaration.

- The evaluation of an expression on the form E_0`(`E_1`,...,`E_k`)` (where E_0 is not an atomic literal) begins with evaluating the expressions E_0, E_1, ... , E_k in some order. Then it depends on the value of E_0:

  ∗ If the value of E_0 is a 2-tuple `{Mod,Fun}`, where `Mod` and `Fun` are atoms, then the expression is a *remote function application* and evaluation proceeds as described in case 1 of §6.7.

  ∗ If the value of E_0 is the result of evaluating an implicit `fun` expression (§6.19.10), then evaluation proceeds as described in case 3 of §6.7.

  ∗ If the value of E_0 is the result of evaluating an explicit `fun` expression (§6.19.10), then evaluation proceeds as described in case 4 of §6.7.

  ∗ Otherwise, evaluation of the application expression exits with `badarg`.

- An expression on the form E_m`:`E_0`(`E_1`,...,`E_k`)` is a remote function application and is evaluated exactly as if it were an application expression `{`E_m`,`E_0`}(`E_1`,...,`E_k`)` (see above).

Output environment

The input environment of the whole expression is used as input environment of [[E_m,] E_0,] E_1, ... , E_k and the output environment of [[E_m,] E_0,] E_1, ... , E_k is used as output environment of the whole expression.

6.19 Primary Expressions

The primary expressions are the building blocks from which other expressions are constructed. Note that many of them are themselves compound but the enclosed expressions can be thought of as being parenthesized so primary expressions have infinite precedence.

PrimaryExpr:
- *Variable*
- *AtomicLiteral*
- *TupleSkeleton*
- *ListSkeleton*
- *ListComprehension*
- *BlockExpr*
- *IfExpr*
- *CaseExpr*
- *ReceiveExpr*
- *FunExpr*
- *QueryExpr*
- *ParenthesizedExpr*

6.19.1 Variables

An applied occurrence of a variable has its value given by a variable binding in the current environment. The requirement to verify at compile time that every applied variable occurrence is in the input context at the occurrence (§6.2) guarantees that there must be such a binding at run time.

Syntax

A *Variable* is a *Token* and its syntax is described in §3.16.

Evaluation

Evaluation of an expression `V`, where `V` is a variable, consists of looking up the value for `V` in the input environment. If there is no binding for `V` in the input environment (§6.1), then it is a compile-time error (§6.2, §6.3).

Output environment

The input and output environments of a variable are the same.

6.19.2 Atomic literals

An *atomic literal* always denotes the same value regardless of the context so no evaluation is necessary to determine which term an atomic literal denotes. All atomic literals except string literals are elementary terms.

Syntax

AtomicLiteral:
  IntegerLiteral
  FloatLiteral
  CharLiteral
  StringLiterals
  AtomLiteral

StringLiterals:
  StringLiteral
  StringLiterals StringLiteral

Evaluation

The compiler can be expected to compile a literal into code that directly creates the internal representation of the term.

- An atom literal always denotes an atom, cf. §4.2.
- An integer literal always denotes an integer, cf. §4.3.
- A float literal always denotes a float, cf. §4.3.
- A character literal always denotes a character, cf. §4.3.1.
- A string literal always denotes a string, cf. §4.9.1.

Note that a string literal may be written as several subsequent stubs, which are nevertheless thought of as constituting a single string literal. For example,

```
"this " "is o" "ne string"
```

is indistinguishable from the string literal

`"this is one string"`.

This suggests one way to write a string literal on more than one line: dividing it into several parts and breaking the lines (and adding indentation) between them.

Output environment

The input and output environments of an atomic literal are the same.

6.19.3 Tuple skeletons

For a tuple skeleton, the "surface structure" is obvious and no evaluation is needed to obtain it. If all immediate subexpressions of a tuple skeleton are literals, the tuple skeleton is itself a (tuple) literal.

Syntax

TupleSkeleton:
    `{` $Exprs_{opt}$ `}`

A tuple skeleton may have at most 255 elements so tuples with more elements must be created in some other way than through syntax (e.g., by calling the BIF `list_to_tuple/1` [§13.5.2]).

Evaluation

For a tuple skeleton it is always obvious that it denotes a tuple with k elements, for some natural number k.

Evaluation of an expression `{`E_1`,...,`E_k`}` begins with evaluating the immediate subexpressions E_1, ... , E_k in some order in the input environment of `{`E_1`,...,`E_k`}`. Let the values of the immediate subexpressions be v_1, ... , v_k. The value of the tuple skeleton is then a k-tuple mapping i to v_i, $1 \leq i \leq k$.

Output environment

The output environment of the immediate subexpressions is used as output environment of `{`E_1`,...,`E_k`}`.

6.19.4 List skeletons

(As for list patterns, "cons skeletons" may be a more accurate name but the main use is for lists.)

For a list skeleton, the "surface structure" is obvious and no evaluation is needed to obtain it. If all immediate subexpressions of a list skeleton are literals, the list skeleton is itself a (list) literal.

Syntax

ListSkeleton:
    `[]`
    `[` *Exprs* $ListSkeletonTail_{opt}$ `]`

ListSkeletonTail:
  `|` *Expr*

Evaluation

For a list skeleton it is always obvious that for some natural number k, it denotes either

- a list with exactly k elements (when on the form `[`E_1`,...,`E_k`]`),
- a list with $k+l$ elements (when on the form `[`E_1`,...,`E_k`|`E_{k+1}`]` and the value of E_{k+1} is a list with l elements), or
- a term that is not a list at all (when on the form `[`E_1`,...,`E_k`|`E_{k+1}`]` and the value of E_{k+1} is not a list).

As for list patterns we should note that certain list skeletons are equal:

- `[`E_1`]` equals `[`E_1`|[]]`.
- `[`E_1`,`E_2`,...,`E_k`]` (where $k > 1$) equals `[`E_1`|[`E_2`,...,`E_k`]]`.
- `[`E_1`,`E_2`,...,`E_k`|`E_{k+1}`]` (where $k > 1$) equals `[`E_1`|[`E_2`,...,`E_k`|`E_{k+1}`]]`.

We can therefore describe evaluation and other properties of list skeletons as if each *ListSkeleton* is either `[]` or `[` *Expr* `|` *Expr* `]`.

A list skeleton `[]` is a literal and its value is an empty list.

An expression `[`E_1`|`E_2`]` can be thought of as if it were an application of a cons operator.[5] We shall therefore refer to E_1 and E_2 as the operands of the expression.

Evaluation of an expression `[`E_1`|`E_2`]` begins with evaluating the operands E_1 and E_2 in some order in the input environment of `[`E_1`|`E_2`]`. Let the values of the operands be v_1 and v_2, respectively. The value of the `[`E_1`|`E_2`]` is then a cons with v_1 as its head and v_2 as its tail. (If v_2 is a list, then so is the value of `[`E_1`|`E_2`]`.)

Output environment

The output environment of the operands is used as output environment of `[`E_1`|`E_2`]`.

6.19.5 List comprehensions

A list comprehension always denotes a list, produced by evaluating an expression for each collection of values of its variables. These collections of values are produced by some generators and are those that in addition satisfy certain filters.

[5]Indeed the corresponding language element in some related languages is a proper (right-associative) operator. For example, in Standard ML [19] the corresponding expression is written as E_1 `::` E_2.

Syntax

ListComprehension:
    `[` *Expr* `||` *ListComprehensionExprs* `]`

ListComprehensionExprs:
    ListComprehensionExpr
    ListComprehensionExprs `,` *ListComprehensionExpr*

ListComprehensionExpr:
    Generator
    Filter

Generator:
    Pattern `<-` *Expr*

Filter:
    Expr

The syntactic constituents of a list comprehension is a *template expression* (to the left of `||`) and a collection of generators and filters that we will call the *body* (to the right of `||`).

A nontrivial generator has at least one variable in its pattern. Each such variable can be expected to appear either in the template expression or in the right-hand side of some later generator or in some later filter. The list is generated by substituting in the template expression all combinations of values that are yielded by the generators, such that the values satisfy all filters.

Each variable occurrence in the pattern of a generator is a binding occurrence so it is a compile-time error if a variable occurs twice in such a pattern. To the right of a generator, all variables occurring in its pattern shadow variables in the input environment (or bound by generators to the left of it). The template expression of a generator is used in the environment in which all generators have contributed their bindings, as we explain in more detail below.

Evaluation

We will explain the result of evaluating a list comprehension

`[` E `||` W_1`,...,`W_k `]`

in terms of a sequence of sequences of environments Φ_0, Φ_1, ... , Φ_k.

The initial sequence of environments, Φ_0, contains exactly one environment, which is the input environment of the list comprehension. Each other sequence of environments Φ_i $(1 \leq i \leq k)$ consists of all extensions of Φ_0 that are generated by the generators in W_1, ... , W_i and that satisfy the filters in W_1, ... , W_i. The value of the list comprehension is a list l of the same length as the number n of environments in Φ_k and for each i, $1 \leq i \leq n$, the ith element of l is the value of the template `E` when evaluated in the ith environment of Φ_k.

We shall now explain how given a sequence of environments Φ_i, we obtain the next sequence of environments Φ_{i+1}, where $0 \leq i < k$.

There are two cases depending on whether W_{i+1} is a generator or a filter.

- W_{i+1} is a generator P_{i+1} `<-` E_{i+1}. Suppose that Φ_i consists of the environments $\epsilon_{i,1}$, ... , ϵ_{i,m_i} for some natural number m_i. For each j $(1 \leq j \leq m_i)$, let $v_{i,j}$ be the value of E_{i+1} when evaluated in $\epsilon_{i,j}$. If there is a j $(1 \leq j \leq m_i)$ for which $v_{i,j}$ is not a list, then the computation of the list comprehension exits with reason `badmatch`. Otherwise, for each j $(1 \leq j \leq m_i)$, let $\Psi_{i,j}$ be the sequence of environments such that each element of $\Psi_{i,j}$ consists of the environment $\epsilon_{i,j}$ extended with the bindings resulting from matching the pattern P_i against the corresponding element of the list $v_{i,j}$. If P_i does not match an element of $v_{i,j}$, that element is discarded, so $\Psi_{i,j}$ may contain fewer elements than $v_{i,j}$. The order between elements in $v_{i,j}$ must be preserved in $\Psi_{i,j}$. Finally, Φ_{i+1} is the result of concatenating the sequences $\Psi_{i,1}$, ... , Ψ_{i,m_i}, in order.

- W_{i+1} is a filter. Φ_{i+1} is obtained by evaluating W_{i+1} in each environment of Φ_i, keeping those environments in which the result is `true` and discarding those environments in which the result is `false`. If for some environment the value of W_{i+1} is not a Boolean, the computation of the list comprehension completes abruptly with reason `badarg`. The order between environments in Φ_i must be preserved in Φ_{i+1}.

Output environment

The input and output environments of a list comprehension are the same.

Examples

The value of the list comprehension in the body

```
Y = [1,2],
Z = 42,
[{X,Y,Z,W} || X <- Y, W <- [a,b], Y <- [4,5], W <- [c,d]]
```

is

```
[{1,4,42,c},{1,4,42,d},{1,5,42,c},{1,5,42,d},
 {1,4,42,c},{1,4,42,d},{1,5,42,c},{1,5,42,d},
 {2,4,42,c},{2,4,42,d},{2,5,42,c},{2,5,42,d},
 {2,4,42,c},{2,4,42,d},{2,5,42,c},{2,5,42,d}]
```

Note that the generator for `X` is evaluated in an expression in which `Y` is `[1,2]`, that the value for `Z` in the input environment is never shadowed by any generator so it is the same for every tuple in the result, and that although the leftmost generator for `W` produces values for `W` that are not accessible in the template pattern, they cause the results to be duplicated (as there are two possible values for the leftmost binding of `W`).

6.19.6 Block expressions

A block expression has no effect on evaluation and is merely a way to parenthesize and delimit a sequence of expressions, i.e., a body. This allows using a body where otherwise only a single expression would be allowed.

Syntax

BlockExpr:
    `begin` *Body* `end`

Evaluation

Evaluating an expression `begin` *`B`* `end` in an environment ϵ means to evaluate the body *`B`* in ϵ (§6.8).

If the evaluation of *`B`* completes abruptly with reason *`R`*, evaluation of the block expression also completes abruptly with reason *`R`*. If it completes normally, the value of `begin` *`B`* `end` is the value of *`B`*.

Output environment

The output environment of *`B`* is used as output environment of `begin` *`B`* `end`.

6.19.7 If expressions

An `if` expression goes through a sequence of clauses, each consisting of a guard and a body, and its value is the value of the first body having a successful corresponding guard.

Syntax

IfExpr:
    `if` *IfClauses* `end`

IfClauses:
    IfClause
    IfClauses `;` *IfClause*

IfClause:
    Guard ClauseBody

ClauseBody:
    `->` *Body*

(The rule for *Guard* appears in §6.20.)

Each clause consists of a *guard* and a *body*.

Evaluation

Evaluation of an expression

```
if
    G_1 -> B_1 ;
    ⋮ ;
    G_k -> B_k
end
```

in an environment ϵ is carried out as follows.

Each guard G_i ($1 \leq i \leq k$) is evaluated (in some order) in ϵ, as described in §6.20, until one succeeds or all guards have failed. Let s be the smallest number such that G_s succeeds, if such a number exists; otherwise evaluation of the `if` expression exits with reason `if_clause`.

The value of the `if` expression is obtained by evaluating the body B_s with the output environment of G_s as input environment.

Output environment

For each clause G_i `->` B_i, $1 \leq i \leq k$, let d_i be the domain of the output environment of G_i when ϵ is the input environment, and let d_i' be the domain of the output environment of B_i when d_i is the domain of its input environment. The output environment of the `if` expression is obtained as the output environment of B_s restricted to the intersection of all d_i', $1 \leq i \leq k$.

Note

The `if` expression above is equal to the following `case` expression (§6.19.8):

```
case L of
    _ when G_1 -> B_1 ;
    ⋮ ;
    _ when G_k -> B_k
end
```

where L is any literal, such as `42`. An `if` expression is thus like a `case` expression but with trivial matching.

6.19.8 Case expressions

A `case` expression chooses between sequences of expressions to evaluate, depending on the value of some expression.

Syntax

CaseExpr:
    `case` *Expr* `of` *CrClauses* `end`

CrClauses:
    CrClause
    CrClauses ; *CrClause*

CrClause:
    Pattern $ClauseGuard_{opt}$ *ClauseBody*

ClauseGuard:
    `when` *Guard*

Guard:
    Body

(The rule for *ClauseBody* appears in §6.19.7.)

We refer to the expression between `case` and `of` as the *switch expression* and to the sequence of clauses to the right of `of` as the *clauses* of the `case` expression. Each clause consists of a *pattern*, an optional *guard* and a *body.* An omitted guard is equivalent with a trivially satisfied `true` guard.

Evaluation

Evaluation of an expression

```
case E of
    P1 [when G1] -> B1 ;
    ⋮ ;
    Pk [when Gk] -> Bk
end
```

in an environment ϵ is carried out as follows.

First the switch expression `E` is evaluated in ϵ. If that evaluation completes abruptly with reason `R`, then evaluation of the `case` expression also completes abruptly with reason `R`. If the evaluation of the switch expression completes normally, let us call its value `v` and its output environment ϵ'.

Next each pattern P_i $(1 \leq i \leq k)$ is matched (in some order) against `v` in ϵ' and (in case of a successful match) the corresponding guard G_i is evaluated in the output environment of P_i. Let s be the smallest number such that P_s matches `v` and G_s succeeds, if such a number exists; otherwise evaluation of the `case` expression exits with reason `case_clause`.

The value of the `case` expression is obtained by evaluating the body B_s in the output environment of G_s.

Output environment

For each clause P_i `when` G_i `->` B_i, $1 \leq i \leq k$, let d_i be the domain of the output environment of P_i when ϵ' is its input environment, let d'_i be the domain of the output environment of G_i when d_i is the domain of its input environment, and let d''_i be the domain of the output environment of B_i when d'_i is the domain of its input environment. The output environment of the `case` expression is obtained as the output environment of B_s restricted to the intersection of all d''_i, $1 \leq i \leq k$.

Note

The name *CrClause* refers to the fact that *CaseExpr* and *ReceiveExpr* both have the same kind of clauses.)

6.19.9 Receive expressions

A `receive` expression normally consumes one message from the message queue of the process evaluating it. The exception is when the `receive` expression specifies an expiry time, there is no suitable message waiting when the processing of the `receive` expression begins and the specified amount of time passes before such a message arrives. `receive` expressions are similar to `case` expressions (S6.19.8) both in syntax and in semantics but cannot be defined in terms of them in a useful way (because a `receive` expression matches expressions against the message queue without removing them unless there is a matching clause).

Syntax

ReceiveExpr:
  `receive` *CrClauses* `end`
  `receive` $\mathit{CrClauses}_{opt}$ `after` *Expr* *ClauseBody* `end`

(The rules for *CrClauses* and *ClauseBody* appear in §6.19.8 and §6.19.7, respectively.)

We refer to the sequence of clauses as the *clauses* of the `receive` expression. Each clause consists of a *pattern*, an optional *guard* and a *body*. An omitted guard is equivalent with a trivially satisfied `true` guard. A `receive` expression containing a part `after` *E* `->` *B* is said to have an *expiry part*. Then *E* is called the *expiry expression* of the `receive` expression and *B* is called the *expiry body*. A clause must either have at least one clause or an expiry part, or both.

Evaluation

Evaluation of an expression

```
receive
    P₁ [when G₁] -> B₁ ;
    ⋮ ;
    Pₖ [when Gₖ] -> Bₖ
[after
    E -> Bₖ₊₁]
end
```

(where k may be zero, in which case there must be an expiry part) in an environment ϵ is carried out as follows by a process *Q*.

The evaluation of a `receive` expression has three parts. The first part only takes place if the `receive` expression has an expiry part. The second part is the same for all `receive` expressions. The third part is somewhat different if the `receive` expression has an expiry part.

- **Part 1.** The purpose of this part is to determine the expiry time of the `receive` expression, if it has an expiry part.

  If the `receive` expression has an expiry part, the expiry expression *E* is evaluated in ϵ. If that evaluation completes abruptly with reason *R*, then evaluation of the `receive` expression also completes abruptly with reason *R*. If the evaluation of the expiry expression completes normally, let us call its value *WaitingTime*. If *WaitingTime* is neither the atom `infinity`, nor a nonnegative integer, the evaluation of the `receive` expression completes abruptly with reason `badarg`. *WaitingTime* will be used in part 3, if evaluation reaches that part. In ERLANG 4.7.3 *WaitingTime* must be a fixnum. Next the system clock of `node[`*Q*`]` is read before anything else happens. Let its value be *Start*.

  Note that the output environment of the expiry expression is not used. Its variable bindings are therefore local.[6]

- **Part 2.** The purpose of this part is to process an existing message if there is one in the queue.

[6]From the syntax it might seem as if bindings of the expiry expression ought to be visible in the expiry body but if the bindings of the expiry expression were visible at all, they could then just as well be visible for all clauses. A careful approach has therefore been to make all its bindings local.

Suppose that `message_queue[Q]` contains the n terms $M_1, \ldots, M_n$, in that order.

For each term M_j ($1 \leq j \leq n$, in that order):

> for each clause i ($1 \leq j \leq k$, in some order):
>
> > match P_i against M_j in ϵ and if that succeeds, evaluate G_i in the output environment of P_i,

until a (first) term M_t has been found for which there is a matching pattern P_s with successful guard G_s, if they exist at all; otherwise evaluation of the `receive` expression continues with part 3 below.

The term M_t is removed from `message_queue[Q]`.

The value of the `receive` expression is then obtained by evaluating the body B_s in the output environment of G_s.

(Note that if necessary, every message in the message queue will be tried against every clause, even if there is an expiry.)

- **Part 3.** The purpose of this part is to wait for a receivable term to appear in the message queue, or to expire. There are three alternatives:

  1. If the `receive` expression has no expiry part, or *`WaitingTime`* is the atom `infinity`, then:

     ➦ `status[Q]` is changed to `waiting` and `timer[Q]` is set to `0`. When `status[Q]` becomes `running` again, if there are messages in `message_queue[Q]` which have not previously been examined, process them as in part 2. If that does not complete the evaluation of the `receive` expression, execution continues at ➦ above.

  2. If the `receive` expression has an expiry part and *`WaitingTime`* is the integer `0`, then:

     The evaluation of the `receive` expression finishes by letting s be $k+1$ and evaluating the expiry body B_s with ϵ as input environment. If its evaluation completes abruptly with reason R, then evaluation of the `receive` expression also completes abruptly with reason R. If the evaluation of the expiry body completes normally, its value is also the value of the `receive` expression. (Note that `status[Q]` is not affected.)

  3. Otherwise, the `receive` expression has an expiry part for which *`WaitingTime`* is a positive integer. `timer[Q]` is set to *`WaitingTime`*.

➡ `status[`*`Q`*`]` is changed to `waiting`. If there are messages in `message_queue[`*`Q`*`]` that have not previously been examined when `status[`*`Q`*`]` becomes `running` again, process them as in part 2. If that does not complete the evaluation of the `receive` expression, the system clock of `node[`*`Q`*`]` is read and if its value is greater than or equal to *`Start`*`+WaitingTime`, evaluation of the `receive` expression finishes as described for case 2. Otherwise, execution continues at ➡ above.

(Note that when there are unexamined messages in the queue, they will be processed even if the expiry time has been reached. This implies that the value of the expiry expression cannot be seen as a hard limit on how much time the evaluation of the receive expression may take.)

Output environment

For each clause P_i `when` G_i `->` B_i, $1 \leq i \leq k$, let d_i be the domain of the output environment of P_i when ϵ is its input environment, let d'_i be the domain of the output environment of G_i when d_i is the domain of its input environment, and let d''_i be the domain of the output environment of B_i when d'_i is the domain of its input environment. Let d''_{k+1} be the domain of the output environment of B_{k+1} when its input environment is ϵ.

The output environment of the `receive` expression is the output environment of B_s restricted to the intersection of all d''_i, $1 \leq i \leq k+1$.

6.19.10 `fun` expressions

A `fun` expression denotes a function. Its value can therefore be applied.

A `fun` expression is similar to a literal in that it is normal (i.e., it cannot be further simplified). However, it is not expected that the value of a `fun` expression is represented internally in such a way that the `fun` expression can be reconstructed.

Syntax

FunExpr:
  `fun` *FunctionArity*
  `fun` *FunClauses* `end`

FunClauses:
  FunClause
  FunClauses `;` *FunClause*

FunClause:
  `(` $Patterns_{opt}$ `)` $ClauseGuard_{opt}$ *ClauseBody*

Evaluation

A `fun` expression is either *implicit* or *explicit*. In the former case, it refers to a named function in the same module and in the latter case, it explicitly describes a function.

- Consider first an implicit `fun` expression on the form `fun F/A`, where *F* is an atom and *A* is a decimal literal. If there is no function with name *F* and arity *A* in the module in which the `fun` expression lexically appears, then it is a compile-time error. Otherwise, the expression `fun F/A` denotes a function term that can be applied. Evaluation of an application of such a function term is described in §6.7.

- Consider now an explicit `fun` expression, i.e., an expression

  $$\begin{array}{l}\texttt{fun }(P_{1,1}\texttt{,}\ldots\texttt{,}P_{1,n_1})\ [\texttt{when } G_1]\ \texttt{->}\ B_1\ \texttt{;}\\ \qquad\vdots\ \texttt{;}\\ \qquad(P_{k,1}\texttt{,}\ldots\texttt{,}P_{k,n_k})\ [\texttt{when } G_k]\ \texttt{->}\ B_k\\ \texttt{end}\end{array}$$

  where k is a natural number, each $P_{i,j}$ ($1 \leq i \leq k$ and $1 \leq j \leq n_i$) is a *Pattern*, each (optional) G_i is a *Guard* and each B_i is a *Body*. It is a compile-time error if there is no number A such that $A = n_1 = \cdots = n_k$. Otherwise the `fun` expression denotes a function term that can be applied. Evaluation of an application of such a function term is described in §6.7.

Output environment

The input and output environments of a `fun` expression are the same, i.e., any variables bound in (an explicit) `fun` expression are local to it.

6.19.11 `query` expressions

Query expressions are only syntactically part of ERLANG but is a query interface to the database system Mnesia, part of OTP [7, ch. 6]. Their semantics will not be described in detail here.

Syntax

QueryExpr:
  `query` *ListComprehension* `end`

There is a syntactic extension to *RecordExpr* that is only valid inside a *QueryExpr*:

RecordExpr:
  RecordExpr `.` *RecordFieldName*

6.19.12 Parenthesized expressions

An expression may always be enclosed in parentheses without changing its meaning. Such parentheses may be used in order to express the syntactic structure of an expression when the grammatical rules would otherwise give another structure.

Syntax

ParenthesizedExpr:
  `(` *Expr* `)`

Evaluation

Evaluating an expression `(E)` means evaluating `E`.

Output environment

The output environment of `E` is used as the output environment of `(E)`.

6.20 Guards

In ERLANG a pattern can optionally be augmented with a *guard* for expressing additional conditions on the term that is to be matched against the pattern. A guard consists of a nonempty sequence of *guard tests*.

The guard tests have subexpressions, which are *guard expressions*. When compared with expressions, both guard tests and guard expressions are syntactically restricted. They are built from a small repertoire of primitives for which it is guaranteed that:

- Their evaluation takes bounded (often constant) time.
- They do not have any effects.
- There are only “simple” errors.

Syntax

Guard:
  GuardTest
  Guard **,** *GuardTest*

Evaluation

Evaluation of a guard completes with success or with failure; there is no concept of abrupt completion.

A guard is evaluated by evaluating the guard tests from left to right until one is found that fails, in which case evaluation of the whole guard fails without evaluating any more guard tests, or all guard tests have been evaluated, in which case evaluation of the whole guard succeeds.

Output environment

The input and output environments of a guard are the same.

6.20.1 Guard tests

Guard tests are syntactically identical with certain Boolean expressions but their semantics are slightly different in the case of abrupt completion.

There are five kinds of guard tests: trivially true tests, record tests, recognizers, term comparisons and parenthesized guard tests.

Syntax

GuardTest:
  `true`
  GuardRecordTest
  GuardRecognizer
  GuardTermComparison
  ParenthesizedGuardTest

GuardRecordTest:
  `record (` *GuardExpr* `,` *RecordType* `)`

GuardRecognizer:
  RecognizerBIF `(` *GuardExpr* `)`

RecognizerBIF:
  AtomLiteral

GuardTermComparison:
  GuardExpr RelationalOp GuardExpr
  GuardExpr EqualityOp GuardExpr

ParenthesizedGuardTest:
  `(` *GuardTest* `)`

Evaluation

We discuss the kinds of guard tests one by one.

- A `true` test succeeds trivially.
- A *GuardRecordTest* `record(E,R)` (where `E` is a guard expression and `R` is a record type) is evaluated by evaluating `E` and then testing the result. The test succeeds if the evaluation of `E` completes normally

with some result `v` and `v` is a record of type `R`; otherwise the test fails. Note that there is no BIF `record/2`, a *GuardRecordTest* just happens to have the same syntax as a function application.

- A *GuardRecognizer* is an application of one of the recognizer BIFs of §13.1 to a guard expression. It is evaluated by evaluating the guard expression and applying the BIF to the result. The test succeeds if the evaluation of the guard expression completes normally with some result `v` and the BIF returns `true` for `v`; otherwise the test fails.
- A *GuardTermComparison* is a *RelationalOp* or an *EqualityOp* applied to a pair of guard expressions. It is applied by evaluating both operands and then computing a boolean as described in §6.12. The test succeeds if evaluations of both operands complete normally with some results v_1 and v_2 and the subsequently computed boolean is `true`; otherwise the test fails.
- A *ParenthesizedGuardTest* `(G)` succeeds if, and only if, the guard test `G` succeeds.

Note that if any subexpression of a guard test completes abruptly, the guard test (and thus the guard of which it is a part) fails. A guard test can never complete abruptly.

Output environment

The input and output environments of a guard test are the same.

6.20.2 Guard expressions

The guard expressions are syntactically identical with certain expressions.

GuardExpr:
  GuardAdditionShiftExpr

GuardAdditionShiftExpr:
  GuardAdditionShiftExpr AdditionOp GuardMultiplicationExpr
  GuardAdditionShiftExpr ShiftOp GuardMultiplicationExpr
  GuardMultiplicationExpr

GuardMultiplicationExpr:
  GuardMultiplicationExpr MultiplicationOp GuardPrefixOpExpr
  GuardPrefixOpExpr

GuardPrefixOpExpr:
  PrefixOp GuardApplicationExpr
  GuardApplicationExpr

GuardApplicationExpr:
    GuardBIF `(` *GuardExprs*$_{opt}$ `)`
    GuardRecordExpr
    GuardPrimaryExpr

GuardBIF:
    AtomLiteral

GuardExprs:
    GuardExpr
    GuardExprs `,` *GuardExpr*

GuardRecordExpr:
    GuardPrimaryExpr$_{opt}$ `#` *AtomLiteral* `.` *AtomLiteral*

GuardPrimaryExpr:
    Variable
    AtomicLiteral
    GuardListSkeleton
    GuardTupleSkeleton
    ParenthesizedGuardExpr

GuardListSkeleton:
    `[ ]`
    `[` *GuardExprs* *GuardListSkeletonTail*$_{opt}$ `]`

GuardListSkeletonTail:
    `|` *GuardExpr*

GuardTupleSkeleton:
    `{` *GuardExprs*$_{opt}$ `}`

ParenthesizedGuardExpr:
    `(` *GuardExpr* `)`

It is described in §13 which BIFs are guard BIFs.

Evaluation

All guard expressions are expressions and a guard expression is evaluated exactly as the corresponding expression. Evaluation of a guard expression may complete abruptly but a guard expression always occurs as part of a guard test which will restore normal evaluation by simply failing.

Output environment

The input and output environments of a guard expression are the same.

Chapter 7

Compiling a module

The unit of compilation in ERLANG is a module. The compilation of a module is carried out in the following steps:

1. Lexical processing, as described in §3. This takes a sequence of ASCII characters and produces a sequence of tokens and full stops, i.e., a sequence of *TokenSequence* (§3.18).

2. Preprocessing, as described in §7.1. This takes a sequence of *TokenSequence* and produces a new sequence of *TokenSequence*, after conditional compilation and macro expansion.

3. Parsing, as described in §7.5. Each *TokenSequence* in the sequence is parsed as an ERLANG form (§8). The result is a list of parse trees, which are represented as ERLANG terms (§B).

4. Parse transformation, as described in §7.6. If a parse transform function has been specified, it is applied to the list of parse trees, resulting in a new list of parse trees. Otherwise the result is the same list of parse trees.

5. Code generation, as described in §7.7. This takes a list of parse trees and returns a binary representation of a module, which can be loaded (§9).

The resulting binary can be loaded into a running ERLANG system or be saved in a file and loaded from there (§9.1).

7.1 Preprocessing

The preprocessing step takes a sequence of *TokenSequence* as input and produces a new sequence of *TokenSequence*. (A *TokenSequence* is a sequence of tokens followed by a full stop, cf. §3.18.)

During preprocessing, the compiler is in one of two modes: skipping or processing. When the compiler is in skipping mode, a *TokenSequence* is ignored unless it is one of the directives for conditional compilation discussed in §7.4. When the compiler is in processing mode, the treatment of a *TokenSequence* is as described in this section. The compiler is originally in processing mode.

Each *TokenSequence* that is one of the following directives is processed and is not part of the output sequence of *TokenSequence*.

Directive:	
MacroDefinition	(§7.2.1)
MacroUndefinition	(§7.2.2)
IncludeDirective	(§7.3)
IncludeLibDirective	(§7.3)
IfdefDirective	(§7.4)
IfndefDirective	(§7.4)
ElseDirective	(§7.4)
EndifDirective	(§7.4)

The directives `-define(`*M*`[(`V_1`,...,`V_k`)],`*Toks*`)` and `-undef(`*M*`)` maintain the set of macro definitions, the directives `-include(`*F*`)` and `-include_lib(`*F*`)` control file inclusion, and the directives `-ifdef(`*M*`)`, `-ifndef(`*M*`)`, `-else` and `-endif` control conditional compilation (§7.4).

The tokens in a *TokenSequence* that is not a *Directive* are subject to macro expansion (§7.2.3) and the resulting tokens, followed by a full stop, form a *TokenSequence* that is part of the output of the preprocessing.

7.2 Macros

As the preprocessor goes through the sequence of *TokenSequence*, it maintains a set of macro definitions. A *MacroDefinition* adds to the set (§7.2.1), a *MacroUndefinition* may remove from the set (§7.2.2), any other *TokenSequence* leaves it unchanged.

A *TokenSequence* that is not recognized as a *Directive* (§7.1) is subject to macro expansion (§7.2.3).

The initial set of macro definitions is described in §7.2.4.

7.2.1 Macro definition

A macro definition is a directive that adds a macro definition.

MacroDefinition:
  `- define (` *MacroName* *MacroParams*$_{opt}$ `,` *MacroBody* `)` *FullStop*

MacroName:
  AtomLiteral
  Variable

MacroParams:
  `(` *Variables*$_{opt}$ `)`

Variables:
  Variable
  Variables `,` *Variable*

MacroBody:
  Tokens

A macro definition `-define(`*`M`*`,`*`Toks`*`)` associates the macro name *`M`* with no sequence of parameters and an arbitrary (and possibly empty) sequence of tokens *`Toks`*.

A macro definition `-define(`*`M`*`(`V_1`, ... ,`V_k`),`*`Toks`*`)`, where $k \geq 0$, associates the macro name *`M`* with a (possibly empty) sequence of macro parameters V_1, ... , V_k which must be distinct variables, and a (possibly empty) sequence of tokens *`Toks`*.

Unlike the case for function names, which associate a symbol and an arity with a function, a macro name is simply a symbol and the macro named *`M`* either takes no parameters at all or obtains the arity k above. It is thus not possible to define two macros with the same name *`M`* but different arities.

In either case, the scope of the association for *`M`* begins immediately after the macro definition. It is a compile-time error if a macro definition of *`M`* occurs in the scope of another macro definition of *`M`*.

Note the difference between the macro definitions `-define(abc,123)` and `-define(abc(),123)`. The former is associated with no parameters while the latter is associated with an empty sequence of parameters and an application of it must therefore have an empty argument sequence (§7.2.3).

In ERLANG 4.7.3, a macro name `'Foo'` and a macro name `Foo` denote different macros. This is to be viewed as a bug (not a feature) and will change in a future version so a macro name that is a quoted atom and a macro name that is a variable denote the same macro if the print name of the atom consists of the same sequence of characters as the variable. For example, the macro names `'Foo'` and `Foo` will then denote the same macro.

7.2.2 Macro undefinition

The scope of a macro definition is terminated by a macro undefinition or the end of the module definition.

MacroUndefinition:
  `- undef (` *MacroName* `)` *FullStop*

Beginning immediately following an undefinition `-undef(M)`, the macro name `M` is undefined. It is not an error if `M` was undefined also immediately preceding the undefinition.

7.2.3 Macro expansion

Each macro application in a *TokenSequence* is expanded, i.e., replaced with tokens according to the macro definition that is in force at that point.

MacroApplication:
  `?` *MacroName*
  `?` *MacroName* `(` *MacroArguments*$_{opt}$ `)`

MacroArguments:
  MacroArgument
  MacroArguments `,` MacroArgument

MacroArgument:
  BalancedExpr that is not one of `,` or `)`

BalancedExpr:
  `(` *BalancedExprs* `)`
  `[` *BalancedExprs* `]`
  `{` *BalancedExprs* `}`
  `begin` *BalancedExprs* `end`
  `if` *BalancedExprs* `end`
  `case` *BalancedExprs* `end`
  `receive` *BalancedExprs* `end`
  `query` *BalancedExprs* `end`
  OtherToken
  BalancedExpr *BalancedExpr*

Macro expansion of a *TokenSequence* is defined as follows. Let us write $\langle T_1\ T_2\ \ldots\ T_k\rangle$ for the macro expansion of a *TokenSequence* $T_1\ T_2\ \ldots\ T_k$. We write τ or τ' for an arbitrary sequence of tokens and $\beta_1, \ldots, \beta_k$ for k balanced expressions, i.e., *BalancedExpr* above.

- $\langle$`? A` τ *FullStop*$\rangle = \langle\tau'\ \tau$ *FullStop*$\rangle$ if `A` is an atom and there is a macro definition for `A` with no parameters and a replacement sequence τ'.

- $\langle$`? A (` β_1 `,` $\ldots$ `,` β_k `)` τ *FullStop*$\rangle =$
  $\langle\tau'[\beta_1/v_1, \ldots, \beta_1/v_k]\ \tau$ *FullStop*$\rangle$
  if `A` is an atom and there is a macro definition for `A` with k parameters $v_1, \ldots, v_k$ and a replacement sequence τ'.

- $\langle$`?` τ *FullStop*$\rangle$ in any other case is a compile-time error.

- $\langle$`T` τ *FullStop*$\rangle$ = `T` $\langle\tau$ *FullStop*$\rangle$ where `T` is any token but `?`.
- $\langle$*FullStop*$\rangle$ = *FullStop*.

To summarize in words, after `?` must follow a defined macro name. Either it is defined to have no parameters, in which case `?` and the atom are replaced by the expansion and macro expansion starts over from there, or it is defined to have a number of parameters in which case `?` and the atom must be followed by that many macro arguments. In that case `?`, the atom and the macro arguments are replaced by the expansion in which each occurrence of a macro parameter has been replaced by the corresponding macro argument and macro expansion starts over from there. If the first token is not `?`, it is made part of the result of the macro expansion.

For example, suppose the following macro definitions are in force:

```
-define(foo,fum(?).
-define(bar(X),(X))).
```

They associate the macro name `foo` with no parameters and the three tokens `fum`, `(` and `?`, and the macro name `bar` with one macro parameter `X` and an expansion consisting of `(`, the token sequence given as macro argument, `)` and `)` again. (This would be an extremely objectionable programming style.) Then the *TokenSequence*

```
foo(X) -> ?foo bar(6*X).
```

which consists of the thirteen tokens `foo`, `(`, `X`, `)`, `->`, `?`, `foo`, `bar`, `(`, `6`, `*`, `X` and `)`, followed by a full stop, is macro expanded to

```
foo(X) -> fum((6*X)).
```

in the following steps:

1. $\langle$`foo ( X ) -> ?  foo bar ( 6 * X )` *FullStop*$\rangle$
   The first five tokens (including the atom `foo` even though there happens to be a macro with that name) are unaffected by macro expansion.

2. `foo ( X ) ->` $\langle$`?  foo bar ( 6 * X )` *FullStop*$\rangle$
   The next two tokens are the separator `?` and the atom `foo`. There is a macro definition of `foo` with no parameters and expansion consisting of the tokens `fum`, `(` and `?`; thus `?` and `foo` are replaced by `fum`, `(` and `?` and macro expansion starts over from there.

3. `foo ( X ) ->` $\langle$`fum ( ?  bar ( 6 * X )` *FullStop*$\rangle$
   The next two tokens are unaffected by macro expansion.

4. `foo ( X ) -> fum (` ⟨`?  bar ( 6 * X )` *FullStop*⟩
   The next two tokens are `?` and `bar` and there is a macro definition of `bar` with one macro parameter. The macro argument consists of the tokens `6`, `*` and `X`. The tokens `?`, `bar`, `(`, `6`, `*`, `X` and `)` are then replaced with the expansion `(`, `6`, `*`, `X`, `)`, `)` and macro expansion starts over from there.

5. `foo ( X ) -> fum (` ⟨`( 6 * X ) )` *FullStop*⟩
   The remaining six tokens are unaffected by macro expansion.

6. `foo ( X ) -> fum ( ( 6 * X ) )` *FullStop*
   Macro expansion is complete.

Note that directives are not macro expanded (and their arguments are not evaluated). For example, it is not possible to write

```
-define(SRCDIR(FN),"/usr/local/src/myproj/" FN ".erl").

-include(?SRCDIR("bliss")).
```

because the argument of `include` must be an *IncludeFileName*, i.e., a *OneStringLiteral.*

7.2.4 Initial set of macro definitions

Initially the following three macros are defined, each one associated with no parameters:

- `?MODULE` is expanded to a single token: an atom literal that is the name of the module being compiled.

- `?FILE` is expanded to a single token which is a string literal that is a full path to the file that is being compiled.

- `?LINE` is expanded to a single token which is an integer literal that is the number of the line on which the `LINE` token appears. If `LINE` occurs within a macro then `LINE` is expanded to the number of the line in which that macro is expanded. Undefining or redefining the `LINE` macro has no effect. (Note that `LINE` does not have a fixed token sequence and has to be handled as a special case in the macro expansion.)

- `?MACHINE` is expanded to a single token: an atom literal that is the name of the abstract ERLANG machine on which the compiler is running. The known abstract ERLANG machines are:

  ∗ `'JAM'`

  ∗ `'BEAM'`

  ∗ `'VEE'`

7.3 File inclusion

The `include` and `include_lib` directives splice in the contents of a file.

IncludeDirective:
  `- include (` *IncludeFileName* `)` *FullStop*

IncludeLibDirective:
  `- include_lib (` *IncludeFileName* `)` *FullStop*

IncludeFileName:
  OneStringLiteral

The difference between the `include` and `include_lib` directives lies in which paths are searched when the given filename is not absolute. For `-include(F)`, file `F` is searched in each directory for which an include path was given to the compiler (option `{i,Dir}`), in order (as if using `file:path_open/3` [6, p. 230]). Also for `-include_lib(F)`, these paths are tried first. However, if the file is not found, then if `F` can be split into a string `L` that does not contain the character '`/`' and one string `N` that begins with '`/`', `L` is looked up as a library (i.e., as if it was given as argument to `code:lib_dir/1` [6, p. 164]). If that yields a path `P` to a library directory then the path which is the concatenation of `P` and `N` is used.

It is a compile-time error if no readable file is found.

The contents of the file is subject to lexical processing (§3). It is an error if lexical processing of the contents of the included file does not yield a *TerminatedTokens*, i.e., a sequence of *TokenSequence.*

The resulting sequence of *TokenSequence* is processed exactly as described in this section. While processing the included file, the `FILE` and `LINE` macros (§7.2.4) refer to the path of the included file and line numbers in it, respectively. The sequence of *TokenSequence* that is the result of the preprocessing of the included file becomes part of the output of the preprocessing of the including sequence of *TokenSequence* at the location of the `include` or `include_lib` directive.

7.4 Conditional compilation

There are four directives related to conditional compilation and two additional directives for which syntax is reserved.

IfdefDirective:
  `- ifdef (` *MacroName* `)` *FullStop*

IfndefDirective:
  `- ifndef (` *MacroName* `)` *FullStop*

ElseDirective:
  `- else` *FullStop*

EndifDirective:
  `- endif` *FullStop*

It is a compile-time error if the sequence of *TokenSequence* given to the preprocessing step does not contain matching triples or pairs of

- *IfdefDirective*, *ElseDirective* and *EndifDirective*;
- *IfndefDirective*, *ElseDirective* and *EndifDirective*;
- *IfdefDirective* and *EndifDirective*;
- *IfndefDirective* and *EndifDirective*.

This must hold for a whole module definition as well as individually for each included file (§7.3).

If the compiler is in processing mode and encounters an `-ifdef(`*M*`)` directive [or `-ifndef(`*M*`)` directive], the compiler tests whether there is a macro definition for *M*. If this is [not] the case, then the compiler continues in processing mode until it encounters the matching `else` or `endif` directive. (Note that this may involve handling any number of enclosed nested triples or pairs of the kinds above.) If an `else` directive was encountered, the compiler continues in skipping [processing] mode until the matching `endif` directive is encountered. In either case, after the `endif` directive, the compiler continues in processing mode.

If the compiler is in skipping mode and encounters an `-ifdef(`*M*`)` or `-ifndef(`*M*`)` directive, it continues in skipping mode until it encounters the matching `else` or `endif` directive. (Again, this may involve handling any number of enclosed nested triples or pairs of the kinds above.) If an `else` directive was encountered, the compiler continues in skipping mode until the matching `endif` directive is encountered. In either case, after the `endif` directive, the compiler continues in skipping mode. Thus, it is obligatory for the compiler to keep track of the directives for conditional compilation also when in skipping mode. For example, the compiler must report a compile-time error for a module containing the following directives:

```
-define(foo,42).
-ifdef(foo).
-else.
-else
-endif.
```

The compiler may implement this mechanism in any suitable way, but the following machinery serves as an example and may clarify the description above. Let the compiler have a state consisting of a register `Processing` and a stack. `Processing` is either `true` or `false`. Each stack item is a pair where the left half is either `if` or `else` and the right half is either `true` or `false`. Initially `Processing` is `true` and the stack is empty. Here is what happens when the compiler encounters a *TokenSequence*:

- If the *TokenSequence* is `-ifdef(`*M*`)` or `-ifndef(`*M*`)`, then a pair of `if` and the value of `Processing` is pushed onto the stack. If `Processing` is `true` and either the *TokenSequence* is `-ifdef(`*M*`)` and *M* has no macro definition, or the *TokenSequence* is `-ifndef(`*M*`)` and *M* has a macro definition, then `Processing` is changed to `false`.

- If the *TokenSequence* is `-else` then it is a compile-time error if the stack is empty or the left half of the top pair is not `if`. The left half of the top pair is changed to `else` and the contents of `Processing` is set to the logical conjunction of the right half of the top pair and the negation of `Processing`.

- If the *TokenSequence* is `-endif` then it is a compile-time error if the stack is empty. `Processing` is set to the right half of the top stack pair and that pair is popped off the stack.

- If the *TokenSequence* is anything else, then if `Processing` is `true`, it is processed as described in §7.1, otherwise it is ignored.

The directives `if` and `elif` (with any number of arguments) are reserved for future extension of conditional compilation.

7.5 Parsing

The sequence of *TokenSequence* is parsed as a *ModuleDeclaration* (§8.1). The result is a list of parse trees (one for each top-level form), which are represented as Erlang terms (§B).

7.6 Parse transforms

A parse transform is a function mapping lists of parse trees to lists of parse trees. Each parse transform is implemented by a separate module with an exported function named `parse_transform/1`. Suppose that the compiler has been instructed (through compiler options) to use the k parse transforms implemented by modules $M_1, \ldots, M_k$. (The order is relevant and is the same as the order in which the compiler options appear.)

Let L_0 be the list of parse trees resulting from parsing the preprocessed sequence of *TokenSequence*, represented as Erlang terms (§B). For each i, $1 \leq i \leq k$, let L_i be the result of computing

M_i`:parse_transform(`L_{i-1}`)`

if that computation completes normally. The result must be a list of parse trees for a *ModuleDeclaration* (just as L_0 is). If for some M_i, the computation completes abruptly or the result of the computation is not a list of parse trees for a *ModuleDeclaration*, then it is a compile-time error. Otherwise, L_k is used in the code generation step (§7.7).

7.7 Code generation

The code generation step takes a list of parse trees represented as ERLANG terms (§B) and produces a binary that contains a loadable representation of the module. That binary can either be loaded immediately (§9.1) or be written to a file for later loading. The actual format of the binary is implementation-defined.

Chapter 8

Programs and modules

An ERLANG *module declaration* consists of a collection of *forms*. Forms are function declarations together with some attributes and record declarations for the module. It is the smallest unit of code that can be separately compiled and loaded. An ERLANG compiler thus has access to all the code of a module at compile-time. Therefore it can determine at compile-time, for example, the names of all functions being declared in the module.

An ERLANG *program* typically consists of a number of modules, out of which some may be standard ERLANG modules or parts of library applications [6].

8.1 Module declarations

A module declaration may begin with one or more file attributes, as described in §8.2.4.

After those must follow a module attribute `-module(Name)`, in which the argument `Name` must be an atom. The module attribute states the name of the module.

The module attribute is followed by a *header part*, consisting of a (possibly empty) sequence of header forms, which are additional attributes and declarations of the module.

Finally there is a *code part*, consisting of a (possibly empty) sequence of *program forms*: function declarations, possibly interspersed with such attributes and declarations that need not be in the header part.

ModuleDeclaration:
    FileAttributes$_{opt}$ *ModuleAttribute* *HeaderForms*$_{opt}$ *ProgramForms*$_{opt}$

FileAttributes:
    FileAttribute
    FileAttributes *FileAttribute*

ModuleAttribute:
  `- module (` *ModuleName* `)` *FullStop*

ModuleName:
  AtomLiteral

There is a restriction in the file compiler that a file named ***Mod*.erl** must contain the full source code of an ERLANG module named ***Mod***. (This file may however include other files containing header forms, cf. §8.2.) This restriction may be lifted or altered in a future version of ERLANG.

8.2 The header part

The header part is a sequence of header forms, which are *header attributes* or *anywhere attributes*. The former may only appear in the header part while the latter may also appear in the code part.

HeaderForms:
  HeaderForm
  HeaderForms HeaderForm

HeaderForm:
  HeaderAttribute
  AnywhereAttribute

HeaderAttribute:
  ExportAttribute
  ImportAttribute
  CompileAttribute
  WildAttribute

AnywhereAttribute:
  FileAttribute
  MacroDefinition
  RecordDeclaration

Macro definitions are defined in §7.2.

The remaining attributes (both header attributes and others) are described in the following sections.

8.2.1 Export attributes

An export attribute is syntactically like an application of a (hypothetical) function `export/1` to a list of *function names*, preceded by a dash. Each function name consists of a function symbol and an arity numeral, separated by `/`.

ExportAttribute:
  `- export (` *FunctionNameList* `)` *FullStop*

FunctionNameList:
  `[` $FunctionNames_{opt}$ `]`

FunctionNames:
  FunctionName
  FunctionNames `,` *FunctionName*

FunctionName:
  FunctionSymbol `/` *Arity*

FunctionSymbol:
  AtomLiteral

Arity:
  IntegerLiteral

There may be more than one export attribute among the header forms. The order between functions names in an export attribute is irrelevant. The same function name may appear more than once in an export attribute and in several export attributes.

If there is an export attribute in module *M* that lists the function *F/A*, then the function with symbol *F* and arity *A* in module *M* can be applied using a remote application (§6.7).

Note that if there is *not* an export attribute in module *M* that lists the function *F/A*, then the function cannot be applied through a remote application even in the code for module *M*.

It is a compile-time error if there is an export attribute listing a function *F/A* but no declaration of a function *F/A*.

8.2.2 Import attributes

An import attribute is syntactically like an application of a (hypothetical) function `import/1` to a module name and a list of function names (cf. §8.2.1), preceded by a dash.

ImportAttribute:
  `- import (` *ModuleName* `,` *FunctionNameList* `)` *FullStop*

Import attributes have the following role. If there is an import attribute in module *M* that lists the function *F/A* of module M', then a seemingly local function application $F(E_1, \ldots, E_A)$ in module *M* is syntactic sugar for a remote function application $M':F(E_1, \ldots, E_A)$ (unless there is a BIF named *F/A*, cf. §6.18).

There may be more than one import attribute among the header forms. The order between import attributes and between functions names in an import attribute is irrelevant. The same function name may appear more than once in an import attribute and in several import attributes for the same module.

It is a compile-time error if in the same module there are import attributes listing the same function name but different modules.

If in some module there is both an import attribute listing a function *F*/*A* and a function declaration for *F*/*A*, then all applications of the form *F*(E_1,...,E_A) in the module will refer to the imported function. However, if there is an export attribute listing *F*/*A*, it refers to the function being declared. For example, consider the following module:

```
-module(expimp).

-import(lists,[sort/1]).

-export([sort/1]).

sort(X) -> sort(X).
```

The declaration of the function `sort/1` may appear to be recursive but it is not: the application `sort(X)` to the right of '`->`' calls the imported function. However, the export attribute refers to the function being declared in module `expimp`. (Here the function being declared simply returns the result of the imported function but if the module is later replaced by a new version, the function `sort/1` may be defined differently.)

8.2.3 Compile attributes

A compile attribute is syntactically like an application of a (hypothetical) unary function `compile` to a list of terms.

CompileAttribute:
  `- compile ( [` *Terms*$_{opt}$ `] )` *FullStop*

The terms in the list will be appended at the end of the list of options given to the `compile:file/2` function. The terms must therefore be acceptable as options for the ERLANG compiler being used, or compilation will fail.

The compiler options are implementation-defined.

8.2.4 File attributes

A file attribute is syntactically like an application of a (hypothetical) function `file/2` to a string literal (presumably a file name although it is not an error if no such file exists) and a line numeral, preceded by a dash.

FileAttribute:
  `- file (` *StringLiteral* `,` *LineNumeral* `)` *FullStop*

LineNumeral:
  IntegerLiteral

The purpose of a file attribute `-file(`*F*`,`*L*`)` is to inform the compiler that the code being compiled originated from line *L* in the file *F* so that error messages can refer to the proper place in the original file.

8.2.5 Wild attributes

A wild attribute is syntactically like an application of a unary function other than `export/1`, `import/1`, `file/2`, `compile/1`, `type/1`, `deftype/1` and `record/1` to a term.

WildAttribute:
  `-` *AtomLiteral* `(` *Term* `)` *FullStop*

The compiler gathers all wild attributes for a module *M* and at runtime, a BIF call *M*`:module_info(attributes)` returns a list of them (cf. §8.5.2).

8.3 Program forms

The program forms are *function declarations* and such attributes that may appear anywhere in a module declaration. (For unambiguity of the grammar, it is required that the *ProgramForms* begins with a *FunctionDeclaration* but there is no significant difference between an *AnywhereAttribute* that appears as part of the *HeaderForms* and one that appears as part of the *ProgramForms*.)

ProgramForms:
  FunctionDeclaration
  ProgramForms FunctionDeclaration
  ProgramForms AnywhereAttribute

FunctionDeclaration:
  FunctionClauses FullStop

FunctionClauses:
  FunctionClause
  FunctionClauses ; *FunctionClause*

FunctionClause:
  FunctionSymbol FunClause

FunClause:
    `(` *Patterns*$_{opt}$ `)` *ClauseGuard*$_{opt}$ *ClauseBody*

(The rule for *FunClause* is repeated from §6.19.10 for convenience.)

Each function declaration consists of one or more *function clauses*, separated by semicolons. Every clause of a function declaration must begin with the same *AtomLiteral* and the sequence of arguments following it must have the same length in every clause. The *AtomLiteral* is the function symbol and together with the length of the argument sequences, which is the arity of the function, it forms the *function name.* A function declaration for a function named `F/A` is thus on the form

$F(P_{1,1}, \ldots, P_{1,A})$ [`when` G_1] `->` B_1 ;
$\vdots$;
$F(P_{k,1}, \ldots, P_{k,A})$ [`when` G_k] `->` B_k .

where the guard part of each function clause is optional.

It is a compile-time error if in a module there is more than one declaration for a function name.

Application of functions is defined in §6.18.

8.4 Record declarations

RecordDeclaration:
    `- record (` *RecordType* `,` *RecordDeclTuple* `)` *FullStop*

RecordDeclTuple:
    `{` *RecordFieldDecls*$_{opt}$ `}`

RecordFieldDecls:
    RecordFieldDecl
    RecordFieldDecls `,` *RecordFieldDecl*

RecordFieldDecl:
    RecordFieldName *RecordFieldValue*$_{opt}$

(The rules for *RecordType* and *RecordFieldName* appear in §6.6 while the rules for *RecordFieldValue* appear in §6.17.)

In a record declaration

`-record(`R`,{`F_1[`=`E_1], ..., F_n[`=`E_n]`})`

the field names $F_1, \ldots, F_n$ must all be distinct and the (optional) default initializer expressions $E_1, \ldots, E_n$ will be evaluated in an empty environment and must have an empty output environment. It should be a compile-time error if a default initializer expression contains a free variable.

The scope of the record declaration begins immediately after its lexical occurrence and ends at the end of the module declaration. In a module there must be at most one record declaration for each record type. It is possible for two modules that are part of the same application to have incoherent record declarations. This could lead to severe problems. Similar problems may appear, unless caution is taken, when loading a new version of a module if its record declarations have changed (because there may be existing records created according to the old record declarations).

The record declaration establishes R as the name of a record type having n fields named F_1, … , F_n. The optional default initializer expression E_i is used when a record is created (§6.17.3) and no value is specified for the field named F_i. (If no default initializer expression is given either, then the atom `undefined` is used.) For example, a record declaration

```
-record(music,{title,artist,medium=cd,number})
```

declares `music` to be a record type and that each record term of type `music` has four fields named `title`, `artist`, `medium` and `number`. An expression

```
#music{}
```

creates a record in which these fields have the values `undefined`, `undefined`, `cd` and `undefined`, respectively, while an expression

```
#music{title="The Dark Side of the Moon",
       artist="Pink Floyd",
       medium=lp}
```

creates a record in which the fields have the values `"The Dark side of the Moon"`, `"Pink Floyd"`, `lp` and `undefined`, respectively (the default initializer expression `cd` for `medium` was overridden).

8.5 The module information functions

When compiling a module, the compiler automatically adds declarations for two functions `module_info/0` and `module_info/1` in it.

8.5.1 The function `module_info/0`

In any module M, the compiler automatically adds a declaration of a function `module_info/0`, such that an application `M:module_info()` returns an association list (cf. §4.9.2). This association list is such that for every key K that the function `module_info/1` accepts, except `module`, the list contains a 2-tuple `{K,M:module_info(K)}` and no other 2-tuples.

8.5.2 The function `module_info/1`

In any module *M*, the compiler adds a declaration for an exported function `module_info/1` such that for certain terms, the function returns a value. It must hold that:

- *M*`:module_info(module)` returns *M* (i.e., the name of the module as an atom).
- *M*`:module_info(exports)` returns a list of 2-tuples such that for every function name *F*`/`*A* that appears in an export attribute, the list contains a 2-tuple `{`*F*`,`*A*`}`. That is, the first element of each 2-tuple is an atom and the second element is an integer. There are no other 2-tuples in the list and the order between the 2-tuples is undefined.
- *M*`:module_info(imports)` returns a list of 2-tuples such that for every function name *F*`/`*A* that appears in an import attribute for a module *M'*, the list contains a 2-tuple `{{`*F*`,`*A*`},`*M'*`}`. That is, the first element of each 2-tuple is itself a 2-tuple of an atom and an integer, and the second element of each pair is an atom. There are no other 2-tuples in the list and the order between the 2-tuples is undefined.
- *M*`:module_info(attributes)` returns an association list such that for every wild or `compile` attribute `-`*K*`(`*T*`)` of module *M*, the list contains a 2-tuple `{`*K*`,`*T*`}`. That is, the first element of each 2-tuple is an atom and the second element is an arbitrary term. There are no other 2-tuples in the list and the order between the 2-tuples is undefined.
- *M*`:module_info(compile)` returns an association list with information about the compilation of module *M*. The association list must at least map the atom `time` to the time when compilation of module *M* began (represented as a six-tuple of integers: year, month, day, hours, minutes and seconds, like the elements in the result of the BIF `date/0` [§13.13.1] followed by the elements in the result of the BIF `time/0` [§13.13.6]) and the atom `options` to the list of options that were given to the compiler (including those provided through `compile` directives [§8.2.3]).

An implementation may let *M*`:module_info/1` accept additional atoms as argument (in which case the association list returned by *M*`:module_info/0` should be extended accordingly, cf. §8.5.1).

Chapter 9

Dynamics of modules

Erlang has been designed to make it possible to incorporate functionality for replacing a version of a module with a new version of that module, even though at the same time there are processes executing the old version of the module.

An example of such functionality is provided by a collection of BIFs: `load_module/2`, `delete_module/1`, `purge_module/1`, etc., which are described in this chapter.[1]

A process that is evaluating an application of a function in the old version of module can finish the evaluation of that application even though a new version of the function has been loaded. However, there can be only one "old version" of a module so it is presumed that the Erlang code of the modules has been written so that processes will begin using the new version of the module as soon as possible.

As soon as a new version of a module named *`Mod`* has been loaded, evaluation of remote applications of functions in module *`Mod`* will use the new version of module (§9.3).

This implies, for example, that a tail recursive function which is intended to run for a long time should typically use a remote function application for the tail recursive call. The tail recursive call will then use the most recently loaded version of the module. (Note also that the *last call optimization* [§6.7.3] entails that evaluation of such a tail recursive function can run using stack space bounded only by the need for each iteration.)

Obviously, the programmer must be aware of the possibility that modules may be replaced by new versions and design the code so it will cope gracefully with such replacements.

[1]These BIFs are, in turn, used for implementing the `code` module of OTP [6, p. 158–167].

9.1 Loading or replacing a module

A *version* of a module named `Mod` is the result of compiling a *ModuleDeclaration* with a module attribute `-module(Mod)`, represented as a binary (§7.7). (Module declarations are described in §8 and their compilation in §7.)

The *current version* of a module named `Mod` on a node `N` is the one for which there are rows in `entry_points[N]` (§9.3). the current version of `Mod` on `N` is accessible as `current_version[module_table[N](Mod)]` (§9.2).

A version of a module named `Mod`, represented as a binary `B`, is the *current version* on node `N` from the time it has been loaded on node `N` (§9.4) until it is made old on node `N` (§9.5).

The *old version* of a module named `Mod` on a node `N` has all its code intact on `N` but there are no longer any rows for it in `entry_points[N]` so no new remote function calls can reach code in it. The version is the old version until it is purged (§9.6).

When we write that a process `P` uses some version of a module `Mod`, represented as a binary `B`, we mean that `P` is using some exported function `F/A` in `B` (§6.7.3, §9.7).

When a version of a module named `Mod` is to be replaced with a new version on an ERLANG node `N`, the following sequence of steps is intended to be followed:

- The current version of module `Mod` on node `N`, if any, is made the old version (§9.5).

- A version of module `Mod` is made the current version of `Mod` on node `N` (§9.4).

- When it has been ensured that no process on node `N` is using the old version of module `Mod` anymore, the old version of `Mod` is purged (§9.6). This reclaims the space used on node `N` by the old version of the code.

  Ensuring that no process is using the old version of a module can be accomplished in several ways, e.g.:

  ∗ Waiting until all such processes complete or no longer use the old version of the code (cf. §9.7).

  ∗ Killing such processes.

  ∗ Designing the code of the module so that a process using it can be sent a message that makes it prepare for a version change by, e.g., completing or evaluating a remote tail recursive call.

9.2 Loaded modules

At any time, a module is *loaded* on node *N* if there is a current version of *Mod* on *N*, i.e., if `current_version[module_table[N](Mod)]` is a binary.

§9.4 and §9.5 describe how `current_version[module_table[N](Mod)]` is set to a binary or to `none`, respectively.

A process residing on node *N* can find out whether module *Mod* is loaded on node *N* through a BIF call `erlang:module_loaded(Mod)`, which inspects `current_version[module_table[N](Mod)]` and returns `true` if the value is a binary and `false` if it is `none` (§13.7.6).

9.3 Exported functions of a module

Each node maintains a table `entry_points[N]` (§11.7.2) in which the keys are triples consisting of a module name (an atom), a function symbol (an atom) and an arity (a nonnegative integer) and the values are entry points to executable code. The table contains one row for each exported function of each loaded module.

§9.4 and §9.5 describe how rows are are added to and removed from `entry_points[N]`, respectively.

The table `entry_points[N]` is used implicitly when evaluating remote function applications (§6.7.1). Only while a version *B* of a module *Mod* is current on a node *N* can a remote call of an exported function *F/n* in *B* begin.

9.4 Loading a new current version

When a binary *B* is to be made the current version of a module *Mod* on a node *N*, there are two preconditions:

- *B* must represent the result of compiling a module declaration for a module *Mod*, and
- there must be no current version of module *Mod* on node *N*, i.e., `current_version[module_table[N](Mod)]` should be `none`.

The actions required to make *B* the current version are

1. For each exported function *Name/Arity* of the version of *Mod* represented by *B*, add to `entry_points[N]` a row with (*Mod*, *Name*, *Arity*) as key and an entry point of the executable code for that function as value.
2. Set `current_version[module_table[N](Mod)]` to *B*.

9.5 Making a current version old

When the current version of a module *Mod* on a node *N* is to become the old version of that module on the node, the precondition is that there is not already an old version, i.e., that `old_version[module_table[N](Mod)]` is `none`.

Let the value of `current_version[module_table[N](Mod)]`, i.e., the current version of *Mod* on *N* be *B*. The actions required to make *B* the old version of *Mod* on *N* are

1. Remove every row from `entry_points[N]` that has *Mod* as key.
2. Set `old_version[module_table[N](Mod)]` to *B*.
3. Set `current_version[module_table[N](Mod)]` to `none`.

9.6 Purging an old version

When the old version of a module *Mod* on a node *N* is to be purged, there had better be no process using it. If some process residing on *N* is using the old version of *Mod*, the behaviour of that process is thereafter undefined.

The action required to purge the old version of *Mod* on *N* is setting `old_version[module_table[N](Mod)]` to `none`. If that was the last reference to the binary that was the old version of *Mod* on *N*, then the memory management subsystem (§4.13) will eventually reclaim the memory occupied by it. (The reason that processes still using the old version may behave erratically is that it cannot be expected that their references through return addresses will prevent the binary from being "garbage collected".)

9.7 Checking a process for module usage

A BIF call `erlang:check_process_code(P,Mod)` (§13.7.1) inspects the value of `stack_trace[P]` and returns `true` if there is a reference to the code of some function in `old_version[module_table[node[P]](Mod)]` and `false` otherwise.

A list of the PIDs of all processes residing on node *N* can be obtained through a BIF call `processes()` (§13.10.9) on node *N*.

Chapter 10

Processes and concurrency

10.1 An overview of Erlang processes

An ERLANG *process* is an entity that exists for a certain time, is evaluating a function application, has a state and is able to communicate with other processes during its lifetime. It may complete normally or abruptly.

Each process is associated with a unique term, called its *process identifier* or *PID*. The PID of a process is used for referring to the process, for example, when communicating with it. As there is a one-to-one correspondence between processes and PIDs, we will often abuse our terminology and write about the PIDs as if they were the actual processes. A process can obtain its own PID using the BIF `self/0` (§13.8.14).

Normally a function application appears in a program through an expression on the form $F(T_1,\dots,T_k)$ or $M\!:\!F(T_1,\dots,T_k)$ (or through an application of one of the BIFs `apply/2` and `apply/3`). In the normal mode of evaluation, the expression is then evaluated as part of the same process until it completes normally, in which case the result is the value of the expression, or it completes abruptly, in which case the enclosing expression also completes abruptly.

Through the BIFs `spawn/3`, `spawn/4`, `spawn_link/3` and `spawn_link/4` (§13.8) it is possible to request instead that the application is evaluated in a process of its own. An application `spawn(`M`,F,[`$E_1,\dots,E_k$`])` has the effect of spawning a new process that evaluates the application $M\!:\!F(E_1,\dots,E_k)$, but does not wait for its value to be computed. The value of the expression `spawn(`M`,F,[`$E_1,\dots,E_k$`])` is instead the PID of the new process. The value of $M\!:\!F(E_1,\dots,E_k)$ cannot be accessed even though it is computed (when the computation completes normally) so it is obvious that any result of a process must be made available through communication.

Let the values of the expressions M, F, E_1, ... , E_k be Mod, Fun, v_1, ... , v_k. The call of the function $Mod\!:\!Fun/k$ to the values v_1, ... , v_k is known as the *initial call* of the process.

The BIF `spawn/3` spawns the new process on the same node as the process evaluating the application while `spawn/4` spawns the new process on a specified node (§11). The BIFs `spawn_link/3` and `spawn_link/4` are like `spawn/3` and `spawn/4`, respectively, but link the spawning and spawned processes (§10.3).

Processes communicate through *signals*. The two kinds of signals that are immediately noticeable for ERLANG programmers are *messages* and *exit signals*. Both messages and exit signals are terms but they are sent and received differently and with different purposes.

An exit signal is automatically sent upon completion of a process and it may cause abrupt completion of other processes, as described in §10.4.

A message is sent by evaluating a *send expression*, i.e., an application of the `!` operator (§6.11). Correspondingly, evaluating a `receive` expression (§6.19.9) normally receives one message and dispatches on its form. The mechanism for communication by messages is described in §10.5.

10.2 Process names

Each node maintains a registry of names of processes, providing a level of abstraction when referring to a process. These names are atoms and can be used instead of the PID when sending a message to a process. It is also possible to retrieve the current PID that a name is registered to stand for.

Process names are described in more detail elsewhere (§11.5) as are the BIFs `register/2`, `whereis/1`, `unregister/1` and `registered/0` (§13.11).

10.3 Linked processes

A pair of processes P_1 and P_2 may be *linked*, which means that when P_1 completes, an exit signal (§10.4) is sent to P_2, and vice versa. Receiving the exit signal can either cause P_2 to complete abruptly or notify P_2 of the completion of P_1 in the form of a message (§10.5). Links are completely symmetric so there is no distinction between the linked processes. A process may link to itself but that has no effect.

Note that when a process P_1 is linked to two or more processes, say P_2 and P_3, then if P_2 completes abruptly, the exit signal received by P_1 will cause abrupt completion also of P_1 (unless it is trapping exits), which will cause an exit signal to be sent to P_3, etc. The abrupt completion of P_2 thus causes abrupt completion of P_3 even though the processes were not linked directly. In this way, abrupt completion of one process in a collection of linked processes may cause abrupt completion of several or all processes in the group.

A link can be created between two processes P_1 and P_2 in the following ways:

- If one of the processes spawned the other using the BIF `spawn_link/3` or `spawn_link/4`. In this case, the link is in effect as soon as the process has been created.

- If process P_1 calls the BIF `link/1` with P_2 as argument, or process P_1 calls it with P_2 as argument. In this case, setting up the link takes an unspecified amount of time. If the two processes are already linked when one of them evaluates an application of `link/1`, it has no effect. If when (say) P_1 evaluates `link(`P_2`)` the process P_2 has completed, then no link is created and P_1 eventually receives an exit signal `{'EXIT',`P_2`,`*noproc*`}` (§10.4).

  For details see the description of the BIF `link/1` (§13.8.7) and §10.6.3.

The link between two processes P_1 and P_2 is removed if process P_1 evaluates `unlink(`P_2`)` or process P_2 evaluates `unlink(`P_1`)`. It takes an unspecified amount of time before the link is removed. If the two processes are not linked when one of them evaluates an application of `unlink/1`, it has no effect. For details see the description of the BIF `unlink/1` (§13.8.19) and §10.6.3.

Obviously, two processes setting up or removing a link between each other may need to synchronize with messages to ensure the status of the link before further processing.

It is also possible for a process and a port to be linked, as described in §12.9.

10.4 Completion of processes and exit signals

In this section we describe what causes a process to complete and the exit signals and messages that are dispatched when that happens. We also describe how exit signals are sent using the BIF `exit/2` and what happens when exit signals are received.

10.4.1 Process completion

A process may complete for one of the following four reasons:

- The evaluation of the toplevel application of a process completes normally: *Reason* will be `normal`.

- The evaluation of the toplevel application of a process completes abruptly with exit reason *R*: *Reason* will be *R*.

- The evaluation of the toplevel application of a process completes abruptly with a thrown term: *Reason* will be `nocatch`.

- The process receives an exit signal that causes it to complete abruptly (§10.4.3), *Reason* will then be as specified in §10.6.3.

The BIF `exit/1` (§13.8.3) is typically used to make a function call exit with some reason, as some kind of error has been detected. Unless the code has been written to restore normal computation, the process executing the function call will terminate. That is, evaluation of an application `exit(`*Reason*`)` for some term *Reason* completes abruptly with the exit term *Reason* as reason and unless the evaluation is governed by a `catch` expression, the process completes as described above.

The BIF `throw/1` (§13.13.5) is intended for nonlocal control, not for signalling an error or making a process terminate.

When a process completes, an *exit signal* is sent to every process that is linked to it. The order in which these exit signals are sent is not defined. The exit signal contains the PID of the completing process and contains the exit reason.

More precisely, suppose that process P completes for one of the reasons above. For each process P' for which P' is in `linked[`P`]`, dispatch an exit signal with P as sender and *Reason* as reason.

If two processes P_1 and P_2 residing on different nodes (§11) are linked and the nodes lose contact, P_1 and P_2 will receive exit signals with reason `noconnection`, as if the other process had completed.

10.4.2 Sending exit signals explicitly

The BIF `exit/2` (§13.8.4) can be used to send exit signals to processes. If process P_1 evaluates the application `exit(`P_2`,`*Reason*`)`, an exit signal with reason *Reason* is sent to process P_2, similarly to when process P_1 is linked with P_2 and completes with exit reason *Reason*. However, if *Reason* is `kill`, the receiving process will always complete (§10.6.3).

Note that if P_1 is linked to P_2, and `trap_exit[`P_2`]` is `false` (or *Reason* is `kill`), then P_2 will terminate and an exit signal will subsequently be sent to P_1 itself.

For details, see the description of the BIF `exit/2` (§13.8.4).

10.4.3 Receiving an exit signal

We write that a process P is *trapping exits* if `trap_exits[`P`]` is `true`.

When an exit signal is received, one of three things happens:

- If the receiving process is trapping exits and the exit signal was not sent using the BIF `exit/2` with `kill` as reason, then the reason is placed in the message queue of the receiving process.

- Otherwise, if the reason is `normal`, nothing happens.

- Otherwise (the reason is not `normal` and either the process is not trapping exits or the exit signal was sent using the BIF `exit/2` with `kill` as reason), the process completes (§10.6.3).

For details, cf. §10.6.3.

10.5 Communication by messages

A message in ERLANG can be any ERLANG term. A message is sent by a *sending process* to a specific *receiving process*, identified by its PID, by a registered name, which is an atom, or by 2-tuple of a node name and a registered name.

Communication by messages in ERLANG is asynchronous, i.e., the process sending a message does not wait until the message is received. We shall therefore describe sending of messages and reception of messages in three steps:

- A message is dispatched from a process with some process as recipient.
- The message arrives at the message queue of the recipient.
- The recipient retrieves the message from its message queue.

10.5.1 Sending a message

The only primitive for sending a message in ERLANG is the operator `!`. An expression E_1 `!` E_2 is called a *send expression* (§6.11).

The value of the left operand should either be a PID, an atom that is registered on the current node as a process name (§10.2), or a 2-tuple of atoms where the first element is a name for some process on the node named by the second element. The value of the right operand is the message to be dispatched. Evaluation of send expressions is described precisely in §6.11.

There is no direct way for the sending process to find out whether the message ever arrived at the message queue of the recipient process or was retrieved (except that evaluation of a send expression will exit with reason `badarg` when the receiver is specified through an atom that is not a registered name).

The dispatched message contains no other information than the message as such. In particular, it does not identify the sending process unless its PID is made part of the message.

10.5.2 Message arrival

When a term *Msg* arrives at the message queue of the recipient process *P* it is placed at the end of `message_queue[`*P*`]`. If `status[`*P*`]` is `waiting`, then `status[`P`]` is changed to `runnable` and the process will eventually proceed in part 3 of the evaluation of a `receive` expression (§6.19.9).

For details, cf. §10.6.3.

10.5.3 Receiving a message

The only primitive for receiving a message in ERLANG is the `receive` expression (§6.19.9) which has the syntax

```
receive
    P₁ [when G₁] -> B₁ ;
    ...
    Pₖ [when Gₖ] -> Bₖ
[after
    E -> Bₖ₊₁]
end
```

where the `after` part is optional but must be included if $k = 0$.

When a receive expression is evaluated, each term in the message queue of the process is matched (in order) against each clause (i.e., a message is matched against the pattern of the clause and the guard of the clause is evaluated) until a message is found for which there is a matching clause. The message is then removed from the message queue of the process and the body of the first matching clause is evaluated.

If no term in the message queue matched any clause, then the process suspends until at least one term has arrived at the queue, each such term is then tried as above until one arrives that matches some clause. However, if the receive expression has an `after` part, then the waiting for a message expires after the specified number of milliseconds have passed and the expiry body is evaluated.

The evaluation of `receive` expressions is described in detail in §6.19.9.

10.5.4 Order of messages

It is assured (through the rules of signals, cf. §10.6.2) that if a process P_1 dispatches two messages M_1 and M_2 to the same process P_2, in that order, then message M_1 will never arrive after M_2 at the message queue of P_2.

Note that this does not guarantee anything about in which order messages arrive when a process sends messages to two different processes. Also note that linking/unlinking requests are processed as soon as they arrive at a process while messages can be received long after they arrived at the message queue (when the process evaluates a `receive` expression). For example, if a process P_1 evaluates

```
link(P₂), P₂ ! foo, unlink(P₂)
```

then the link between `P`$_1$ and `P`$_2$ may or may not still be in effect when process `P`$_2$ actually receives message `foo`, depending on whether the unlink request from process `P`$_1$ has arrived yet. In order to guarantee that the link is in effect when process `P`$_2$ receives `foo`, process `P`$_2$ could send an answer message to `P`$_1$ which would wait to receive that message before unlinking.

10.6 Signals

In this section we describe a model for how communication between two processes or between a process and a port takes place in Erlang. The model is assumed in other parts of this specification. It is important that it is a model: Erlang 4.7.3 does not use it in the implementation, but communication behaves according to the model.

All communication between processes and ports takes place through *signals.* A signal is characterized by a sending process/port, a destination process/port, a *kind* and additional data depending on the kind of the signal. The kind is one of the following:

Message.
Additional data: a term that is the actual message.

Exit signal.
Additional data: a Boolean flag saying whether the sending process just died or not and a reason which is a term.

Link request.
Additional data: none. See §10.3.

Unlink request.
Additional data: none. See §10.3.

Group leader request.
Additional data: the process that is to be the new group leader of the destination process. See §10.8.

Info request.
Additional data: a key which is an atom. See §13.8.12.

Erlang 4.7.3 has three additional kinds of signals, which are not described in detail here:

Suspend/resume request.
For suspending of resuming a process, respectively.

Garbage collection request.
For requesting that garbage collection be done.

Trace/notrace request.
For turning tracing of a process on or off, respectively

10.6.1 Sending signals

Sending a signal is completely asynchronous: the sending process has no direct way to infer when the signal reached its destination or even whether it was ever received.

Sending the various types of signals is described in §6.11 (*messages*), §10.4 (*exit signals*), §10.3 (*link and unlink requests*), §13.8.6 (*group leader requests*) and §13.8.12 (*info requests*).

10.6.2 Order of signals

The amount of time that passes between the dispatch of a signal s destined for a process `P` and the arrival of s at `P` is unspecified but positive. If `P` has completed, s will never arrive at `P`. In that case it is still possible that s triggers another signal, for example if it is a link request (cf. §10.6.3).

It is guaranteed that if a process P_1 dispatches two signals s_1 and s_2 to the same process P_2, in that order, then signal s_1 will never arrive after s_2 at P_2. It is ensured that whenever possible, a signal dispatched to a process should eventually arrive at it. There are situations when it is not reasonable to require that all signals arrive at their destination, in particular when a signal is sent to a process on a different node and communication between the nodes is temporarily lost.

10.6.3 Arrival of signals

Consider a signal in transit with destination process `P`. When the signal arrives at the node on which `P` resides or resided, what happens primarily depends on whether `P` has completed or not. (If the sender and destination processes reside on the same node, the processing may nevertheless be subject to delay.)

- If `P` has completed, what happens depends on the kind of the signal:
  - ∗ If the signal was a link request from a process P' then an exit signal `noproc` is sent to P'.
  - ∗ Otherwise, the signal is discarded.
- If `P` has not completed, what happens also depends on the kind of the signal:
  - ∗ *Message* with actual message `M`: the term `M` is placed at the end of `message_queue[P]` (§10.5.2).
  - ∗ *Exit signal* with sender P', flag `ProcessCompleted` and term `Cause`. (`ProcessCompleted` is `true` if P' completed and `false` otherwise.) First of all it is established whether reception of this exit signal causes `P` to complete abruptly or not.

  - If *ProcessCompleted* is `true`, P' is in `linked[`P`]`, `trap_exit[`P`]` is `false` and *Cause* is something other than `normal`, then P should complete: *Reason* is *Cause*.
  - Otherwise, if *ProcessCompleted* is `false` and *Cause* is `kill`, then P should complete: *Reason* is `killed`.
  - Otherwise, if *ProcessCompleted* is `false`, `trap_exit[`P`]` is `false` and *Cause* is something other than `normal` then P should complete: *Reason* is *Cause*.
  - Otherwise, P should not complete.

  What happens next depends on *ProcessCompleted*:

  - If *ProcessCompleted* is `true`: P must have been linked with P'. If P' is in `linked[`P`]`, then remove it.
  - If *ProcessCompleted* is `false`: P' must have called the BIF `exit/2` with P and *Cause* as arguments.

  What finally happens depends on whether P should complete abruptly or not.

  - If P should complete, the final processing is as described in §10.4.1.
  - If P should not complete, then unless *Cause* is `normal`, a message `{'EXIT',`P'`,`*Cause*`}` is placed at the end of `message_queue[`P`]` (§10.5.2).

- *Link request* with sender P'. If P' is not in `linked[`P`]`, then P' is added to `linked[`P`]`.
- *Unlink request* with sender P'. If P' is in `linked[`P`]`, then P' is removed from `linked[`P`]`.
- *Group leader request* with new group leader P'. `group_leader[`P`]` is set to P'.
- *Info request* with sender P' and key K. Reply to the sender with a message containing the requested information.

10.7 Scheduling of processes

If a process P is suspended in part 3 of the evaluation of a `receive` expression (§6.19.9), then `status[`P`]` is `waiting`, otherwise it is either `runnable` or `running`. Depending on `status[`P`]`, we say that P is waiting, runnable or running. At any time, at most one of the processes residing on a node is running, if the node is on a uniprocessor system. On a node on a multiprocessor system there may be multiple processes running simultaneously. A newly spawned process is initially runnable.

In §10.9.2 it is described how `status[P]` may change to `waiting` or to `runnable`. In addition, the scheduler of an ERLANG node has as its task to repeatedly choose which runnable process gets to run. When the scheduler changes the `status[P]` to `running`, process `P` will run for one *cycle*, unless during that cycle `status[P]` changes to `waiting`, in which case another runnable process will be scheduled to run. At the end of the cycle, if `B` is still running, the scheduler changes `status[P]` back to `runnable`.

ERLANG 4.7.3 attempts to make cycles be of equal and short duration, the latter to favour interactive processes that do a small amount of work between waiting states.

When we write that ERLANG 4.7.3 schedules a set of processes *fairly*, we mean that each runnable process in the set should eventually be scheduled and preferably in the same order that they became runnable. (This is a very weak requirement.)

Each process `Q` has an associated priority `priority[Q]`, which is an ERLANG term. Three priorities are recognized: `high`, `normal` and `low`.

- While there are runnable processes with priority `high`, ERLANG 4.7.3 schedules the processes with priority `high` fairly, ignoring processes with priority `low` or `normal`.
- While there are no runnable processes with priority `high` or `low`, ERLANG 4.7.3 schedules the processes with priority `normal` fairly.
- While there are no runnable processes with priority `high` or `normal`, ERLANG 4.7.3 schedules the processes with priority `low` fairly.
- While there are no runnable processes with priority `high` but there are runnable processes with `normal` or `low`, ERLANG 4.7.3 schedules the processes with priority `normal` and `low` fairly. It also attempts to schedule processes with priority `normal` *normal_advantage* times between each scheduling of a process with priority `low`.

For ERLANG 4.7.3, *normal_advantage* is 8.

10.8 Process group leaders

Each process `P` has a *group leader*, which is a process, possibly itself, referred to in this specification as `group_leader[P]`. The set of processes that have the same group leader may be thought of as a *process group*, hence the term group leader.

The group leader of a process is the default process for handling in- and output of the process.

A process can retrieve its own group leader (using the BIF `group_leader/0`, §13.8.5) and any process can set the group leader of any process (using the

BIF `group_leader/2`, §13.8.6). When a new process is created, its group leader will be the same as that of the process that spawned it.

In- and output (except for direct communication with a port, cf. §12) is otherwise not covered by this specification.

10.9 Static and dynamic properties of a process

When a process is created, some properties of the process are determined and will be in effect until the process terminates. During that time also a dynamic state is maintained, consisting of properties of the process that change as time passes. This state is affected by and affects the computation of the process.

We refer to these as the *static* and *dynamic properties* of a process. We refer collectively to the values of the latter at a certain time as the *state* of the process at that time.

10.9.1 Static properties

`creation[P]`
: The value of `creation[`*`P`*`]` is the value of `creation[`*`N`*`]` for the node *`N`* on which *`P`* was created.

`ID[P]`
: The value of `ID[`*`P`*`]` is a nonnegative integer that is a serial number for *`P`* on the node on which it was created. The value cannot be obtained directly but is used in the transformation to the external term format (§D).

`initial_call[P]`
: When a process *`P`* is spawned it is evaluating a remote application *`Mod`*`:`*`Fun`*`(`*`Arg`*$_1$`,...,`*`Arg`*$_k$`)` (the initial call). The value of `initial_call[`*`P`*`]` is then the 3-tuple `{`*`Mod`*`,`*`Fun`*`,`*`k`*`]}`.

`node[P]`
: When a process is spawned it is created on some node `node[`*`P`*`]`. This node never changes. The process *`P`* itself can access `node[`*`P`*`]` by evaluating an expression `node()` (§13.10.6). Any process can access `node[`*`Q`*`]` for a process *`Q`* by evaluating an expression `node(`*`Q`*`)` (§13.10.7).

10.9.2 Dynamic properties

`current_function[P]`
: The value is either the atom `undefined`, or a 3-tuple `{`*`Mod`*`,`*`Fun`*`,`*`k`*`}` if the most recently begin function call was to the function *`Mod`*`:`*`Fun`*`/`*`k`* (§6.7.3). It should be updated each time a named function is entered

(§6.7.1). It can be accessed as `process_info(P,current_function)` (§13.8.12).

`dictionary[P]`
: The value is a table (§2.4). A process can access and modify `dictionary[P]` using the six BIFs `get/0`, `get/1`, `get_keys/1`, `put/2`, `erase/0` and `erase/1` (§13.9). The table is initially empty.

`error_handler[P]`
: The value is a module name (an atom) and it affects the evaluation of applications of undefined function names (§6.18). A process `P` can set `error_handler[P]` to a module name `M` by evaluating an expression `process_flag(error_handler,M)` (§13.8.10). The value is initially `error_handler`.

`group_leader[P]`
: The value is a PID which identifies the group leader of process `P` (§10.8). A process `P` can access `group_leader[P]` by calling the BIF `group_leader/0` (§13.8.5) and any process can set the group leader of any process by calling the BIF `group_leader/2` §13.8.6. (Any process can also access `group_leader[P]` as `process_info(P,group_leader)`.) When a process `P` spawns a new process `Q`, the initial value of `group_leader[Q]` is `group_leader[P]`.

`heap_size[P]`
: The value should reflect the amount of memory presently used for storing the terms allocated by process P. `heap_size[P]` can be accessed through the BIF `process_info/2` (§13.8.12).

`linked[P]`
: The value is a representation of the set of PIDs and ports that identify the processes and ports to which `P` is linked (§10.3). It cannot be modified directly but a PID or port `Q` will be added to `linked[P]` if it is not already in it and either

  - `P` evaluates an application `link(Q)` (§13.8.7), or
  - `P` receives a link request signal from `Q`.

  A PID `Q` will be removed from `linked[P]` if it is in the set and

  - `P` evaluates an application `unlink(Q)` (§13.8.19),
  - `P` receives an unlink request signal from `Q`, or
  - `P` receives an exit signal `{'EXIT',Q,Reason}` for some term `Reason`, and the exit signal was sent due to process completion (§10.4).

`memory_in_use[P]`
: The value should reflect the total amount of memory presently used by process P. `memory_in_use[P]` can be accessed through the BIF `process_info/2` with `memory` as second argument (§13.8.12).

`message_queue[P]`
: The value is a representation of a queue of terms, which are the messages that have arrived at the process (§10.5.2) but that have not yet been received (§10.5.3). When a `receive` expression is evaluated by a process *P*, it will first try to match the messages in `message_queue[P]` against the clauses, in the order that they appear in the queue (§6.19.9). New messages will be added at the end of the queue. There is no direct way of accessing or modifying `message_queue[P]` for a process *P*.

`priority[P]`
: The value is one of the atoms `low`, `normal` and `high` and it affects the scheduling priority of process *P* (§10.7). A process *P* can set `priority[P]` to priority atom *R* by evaluating an expression `process_flag(priority,R)` (§13.8.10). The value is initially `normal`.

`reductions[P]`
: The value is an integer that should reflect the number of function calls that the process has begun (§6.7.3) since it was spawned. ERLANG 4.7.3 defines a scheduling cycle (§10.7) as a certain (implementation-defined) number of reductions. `reductions[P]` can be accessed through the BIF `process_info/2` (§13.8.12).

`registered_name[P]`
: The value is an atom *A* if *P* is registered with the name TZA or `[]` if *P* is not registered under any name. `registered_name[P]` must be coherent with the contents of `registry[node[P]]`. The value is initially *[]*. It can be set to an atom by the BIF `register/2` (§13.11.1) and to `[]` by the BIF `unregister/1` (§13.11.3). Given the atom `registered_name[P]`, the PID *P* can be obtained through the BIF `whereis/1` (§13.11.4, cf. §11.5) whereas given *P*, `registered_name[P]` can be obtained through the BIF `process_info/2` (§13.8.12).

`stack_trace[P]`
: `stack_trace[P]` is a dynamic representation of the evaluation that *P* is carrying out. It should contain sufficient information that a BIF call `erlang:check_process_code(P,Mod)` should be able to return `true` if *P* is using `old_version[module_table[node[P]](Mod)]` and `false` otherwise. It cannot be accessed or updated explicitly.

`status[P]`

`status[P]` reflects the scheduling status of *P*: waiting, runnable or running. `status[P]` is initially `runnable`. `status[P]` is changed from `runnable` to `running` or from `running` to `runnable` by the scheduler. If `status[P]` is `waiting` when a message is added to the end of `message_queue[P]`, it is changed to `runnable`. If `status[P]` is `waiting` when `timer[P]` changes from `1` to `0`, it is changed to `runnable`. Evaluation of a `receive` expression (§6.19.9) may set `status[P]` to `waiting`. `status[P]` can be accessed through the BIF `process_info/2` but cannot be changed except as described above.

`timer[P]`

If the value of `timer[P]` is positive, it reflects the number of milliseconds that remain until the status of process *P* should change from `waiting` to `runnable`.

If `timer[P]` is positive, it is decremented automatically once every millisecond. When it changes from `1` to `0`, if `status[P]` is `waiting`, `status[P]` is changed to `runnable`.

`timer[P]` cannot be accessed directly and is set only as part of the evaluation of a `receive` expression (§6.19.9).

`trap_exit[P]`

The value is a Boolean atom and it affects how exit signals arriving at process *P* are processed (§10.4). A process *P* can set `trap_exit[P]` to a Boolean atom *B* by evaluating an expression `process_flag(trap_exit,B)` (§13.8.10). The value is initially `false`. `trap_exit[P]` can be accessed as `process_info(P,trap_exit)` (§13.8.12).

Chapter 11

Nodes

11.1 Single-node and multi-node systems

An ERLANG node is the operating system for ERLANG processes. Every process runs on some specific node and once a process has been created it remains on that node.

By default an ERLANG node is *isolated* and constitutes a single-node system. In order to become a *communicating* node that can be part of a cluster of ERLANG nodes, it must have a process registered under the name `net_kernel` that has opened a port through which it communicates with other nodes using the protocol described in §D.

Every ERLANG node has a name, which is an atom. The printname of the atom always contains exactly one `@` character (`\100`). An isolated node always has the name `nonode@nohost`. In a communicating node, the part of the printname after the `@` must be the network name of the host computer on which the node resides (typically an Internet domain name). We call an atom with such a printname a *valid node name*. The name of each communicating node must be unique. As the node name contains the network name of the host computer (which is assumed to be unique), it is sufficient to ensure that all nodes running on the same host computer have unique names. This is ensured by the use of EPMD, as described in §11.2. Because node names are unique and that nodes are always referred to through their names in the language, we will identify nodes with their names.

A node with a `net_kernel` process that has opened a port *R* as described above becomes a communicating node with name *A* by some process making a BIF call `erlang:set_node(`*A*`,`*R*`)` (§13.10.11).CHECK THE ARGUMENTS!!! It is also possible to make a node communicating from the beginning [6, pp. 5–9].

On an isolated node the effects and result is undefined when any other node name than `nonode@nohost` is used in a BIF that have an explicit node argument or when sending messages. (This is not stated again in the descriptions of BIFs and send expressions.) In the remainder of this chapter, ERLANG nodes are assumed to be communicating unless it is stated explicitly that a node is or may be isolated. Moreover, as every communicating node has a unique name and there is no way to refer to a node except through its name, we will identify nodes with their names. Thus, when we write "node *A*", where *A* is an atom, we mean the node with name *A*, provided that such a node exists.

Each node has a term associated with it, called the *magic cookie* of the node. In order for a processes on some node N_1 to communicate successfully with a process on another node N_2, the *magic cookie* of node N_2 must be provided in the communication from node N_1. A group of nodes can be protected from communication originating on other nodes by sharing a magic cookie and not disclosing it to other nodes. Magic cookies are discussed in more detail in §11.4.

At any given time, each ERLANG node has a number of other nodes as *friends*. The set of such friends may change dynamically over time. When a process running on some node N_1 attempts to communicate with a process on another node N_2, node N_1 attempts to set up a friendship with node N_2 through negotiation between the `net_kernel` processes of the two nodes. Friendship is a symmetric property so in the process, N_1 also becomes a friend of node N_2. The friendship is terminated when the nodes can no longer communicate or when a process residing on one of the nodes calls the BIF `erlang:disconnect_node/1` (§13.10.1) with the name of the other node as argument. Friendship is described in more detail in §11.3.

11.2 Registering a node

MAKE COHERENT WITH ABOVE AND FINISH!!!

Any computer that can be the host of ERLANG nodes must provide the service *EPMD* (ERLANG Port Mapper Daemon). The EPMD service on the host computer stores the names of all ERLANG nodes running on the computer, ensuring that the names are unique. It will provide a handle to the port of the `net_kernel` process of any node residing on the host computer, given the name of the node (cf. §11.3).

MUST OBTAIN INFO ABOUT SET_NODE/2-3 BEFORE FIXING NEXT PARAGRAPH!!!

An ERLANG node becomes a communicating node by spawning a process registered with the name `net_kernel`. It should open a port through which other nodes can communicate with it. It then contacts the EPMD service on its host computer, requesting to register the node with a certain name and the opened port. The protocol for this registration is described in §**??**.

11.3 Initializing and terminating friendship

When a signal is to be delivered from a process residing on some node N_1 to a process residing on a different node N_2, N_2 must be a friend of N_1. NOT CORRECT!!!: Concretely, this means that the `net_kernel` process of node N_1 has opened a port to the `net_kernel` process of node N_2. Associated with this port is an *atom table*, as described in §D.1.

Making node N_2 a friend of node N_1 is thus initiated when a process on node N_1 needs to communicate with a process on node N_2 and node N_2 is not yet a friend of node N_1. The friendship remains until node N_1 detects that it cannot communicate with node N_2 or either node calls the BIF `erlang:disconnect_node/1` (§13.10.1) with the name of the other node as argument.

When node N_1 makes node N_2 a friend, node N_1 should also become a friend of node N_2. Under normal circumstances, friendship is thus a reflexive relation. When the means of communication is broken between two nodes that are friends, they may not detect this simultaneously and friendship may then temporarily be unilateral. Should communications be restored in a situation where node N_1 is still a friend of node N_2 but node N_2 is no longer a friend of node N_1, then N_1 should cease to be a friend of node N_2 before (bilateral) friendship is resumed. (This is to ensure that both processes on N_1 monitoring friendship with N_2 and processes on N_2 monitoring friendship with N_1 will be notified that friendship has been broken, cf. §11.6.)

NEXT PARAGRAPH IS HIGHLY UNCERTAIN!!!

When a node N_1 wishes to initiate friendship with a node N_2 the `net_kernel` process of node N_1 first contacts the EPMD on the host computer of node N_2 and requests a handle of the port of the `net_kernel` process of node N_2, as described in §**??**. If this succeeds, then the `net_kernel` process of node N_1 contacts the `net_kernel` process of node N_2 and completes the setup of the friendship as described in §D. That communication also establishes node N_2 as a friend of node N_1.

The current set of friends of a node N is referred to in this document as `friends[`N`]`.

11.4 Remote communication and magic cookies

All forms of communication between two processes residing on different nodes go through the `net_kernel` processes on those nodes. Note that all communication between processes is in the form of signals (§10.6).

Each node N has associated with it an atom `magic_cookie[`N`]` that is called the *magic cookie* of the node. Any process residing on node N can obtain the magic cookie of node N through a BIF call `erlang:get_cookie()` (§13.10.2).The magic cookie of node N can be set by any process residing on node `N` to some atom `A` through a BIF call `erlang:set_cookie(N,A)` (§13.10.10).

Each node also has a table `magic_cookies[N]` in which the keys are node names and the values are the presumed magic cookies of those nodes. The presumed magic cookie on node N_1 of a node N_2 cannot be retrieved but can be set to some atom `A` by any process residing on node N_1 through a BIF call `erlang:set_cookie(`N_2`,A)` (§13.10.10).

When a process P_1 residing on a node N_1 attempts to send a signal `S` to a process P_2 on a different node N_2, then the following happens:

- The signal `S`, source P_1 and destination P_2 are passed to the `net_kernel` process on node N_1.
- If N_2 is not in `friends[N]`, then the `net_kernel` attempts to make it a friend of N_1. Should this fail, the signal s is simply discarded.
- The `net_kernel` process on N_1 looks up the atom N_2 in `magic_cookies[`N_1`]` and if there is no value for N_2, then the atom `nocookie` is inserted for it. Let `C` be the result of the lookup (possibly after insertion of `nocookie`).
- The `net_kernel` process on N_1 passes P_1, `C`, `S` and P_2 to the `net_kernel` process on N_2 using the protocol described in §D.
- The `net_kernel` process on N_2 compares `C` with `magic_cookie[`N_2`]`.
    - ∗ If `C` and `magic_cookie[`N_2`]` are (exactly) equal (§4.11.1), then the signal `S` is passed on to process P_2 as described in §10.6.3.
    - ∗ Otherwise, if the signal `S` is a message with `T` as body, a message `{`P_1`,badcookie,`P_2`,T}` is placed at the end of the message queue of the `net_kernel` process of node N_2.
    - ∗ Otherwise, the signal is passed on to process P_2 as described in §10.6.3.

The magic cookie mechanism is built into the communication mechanism of Erlang in order to ensure that processes on a node cannot disrupt processes running on an arbitrary node. The security model of Erlang is such that if a process P_1 resides on the same node as on a process P_2, or P_1 resides on a node that has the correct magic cookie for the node on which P_2 resides, then process P_1 has complete access to process P_2. Process P_1 can send arbitrary signals (including exit signals with `kill` as reason, cf. §10.6.3) to process P_2. It is thus up to the programmer to ensure that processes running on the same node or on nodes that have each others' correct magic cookies can trust each other. (Typically but not necessarily one will make sure that whenever a node N_1 has the correct magic cookie of node N_2, node N_2 has the correct magic cookie of node N_1. In particular, a cluster of nodes may share a magic cookie.)

11.5 Process registry

For each node there is a table `registry[N]` in which the keys are atoms naming processes residing on that node and the values are the PIDs of the processes. The process registry makes it possible to send messages to registered processes on a node without knowing their PIDs.

Some processes, such as the `net_kernel` of a communicating node, are spawned and registered automatically by the ERLANG system. It is also possible for any process on a node *N* to register a name *A* for a live process *P* residing on *N* through a BIF call `register(A,P)` (§13.11.1). There can be at most one name registered for a process. A process name is removed from the registry automatically when the process completes. It is also possible for any process on a node *N* to remove the name *A* from the registry of *N* through a BIF call `unregister(A)` (§13.11.3). A process *P* can look up a name *A* in the registry on `node[P]` and obtain the PID through a BIF call `whereis(A)` (§13.11.4) or obtain a list of all currently registered names on `node[P]` through a BIF call `registered()` (§13.11.2).

The BIF `register/2` must ensure that a process cannot be registered on node *N* unless it resides on that node and is alive. Also it must be ensured that when a process completes, any name for it in the registry of the node on which it resides is removed.

11.6 Monitoring of nodes

THERE ARE THINGS MISSING ABOUT THIS, ESPECIALLY WHEN DISCUSSING SIGNALS AND THE DYNAMIC STATE.

Any process on a node N_1 can be notified when some node N_2 ceases to be a friend of N_1.

After a process *P* on a node N_1 has made a BIF call `monitor_node(`N_2`,true)` (§13.10.5),where N_2 is an atom, a message `{nodedown,`N_2`}` will be sent to *P* if N_2 ceases to be a friend of N_1. If N_2 is not a friend of N_1 when the call is made, the message `{nodedown,`N_2`}` is sent to *P* immediately. A BIF call `monitor_node(`N_2`,false)` cancels the effect of a call `monitor_node(`N_2`,true)`. The effect of a BIF call `monitor_node(`N_2`,false)` when there is no call `monitor_node(`N_2`,true)` to cancel is undefined. There can be more than one call `monitor_node(`N_2`,true)` in effect and *P* will receive one message for each call `monitor_node(`N_2`,true)` that has not been canceled when N_2 ceases to be a friend of N_1.

11.7 The state of a node

When a node is started, some properties of the node are determined and will be in effect until the node is terminated. During that time also a dynamic state is maintained, consisting of properties of the node that change as time passes. This state is affected by and affects the behaviour of the node.

We refer to these as the *static* and *dynamic properties* of a node. We refer collectively to the values of the latter at a certain time as the *state* of the node at that time.

11.7.1 Static properties

`creation[N]`

When a node is started, the value of `creation[`*`N`*`]` — a nonnegative integer — is obtained from EPMD. Its purpose is to to distinguish the current instance of the node from previous instances that had the same name. The value is used when creating new refs, PIDs and ports.

`communicating[N]`

This is a Boolean atom which is `false` if the node is isolated and `true` if it is communicating. When a node becomes communicating, the property is changed from `false` to `true` but after that it cannot be changed. (The property is thus not strictly static as it can change once but in practice we can treat it as being static.) It can be accessed on the node itself through the BIF `erlang:is_alive/0` (§13.10.4).

`name[N]`

This is the name of the node, which is an atom. For an isolated node it is always `nonode@nohost`. When a node becomes communicating, the name is changed but after that it cannot be changed. (The property is thus not strictly static as it can change once but in practice we can treat it as being static.) It can be accessed on the node itself through the BIF `node/0` (§13.10.6).

`preloaded[N]`

This is a set of the names of the modules that were loaded as part of starting the node. Although there should normally never be any reason to delete these modules, their presence in the (static) set does not imply that they are still loaded. The set can be accessed on the node itself through the BIF `erlang:preloaded/0` (§13.7.4).

11.7.2 Dynamic properties

`atom_tables[N]`

This is a table that for each friend of *`N`* contains a row with the name and the atom table (§D.1) for that friend as key and value, respectively. When *`N`* gets a new friend, a row should be added to `atom_tables[N]` with the name of the friend as key and an empty atom table as value. When a node ceases to be a friend of *N*, its row in `atom_tables[N]` must be removed.

`distribution_port[N]`
: On a communicating node, this is the port that was given as second argument to `node:alive/2` ???. The EPMD server for the computer on which the node resides will communicate with the `net_kernel` process of *N* through that port.

`entry_points[N]`
: This is a table mapping triples consisting of a module name (an atom), a function symbol (an atom) and an arity (a nonnegative integer) to entry points to executable code. The table cannot be accessed directly but is used when evaluating a function application where the function is specified through a module name, a function symbol and an arity (§6.7.1). The table is updated by BIFs for module dynamics (§9, §13.7).

`friends[N]`
: This is a set of atoms that are names of nodes that are friends of node *N* (§11.3). It can be accessed on node *N* by calling the BIF `nodes/0` (§13.10.8). The set is added to when a friendship has been established with another node, typically because a process on one of the nodes has attempted to communicate with a process on the other node. The set is subtracted from when node *N* loses contact with a friend node (e.g., because communication between host computers has been lost, the other node has been terminated, or the host computer on which it resides has been restarted), or when a process on either node calls the BIF `erlang:disconnect_node/1` (§13.10.1) with the name of the other node as argument.

`garbage_collection[N]`
: This is the state of the garbage collection gauge, which is updated automatically when processes collect garbage to reflect the number of garbage collection operations that have been carried out by processes on the node and the number of memory words reclaimed by such operations. It can be accessed through the operation `current_gc[N]`, which returns a 3-tuple `{NumberOfGCs,WordsReclaimed,0}` of integers where `NumberOfGCs` is the total number of garbage collection operations that have been carried out by processes on the node and `WordsReclaimed` is the total number of memory words reclaimed by such operations. (The third integer is always zero and is there to confuse the enemy.)

`magic_cookie[N]`
: This term must be provided by any process on another node that

wishes to communicate with a process on node *N* (cf. §11.4). The value of `magic_cookie[N]` can be obtained by a process on node *N* through the BIF `erlang:get_cookie/0` (§13.10.2).`magic_cookie[N]` can be set by a process on node *N* by calling the BIF `erlang:set_cookie/2` (§13.10.10) with `name[N]` and the new magic cookie as arguments.

`magic_cookies[N]`
: This is a mapping from atoms to terms. Its role in process communication across nodes is described in §11.4. `magic_cookies[N]` cannot be accessed directly but can be modified by a process on node *N* by calling the BIF `erlang:set_cookie/2` (§13.10.10) with the name of another node and the magic cookie to be used as arguments.

`module_table[N]`
: This is a table where the keys are module names, i.e., atoms. The table contains a row with key *Mod* if a module named *Mod* has ever been loaded on the node. The value *R* of each row contains two fields: `current_version[R]` and `old_version[R]`.

- `current_version[R]` is either a binary representing the compiled code for the current version of the module, or `none`.
- `old_version[R]` is either a binary representing the compiled code for the current version of the module, or `none`.

The table is initially empty. For a new row both fields are `none`. It cannot be accessed or modified directly but is accessed or updated by most BIFs for module dynamics (§9, §13.7).

`monitored_nodes[N]`
: This is a table where each row has a node name as key and a table as value. Each such table has the PID of a process residing on *N* as key and a nonnegative integer as value. If `monitored_nodes[N]` contains a row with key N' and value t and t contains a row with key *P* and value *I*, then each time node N' ceases to be a friend of node *N*, $\Re^{-1}[I]$ messages `{node_down,N'}` will be sent to process *P*. The value of `monitored_nodes[N]` cannot be accessed directly. It can be modified using the BIF `monitor_node/2` (§13.10.5). It is initially empty.

`ports[N]`
: This is a set of the ports that are open on node *N*. It can be accessed by processes on *N* through the BIF `ports/0` (§13.12.5).The set is updated implicitly when a new port is opened on *N* (§12.4) and when a port on *N* is closed (§12.8).

`processes[N]`

This is a set of the PIDs of the processes that run on node *N*. It consists of the PIDs of all processes that have been spawned on *N* and that have not yet completed. It can be accessed by processes on *N* through the BIF `processes/0` (§13.10.9).The set is updated implicitly when a new process is spawned on *N* (§10.1) and when a process on *N* completes (§10.4.1).

`reductions[N]`

This is the state of the reduction gauge. It is updated automatically to reflect the number of function calls that have been made on node *N*.

`reductions[N]` cannot be accessed directly but the atomic operation `current_reductions[N]` uses and updates the value of `reductions[N]` to return a 2-tuple `{Total,SinceLast}` of integers where both measure the number of function calls on node *N*. The first is the number of reductions since node *N* was started while the second is since the previous invocation of `current_reductions[N]`.

The first time `current_reductions[N]` is invoked, *Total* and *SinceLast* will be the same. This operation should only be invoked as part of an explicit application `statistics(reductions)` (§13.10.13) in a user program.

`ref_state[N]`

This is the state of the ref generator. It cannot be accessed directly but the atomic operation `next_ref[N]` uses the value of `ref_state[N]` to provide the ID for a new ref and at the same time increments the value of `ref_state[N]` so that the next invocation of `next_ref[N]` will produce a unique ref.

`registry[N]`

This is a mapping from atoms to PIDs. It is ensured that at any time,

- the mapping is invertible (i.e., that at most one name is registered for each PID),
- that all PIDs refer to processes residing on node *N*, and
- that all PIDs refer to live processes.

This implies that when a process completes, any name registered for it must be removed from the registry.

`registry[N]` can be accessed by processes running on node *N* through the BIFs `register/2`, `unregister/1`, `registered/0` and `whereis/1` (§13.11). It can be accessed indirectly by processes running on any node when sending messages (§6.11).

`runtime[N]`

This is the state of the run time gauge. It is updated automatically to reflect the amount of time spent running processes on node *N*. It cannot be accessed directly but the atomic operation `current_runtime[`*N*`]` uses and updates the value of `runtime[`*N*`]` to return a 2-tuple `{`*Total*`,`*SinceLast*`}` of integers where both measure the total time in milliseconds spent on running processes on node *N*. The first element is the time since node *N* was started while the second is since the previous invocation of `current_runtime[`*N*`]`.

The first time `current_runtime[`*N*`]` is invoked, *Total* and *SinceLast* will be the same. This operation should only be invoked as part of an explicit BIF application `statistics(runtime)` (§13.10.13) in a user program.

`wall_clock[N]`

This is the state of the real time gauge. It is updated automatically to reflect the passing of real time in the world where node *N* resides. It cannot be accessed directly but the value of `wall_clock[`*N*`]` is used and updated by the atomic operation `current_wall_clock[`*N*`]` to return a 2-tuple `{`*Total*`,`*SinceLast*`}` of integers where both measure the time in milliseconds that has passed in the world. The first element is the time since node *N* was started while the second is since the previous invocation of `current_wall_clock[`*N*`]`.

The first time `current_wall_clock[`*N*`]` is invoked, *Total* and *SinceLast* will be the same. This operation should only be invoked as part of an explicit BIF application `statistics(wall_clock)` (§13.10.13) in a user program.

Chapter 12

Ports

12.1 Overview of ports

An Erlang node and thus an Erlang process on it communicates with resources in the outside world (including the rest of the computer on which it resides) through *ports*. Examples of such external resources are files, drivers (§12.2) and non-Erlang processes running on the same host. Information sent to a port from outside the Erlang node can be read by an Erlang process as messages and messages sent by an Erlang process to a port are made available to the external program (or written to a file).

A port can be unidirectional or bidirectional, as requested when the port is opened. If the port is for communication with an external process, it would typically be bidirectional. If the port is connected to a file that is being read, it would typically be input-only. If the port is connected to a file that is to be written, it would typically be output-only.

A port is always *owned* by some process that resides on the same node as the port. That process will receive input in the form of messages from the port when information is sent to the port from outside Erlang. The port is initially owned by the process that opened the port but ownership of a port can be transferred to another process on the same node.

An external resource has much in common with a process and thus a port has much in common with a PID:

- Communicating with an external resource is similar to communicating with a process: messages to a port are sent using the ! operator and messages from a port are received using `receive` expressions.
- A process can link to a port in order to be notified when an external resource completes.

However, ports are restricted as compared with processes. For example, information written to a port from outside Erlang is always sent as a message to the process that owns the port. An external resource thus cannot communicate directly with an arbitrary ERLANG process.[1] Also, it is not possible to register a name for a port.

Each port is identified by an ERLANG term that is itself referred to as a port (although calling it a *port identifier* would be more in harmony with process vs. process identifier).

When a port is created, it is connected with an external entity, which is either

- a recently spawned external process or recently opened driver (§12.2);
- an external resource, such as a file.

As in the case of PIDs, there can be ports that were created for communication with an external process or resource that no longer exists. What happens when a BIF is given such a port varies, cf. §13.12.

The interface to a port from the outside of Erlang is dependent on the operating system where the node resides. A driver or external process that has been started by the ERLANG system when a port was opened can read from a port and write to it using a pair of procedures that we refer to below as *read* and *write*.

ERLANG BIFs relating to ports are described in §13.12.

12.2 Drivers

An ERLANG system may provide a way to extend a node with software written in other programming languages than ERLANG. Such software is called a *driver* and would normally communicate with ERLANG processes through a port.

The interface to external software is thus the same regardless of whether it runs as part of the node or as external processes.

12.3 I/O terms

Define an I/O term to be either

- a binary,
- nil, or
- a cons where the head is a byte or an I/O term and the tail is an I/O term.

The *contents* of an I/O term is a sequence of bytes defined recursively as follows:

[1]A typical way to establish communication between an external resource and arbitrary processes is to let the process that is connected to the port be a server acting as a proxy.

- The contents of a binary is the sequence of bytes of the binary, in the order they appear in the binary.
- The contents of an empty list is an empty sequence.
- The contents of a cons of a byte and an I/O term is the byte followed by the contents of the tail.
- The contents of a cons of two I/O terms is the contents of the head followed by the contents of the tail.

To clarify, the function `contents/1` defined below returns the contents of an I/O term, represented as a list of bytes.[2]

```
contents(B) when binary(B) -> binary_to_list(B) ;
contents([]) -> [] ;
contents([I|Y]) when integer(I) -> [I | contents(Y)] ;
contents([X|Y]) -> contents(X) ++ contents(Y).
```

12.4 Opening ports

A port is opened by calling the BIF `open_port/2` (§13.12.1). In an application `open_port(`*`Resource`*`,`*`Options`*`)`, the *`Resource`* argument identifies the resource to which a port is being opened and the *`Options`* argument determines the details of the behaviour of the port.

`Resource` is one of

- a 2-tuple `{spawn,`*`Command`*`}`, where *`Command`* is a string or an atom (in which case its printname will be used). A driver is opened inside the ERLANG node or an external process is spawned and a new port is connected with the driver or the standard input and output of the external process. *`Command`* must not contain a null character (`\000`). Suppose that the result of extracting whitespace-separated words from *`Command`* is $\mathtt{Wd}_1$, $\mathtt{Wd}_2$, ... , $\mathtt{Wd}_k$.
  - ∗ If $\mathtt{Wd}_1$ is the name of an installed driver, then such a driver will be started with the string $\mathtt{Wd}_2$, ... , $\mathtt{Wd}_k$ as input.
  - ∗ Otherwise $\mathtt{Wd}_1$ will be interpreted as the name of an external program which is started with $\mathtt{Wd}_2$, ... , $\mathtt{Wd}_k$ as its arguments.
- a string or atom (in which case its printname will be used) *`Resource`*. If *`Resource`* can be interpreted by ERLANG 4.7.3 as identifying a resource (for example, naming a file), a new port is connected with the resource. If the resource is a file, then either the option `in` (for reading from the file) or the option `out` (for writing to the file) must be given.

[2]It is designed for clarity, not efficiency.

- a 3-tuple `{fd, In, Out}`, where *`In`* and *`Out`* are integers. On a node running on a POSIX-compliant operating system, *`In`* and *`Out`* are interpreted as file descriptors and a new port is connected writing to *`In`* and reading from *`Out`*. On other nodes, the behaviour is implementation-defined.

If *`Resource`* is not one of the permitted alternatives, `open_port/2` should exit with `badarg`.

`Options` is a list of items, each of which is a term from one of the following groups. At most one term from each group should be present. If no term from a group is in *`Options`*, it is as if the default term for the group was present.

Stream/packets

- `stream` (the default). Messages are sent without packet headers.
- `{packet,`*`N`*`}`, where *`N`* is one of the integers `1`, `2` and `4`. Messages are sent in packets with the packet headers in *`N`* bytes (§12.6).

Process inputs and outputs

(Only relevant on POSIX-compliant systems and for `{spawn,`*`Command`*`}`.)

- `use_stdio`. (The default.) Use the standard input and standard output (i.e., file descriptors 0 and 1 on UNIX systems) of the spawned process for communicating with Erlang.
- `nouse_stdio`. Use alternative input and output channels of the spawned process (file descriptors 3 and 4 on UNIX systems) for communicating with Erlang.

Direction

- `in`. The port can only be used for input from the external resource.
- `out`. The port can only be used for output to the external resource.
- *None*. The port can be used for input from and output to the external resource.

Received data

- `binary`. Input from the external resource is received as binaries.
- *None*. Input from the external resource is received as lists of bytes.

End of stream

- `eof`. The port is not closed when the external resource is depleted, instead a message `{Port,eof}` is sent to the owning process.
- `noeof`. (The default.) The port is closed when the external resource is depleted. All processes linked to the port will then receive exit signals.

If *`Options`* is not a list or if *`Options`* contains unrecognized terms, `open_port/2` should exit with `badarg`.

When a port is opened, it is automatically linked to the owning process (§12.9).

12.5 Controlling a port from an Erlang process

By controlling a port *`R`* to an external resource we mean to write data to the port or changing properties of the port itself. Such control is exercised by sending a message with *`R`* as destination (§6.11). As before, evaluation of such a send expression always succeds. However, the ERLANG system subsequently attempts to interpret the message as port control information and if that interpretation fails, an exit signal (§10.4) will be sent to the process owning *`R`*, i.e., `owner[`*`R`*`]` (§12.10.2).

Any process may control a port. However, the process that sends a message to *`R`* must have the PID of the process that owns *`R`* (because `owner[`*`R`*`]` must be part of the message, as described below).

Consider a message *`v`* sent to a port *`R`*.

- If the port has been closed, then nothing happens.
- If *`v`* is a tuple *`{owner[R], {command, Data}}`*, then:
  - ∗ If `in[`*`R`*`]` is `false`, then an exit signal `badsig` is sent to `owner[`*`R`*`]`.
  - ∗ If *`Data`* is an I/O term (§12.3), then the contents of *`Data`* will be transmitted through *`R`*, as described in §12.6 (where it is also described how an exit signal `badsig` is sent to `owner[`*`R`*`]` if the number of bytes in the contents of *`Data`* is too large to fit in the packet header)
  - ∗ Otherwise, an exit signal `badsig` is sent to `owner[`*`R`*`]`.
- If *`v`* is a tuple *`{owner[R], close}`*, then the port *`R`* is closed and a message `{`*`R`*`, closed}` is sent to `owner[`*`R`*`]`.
- If *`v`* is a tuple *`{owner[R], {connect,`* P_2*`}}`*, then what happens depends on P_2:

  - ∗ If P_2 is the PID of a process residing on the same node as R (and thus `owner[`R`]`, then a message `{`R`, connected}` is sent to `owner[`R`]` and the value of `owner[`R`]` is changed to P_2.
  - ∗ Otherwise, an exit signal `badsig` is sent to `owner[`R`]`.
- Otherwise, an exit signal `badsig` is sent `owner[`R`]`.

It should be noted that all processing when writing to a port happens asynchronously, also error processing.

12.6 Transmitting data from an Erlang process to the outside

Suppose that an ERLANG process has sent an I/O term T to a port R and that the contents of the I/O term is a sequence of bytes $b_1, \ldots, b_k$.

Depending on the value of `packeting[`R`]`, the byte sequence transmitted to the external resource may be prepended with a packet header by the ERLANG system:

- If `packeting[`R`]` is `stream`, then the byte sequence will be transmitted as is.
- If `packeting[`R`]` is `{packet,`N`}`, the transmitted byte sequence will be $h_1, \ldots, h_N, b_1, \ldots, b_k$. The first N bytes constitute a *packet header*, where for each i, $1 \leq i \leq N$, $h_i = \lceil k/256^{N-i} \rceil \bmod 256$. (That is, the packet header encodes the length of the byte sequence, k, as a big-endian numeral, cf. p. 11.) However, if $k \geq 256^N$, no byte sequence at all will be transmitted to the external resource and instead an exit signal `badsig` will be sent to `owner[`R`]`.

If the external resource is a file opened for writing, then the transmitted byte sequence written to the file (but the file is not closed until the port is closed). If the external resource is an opened driver or an external process, it can read the transmitted byte sequence using the *read* function. However, no assumption should be made about how many bytes of data will be read by each invocation of the *read* function. Note that the transmission of a byte sequence is atomic in that the bytes of a single transmission (including any prepended bytes) will never be separated by bytes from other transmissions.

12.7 Transmitting data from the outside to an Erlang process

When the external resource to which a port R is opened is a file and `in[`R`]` is `true`, then the contents of the file is written to the port as if by a single invocation of *write*. We will now describe what happens when an opened driver or an external process writes data to a port using the *write* function.

If `out[R]` is `false`, then nothing is transmitted. Otherwise messages will be sent to `owner[R]`.

Let the complete sequence of bytes written (through one or more invocations of *write*) by the driver or external process be $\mathtt{b}_1, \ldots, \mathtt{b}_n$. Depending on the value of `packeting[R]`, the byte sequence may be interpreted as containing packet headers, in which case each packet will be delivered to `in[R]` as a separate message.

- If `packeting[R]` is `stream`, all n bytes will be delivered but it is not specified how the byte sequence is to be divided into messages. Each message i (where $i \geq 1$) contains the bytes $\mathtt{b}_{k_i}, \ldots, \mathtt{b}_{k_{i+1}-1}$ where $k_1 = 1$ and for each i, either $k_i \leq n$ and $k_i < k_{i+1}$, or $k_i = n + 1$. (Thus there should be no empty messages.) It is encouraged that each invocation of *write* (with a nonempty sequence of bytes) should cause a message to be sent, in order to facilitate interaction between the driver or external process and the Erlang processes.

- If `packeting[R]` is `{packet,N}`, the sequence $\mathtt{b}_1, \ldots, \mathtt{b}_n$ is interpreted as containing packet headers of $\mathtt{N}$ bytes each. Each message i (where $i \geq 1$) will contain the bytes $\mathtt{b}_{k_i+\mathtt{N}}, \ldots, \mathtt{b}_{k_{i+1}-1}$ where $k_1 = 1$ and for each i, either $k_i \leq n$,

  $$l_i = \mathit{BigEndianValue}(\mathtt{b}_{k_i}, \ldots, \mathtt{b}_{k_i+\mathtt{N}})$$

  (cf. p. 11) and $k_i+\mathtt{N}+l_i = k_{i+1}$, or $k_i = n+1$. (That is, the first $\mathtt{N}$ bytes are interpreted as a packet header and the subsequent bytes [where the number of bytes is given by the packet header] will be contained in a single message. Then comes another packet header, etc. Note that there can be empty messages.) It is encouraged that a message should be sent as soon as all the bytes of a package have been written, in order to facilitate interaction between the driver or external process and the Erlang processes.

Each message to `owner[R]` is on the form `{R,T}` where `T` contains k transmitted bytes, but the term `T` depends on whether `in_format[R]` is `binary` or `list`:

- If `in_format[R]` is `binary`, then `T` is a binary consisting of the k bytes in the order they were written.

- If `in_format[R]` is `list`, then `T` is a list of the k bytes in the order they were written.

12.8 Closing ports

A port can be closed (through an implementation-defined method) by the external resource. It can also be closed by an ERLANG process using the BIF `port_close/1` (§13.12.2).

The options `eof` and `noeof` to the BIF `open_port/2` (§12.4) determine how the ERLANG process that owns the port is notified if the port is closed by the external resource.

12.9 Ports, links and exit signals

An ERLANG process may be linked to a port in the same way that it may be linked to a process (§10.3). A process *P* may set up (or remove) a link with a port *R* by calling the BIF `link/1` (or `unlink/1`) with *R* as argument. When a port is closed, an exit signal will be sent to every process linked to the port. The actual exit signal is implementation-defined, except that implementations are encouraged to use the exit signal `normal` when the driver or external process completes normally.

When a port *R* receives an exit signal *T* from a process *P*, the following happens.

- If *T* is `normal` and *P* is not `owner[`*R*`]`, then nothing happens.
- Otherwise, if *T* is `kill`, the port is closed and an exit signal ***`killed`*** is sent to every process that is linked to the port.
- Otherwise, the port is closed and an exit signal *T* is sent to every process that is linked to the port.

Note that this behaviour is somewhat different from how a process handles incoming exit signals (§10.4.3).

12.10 Static and dynamic properties of a port

When a port is created, some properties of the port are determined and will be in effect until the port is closed. During that time also a dynamic state is maintained, consisting of properties of the port that change as time passes. This state is affected by and affects the behaviour of the port.

We refer to these as the *static* and *dynamic properties* of a port. We refer collectively to the values of the latter at a certain time as the *state* of the port at that time.

12.10.1 Static properties

`command[R]`
: For a port that was opened through a call to `open_port/2` with `{spawn,`*`Cmd`*`}`, the value of `command[`*`R`*`]` is a string that is either *`Cmd`*, if *`Cmd`* is a string, or the printname of *`Cmd`*, if *`Cmd`* is an atom. It can be accessed through a BIF call `erlang:port_info(`*`R`*`,name)` (§13.12.4).

`creation[R]`
: The value of `creation[`*`R`*`]` is the value of `creation[`*`N`*`]` for the node *`N`* on which *`R`* was created.

`ID[R]` The value of `ID[`*`R`*`]` is a nonnegative integer that is a serial number for *`R`* on the node on which it was created. The value is used in the transformation to the external term format (§D). It can be accessed through a BIF call `erlang:port_info(`*`R`*`,id)` (§13.12.4).

`in[R]` When a port is opened it is decided whether data can be transmitted from the outside to the process owning the port. The value is `true` if one of the options `in` and `bi` was given when the port was opened and `false` otherwise. The value of `in[`*`R`*`]` cannot be accessed directly by ERLANG processes.

`in_format[R]`
: When a port is opened it is decided whether incoming data is sent as binaries or as lists of bytes. The value is `binary` or `list` depending on which of these options was given when the port was opened. The value of `in_format[`*`R`*`]` cannot be accessed directly by ERLANG processes.

`node[R]`
: When a port is opened it is created on some node `node[`*`R`*`]`. This node never changes. Any process can access `node[`*`R`*`]` for a port *`R`* by calling the BIF `node/1` (§13.10.7) with *`R`* as argument.

`out[R]` When a port is opened it is decided whether data can be transmitted from ERLANG processes to the outside. The value is `true` if one of the options `out` and `bi` was given when the port was opened and `false` otherwise. The value of `out[`*`R`*`]` cannot be accessed directly by ERLANG processes.

`packeting[R]`
: When a port is opened it is decided whether data is transmitted as a stream or as packets beginning with a packet header of some length. The value is `stream` or `{packet,`*`N`*`}` (where *`N`* is `1`, `2` or `4`) depending on which of these options was given when the port was opened. The value of `packeting[`*`R`*`]` cannot be accessed directly by ERLANG processes.

12.10.2 Dynamic properties

`count_in[R]`

`count_in[`*R*`]` is an integer that is the number of bytes read so far from the port *R*. It can be accessed through a BIF call

```
erlang:port_info(R,input)
```

(§13.12.4).

`count_out[R]`

`count_out[`*R*`]` is an integer that is the number of bytes written so far to the port *R*. It can be accessed through a BIF call

```
erlang:port_info(R,output)
```

(§13.12.4).

`linked[R]`

The value is a representation of the set of PIDs that identify the processes to which *R* is linked (§10.3). It cannot be set directly but a PID *P* will be added to `linked[`*R*`]` if it is not already in it and *R* receives a linking request from *P*.

A PID *P* will be removed from `linked[`*R*`]` if it is in the list and either

- *P* receives an unlinking request from *Q*, or
- *P* receives an exit signal `{'EXIT',`*Q*`,`*Reason*`}` for some term *Reason*, and the exit signal was sent due to process completion (§10.4).

The value of `linked[`*R*`]` can be accessed through a BIF call

```
erlang:port_info(R,links)
```

(§13.12.4).

`owner[R]`

The value is a PID and is the process that will receive messages when data is written to the port *R* externally, will receive exit signals when bad messages are sent to *R* and becomes linked with *R* when *R* is opened. `owner[`*R*`]` can be changed to a PID *P* by sending a message `{owner[`*R*`],{connect,`*P*`}}` to *R*. The value of `owner[`*R*`]` can be accessed through a BIF call

```
erlang:port_info(R,connected)
```

(§13.12.4).

There will typically also be buffers for inbound and/or outbound data but these are not referred to above and we do not describe them in detail here.

Chapter 13

Builtin functions

As for any function application, when an application of a BIF is evaluated all arguments are fully evaluated before the BIF itself is called and begins executing. When below we discuss the evaluation of a BIF with k parameters, we assume that the k arguments have already been evaluated and that their values are v_1, ... , v_k.

For convenience we abuse our language slightly and may write

- "calling a BIF" when we mean "evaluating an application of a BIF" (where the values of the k arguments are the terms v_1, ... , v_k),
- "the BIF returns ... " when we mean "the evaluation of an application of the BIF completes normally with result ... ", and
- "the BIF exits with reason ... " when we mean "the evaluation of an application of the BIF exits with reason ... ".

In this chapter we write that a BIF "exits with cause *R*" to express that the evaluation of an application of the BIF completes abruptly with reason

`{'EXIT',{R,{M,F,[v1,...,vk]}}`,

where *M* is the name of the module and *F* is the symbol of the function that called the BIF, and as usual, v_1, ... , v_k are the values of the arguments.

If the abnormal completion was because there was something wrong with the value of an argument, e.g., is was of the wrong type or it was an index outside the meaningful range, then *R* is always the atom `badarg`.

If it is not explicitly said about a BIF that it is a guard BIF, then it is not a guard BIF.

13.1 Recognizer BIFs

In Erlang 4.7.3 the recognizer BIFs are not true BIFs and can only be used in guards as *GuardRecognizer*. There are twelve guard recognizers:

`atom/1`, `binary/1`, `constant/1`, `float/1`, `function/1`, `integer/1`, `list/1`, `number/1`, `pid/1`, `port/1`, `reference/1`, and `tuple/1`. As they all behave similarly, we describe them collectively.

Evaluation

If v_1 is one of the terms indicated in Table 13.1, then the recognizer in the guard succeeds, otherwise it fails.

Recognizer	Succeeds if and only if the argument is
`atom/1`	an atom
`binary/1`	a binary
`constant/1`	of an elementary type (cf. §4)
`float/1`	a float
`integer/1`	an integer
`function/1`	a function term
`list/1`	a cons or nil
`number/1`	a number
`pid/1`	a PID
`port/1`	a port
`reference/1`	a reference
`tuple/1`	a tuple

Table 13.1: Erlang recognizer BIFs.

Note that the name of the recognizer `list/1` is inaccurate: it does *not* test whether its argument is a list. (It succeeds for all lists but also for some terms that are not lists, such as `[a|b]`).

Note that although a *GuardRecordTest* `record(`*E*`,`*R*`)` §6.20.1 is not an application of a BIF, it is used as a recognizer for records of a certain type. In Erlang 4.7.3 function terms are implemented with tuples.

13.2 Builtin functions on atoms

13.2.1 `atom_to_list/1`

An atom is converted to a list of characters.

Type

```
atom_to_list(atom()) -> [int()].
```

Exits

Exits with cause `badarg` if v_1 is not an atom.

Evaluation

The BIF returns the printname of the atom v_1 represented by a list of characters.

Examples

`atom_to_list(foo)` $\Rightarrow$ `[102,111,111]`, i.e., `"foo"`;
`atom_to_list('')` $\Rightarrow$ `[]`, i.e., `""`;
`atom_to_list('T %@\'#')` $\Rightarrow$ `[84,32,37,64,39,35]`, i.e., `"T %@'#"`;
`atom_to_list('456')` $\Rightarrow$ `[52,53,54]`, i.e., `"456"`;
`atom_to_list(456)` $\leadsto$ `{badarg,...}`.

13.2.2 `list_to_atom/1`

A list of characters is converted to an atom.

Type

`list_to_atom([int()]) -> atom().`

Exits

Exits with cause `badarg` if v_1 is not a list of characters.

Evaluation

The BIF returns the atom that has a printname consisting of the characters in v_1.

Examples

`list_to_atom([102,111,111])`, i.e., `list_to_atom("foo")` $\Rightarrow$ `foo`;
`list_to_atom([])`, i.e., `list_to_atom("")` $\Rightarrow$ `''`;
`list_to_atom([84,32,37,64,39,35])`, i.e., `list_to_atom("T %@'#")` $\Rightarrow$ `'T %@\'#'`;
`list_to_atom([52,53,54])`, i.e., `list_to_atom("456")` $\Rightarrow$ `'456'`;
`list_to_atom([102,-5,111])` $\leadsto$ `{badarg,...}`.

13.3 Builtin arithmetic functions

13.3.1 `abs/1`

The magnitude of a number is computed. `abs/1` is a guard BIF.

Type

```
abs(int()) -> int() ;
abs(float()) -> float().
```

Exits

Exits with cause `badarg` if v_1 is not a number. May exit with `integer_overflow` when v_1 is an integer (see below).

Evaluation

If v_1 is an integer, compute $abs_I(\Re^{-1}[v_1])$; otherwise v_1 is a float, compute $abs_F(\Re^{-1}[v_1])$. Let r be the result. If r is a number, the the BIF returns $\Re[r]$; otherwise the BIF exits with cause $\Re[r]$.

Examples

`abs(42)` $\Rightarrow$ `42`;
`abs(-88)` $\Rightarrow$ `88`;
`abs(5.0)` $\Rightarrow$ `5.0`;
`abs(-0.1)` $\Rightarrow$ `0.1`;
`abs(whoopee)` $\rightsquigarrow$ `{badarg,...}`.

13.3.2 `float/1`

There are two BIFs named `float/1` and which one of them is denoted in an application `float(E)` depends on the context in which the application appears.

If the application is a *GuardRecognizer* expression (§6.20), then it is the guard test `float/1` described in §13.1; otherwise it is a function converting numbers to floating-point numbers (which may appear in guard expressions). The following description is for the latter case.

Both BIFs `float/1` are guard BIFs.

Type

```
float(num()) -> float().
```

Exits

Exits with cause `badarg` if v_1 is not a number. May exit with `float_overflow` when v_1 is an integer (see below).

Evaluation

The evaluation depends on the type of v_1:

- If v_1 is a float, it is returned.
- If v_1 is an integer, compute $cvt_{I \to F}(\Re^{-1}[v_1])$, let the result be r. If r is a float, the the BIF returns $\Re[r]$; otherwise the BIF exits with cause $\Re[r]$.

Examples

`float(3)` ⇒ `3.0`;
`float(0)` ⇒ `0.0`;
`float(123456789123456789123456789)` ⇒ `1.23456789123457E26`;
`float(whoopee)` $\rightsquigarrow$ `{badarg,...}`.

13.3.3 `float_to_list/1`

The function produces a list of characters of a printed representation of a float.

Type

`float_to_list(float()) -> [int()]`.

Exits

Exits with cause `badarg` if v_1 is not a float (there is thus no coercion from integers).

Evaluation

The BIF returns the list of characters which is like the canonical decimal numeral for $\Re^{-1}[v_1]$ (§5.9.4) except that:

- There are exactly 20 digits between the decimal point and the 'e'. The right end is adjusted by dropping digits or padding with zeroes.
- If the character after 'e' is not a minus sign, then a plus sign is inserted.
- If the exponent part (i.e., the digits after 'e') has only one digit, a zero is inserted before that digit.

Examples

`float_to_list(-0.00672)` ⇒
`[45,54,46,55,50,48,48,48,48,48,48,48,48,48,48,48,48,48,50,55,56,54,55,101,45,48,51]`,
i.e., `"-6.72000000000000027867e-03"`;
`float_to_list(13e4)` ⇒
`[49,46,51,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,101,43,48,53]`,
i.e., `"1.30000000000000000000e+05"`;

`float_to_list(0.0)` ⇒
`[48,46,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,101,43,48,48]`,
i.e., `"0.00000000000000000000e+00"`;
`float_to_list(636.9e121)` ⇒
`[54,46,51,54,56,57,57,57,57,57,57,57,57,57,57,57,57,53,52,53,53,51,101,43,49,50,51`
i.e., `"6.36899999999999954553e+123"`;
`float_to_list(whoopee)` ⇝ `{badarg,...}`.

13.3.4 `integer_to_list/1`

The function produces a list of characters of a printed representation of an integer.

Type

`integer_to_list(int()) -> [int()].`

Exits

Exits with cause `badarg` if v_1 is not an integer.

Evaluation

The BIF returns the list of characters making up the canonical decimal numeral for $\Re^{-1}[v_1]$ (§5.9.1).

Examples

`integer_to_list(0)` ⇒ `[48]`, i.e., `"0"`;
`integer_to_list(42)` ⇒ `[52,50]`, i.e., `"42"`;
`integer_to_list(-39)` ⇒ `[45,51,57]`, i.e., `"-39"`;
`integer_to_list(whoopee)` ⇝ `{badarg,...}`.

13.3.5 `list_to_float/1`

A float is obtained from a float literal represented by a list of the characters of the literal.

Type

`list_to_float([int()]) -> float().`

Exits

Exits with cause `badarg` if v_1 is not a list of characters making up a float numeral, or if the number it represents is not representable as a float (see below). Leading or trailing spaces are not permitted.

Evaluation

If the sequence of characters denotes a number $f \in F$ (§5.9.5), then $\Re[f]$ is returned, otherwise the BIF exits with cause `badarg`.

Examples

`list_to_float([45,49,55,46,53,48,48])`, i.e.,
`list_to_float("-17.500")` ⇒ `-17.5`;
`list_to_float([49,50,51,46,52,53,101,56,55])`, i.e.,
`list_to_float("123.45e87")` ⇒ `1.23450e89`;
`list_to_float([54,50,46,53,69,45,51])`, i.e.,
`list_to_float("62.5E-3")` ⇒ `0.0625`;
`list_to_float([54,50,53,101,45,52])`, i.e.,
`list_to_float("625e-4")` ⇝ `{badarg,...}`;
`list_to_float(whoopee)` ⇝ `{badarg,...}`.

13.3.6 `list_to_integer/1`

An integer is obtained from an integer literal represented by a list of the characters of the literal.

Type

`list_to_integer([int()]) -> integer().`

Exits

Exits with cause `badarg` if v_1 is not a list of characters making up a decimal integer numeral, or if the integer it represents is too large to be represented (see below). Leading or trailing spaces are not permitted.

Evaluation

If the sequence of characters denotes an integer $i \in I$ (§5.9.2), then $\Re[i]$ is returned, otherwise the BIF exits with cause `badarg`.

Examples

`list_to_integer([52,50])`, i.e., `list_to_integer("42")` ⇒ `42`;
`list_to_integer([48,48,48,48,48,48,53,54,55])`, i.e., `list_to_integer("000000567")` ⇒ `567`;
`list_to_integer([45,51,57])`, i.e., `list_to_integer("-39")` ⇒ `-39`;
`list_to_integer([111,105,110,107])`, i.e., `list_to_integer("oink")` ⇝ `{badarg,...}`;
`list_to_integer(whoopee)` ⇝ `{badarg,...}`.

13.3.7 `round/1`

The integer closest to a given number is computed. `round/1` is a guard BIF.

Type

`round(num()) -> int().`

Exits

Exits with cause `badarg` if v_1 is not a number. May exit with `integer_overflow` when v_1 is a float (see below).

Evaluation

The evaluation depends on the type of v_1:

- If v_1 is an integer, it is returned.
- If v_1 is a float, compute $nearest_{F \to I}(\Re^{-1}[v_1])$, let the result be r. If r is an integer, the the BIF returns $\Re[r]$; otherwise the BIF exits with cause $\Re[r]$.

Examples

`round(42)` $\Rightarrow$ `42`;
`round(88.56)` $\Rightarrow$ `89`;
`round(-88.56)` $\Rightarrow$ `-89`;
`round(0.0)` $\Rightarrow$ `0`;
`round(whoopee)` $\leadsto$ `{badarg,...}`.

13.3.8 `trunc/1`

The first float between a given number and zero is computed. `trunc/1` is a guard BIF.

Type

`trunc(num()) -> int().`

Exits

Exits with cause `badarg` if v_1 is not a number. May exit with `integer_overflow` when v_1 is a float (see below).

Evaluation

The evaluation depends on the type of v_1:

- If v_1 is an integer, it is returned.
- If v_1 is a float, compute $\mathit{truncate}_{F\rightarrow I}(\Re^{-1}[v_1])$, let the result be r. If r is an integer, the the BIF returns $\Re[r]$; otherwise the BIF exits with cause $\Re[r]$.

Examples

`trunc(42)` ⇒ `42`;
`trunc(88.56)` ⇒ `88`;
`trunc(-88.56)` ⇒ `-88`;
`trunc(0.0)` ⇒ `0`;
`trunc(whoopee)` ⇝ `{badarg,...}`.

13.4 Builtin functions on binaries

Binaries are described in §4.5.

13.4.1 `binary_to_list/1`

A list of bytes that are the elements of a binary, in the same order as in the binary, is returned.

Type

`binary_to_list(bin()) -> [int()].`

Exits

`binary_to_list/1` exits with cause `badarg` if v_1 is not a binary.

Evaluation

Suppose that v_1 is a binary consisting of the bytes I_1, ..., I_k. A list `[`I_1`,...,`I_k`]` is returned. The time for computing the answer should be $O(k)$.

13.4.2 `binary_to_list/3`

A list of integers that are part of the elements of a binary, in the same order as in the binary, is returned.

Type

`binary_to_list(bin(),int(),int()) -> [int()].`

Exits

`binary_to_list/3` exits with cause `badarg` if v_1 is not a binary, if v_2 or v_3 is not an integer. It exits with cause `badarg` if the integers represented by v_2 and v_3 are out of range (see below).

Evaluation

Suppose that v_1 is a binary consisting of the bytes I_1, ... , I_k and that $i = \Re^{-1}[v_2]$ and $j = \Re^{-1}[v_3]$. `binary_to_list/2` exits with cause `badarg` if $i < 1$, $j < i$ or $j > k$. A list $[I_i, \ldots, I_j]$ is returned. The result is thus always a nonempty list.

The time for computing the result should be $O(j)$.

13.4.3 `binary_to_term/1`

Given a binary that is a representation in the external format (§D) of a term, that term is returned.

Type

```
binary_to_term(bin()) -> term().
```

Exits

`binary_to_term/1` exits with cause `badarg` if v_1 is not a binary. It also exits with cause `badarg` if the elements of v_1 are not the external format representation of some term (see below).

Evaluation

If the bytes that are the elements of v_1 constitute a representation in the external format (§D) of some term t, then t is returned; otherwise `binary_to_term/1` exits with cause `badarg`.

The time for computing the result should be $O(k)$.

13.4.4 `concat_binary/1`

Given a list of binaries, one binary is returned which has as elements the elements of the binaries in the list, in the same order as in which the elements appear in the binaries and the binaries in the list.

Type

```
concat_binary([bin()]) -> bin().
```

Exits

`concat_binary/1` exits with cause `badarg` if v_1 is not a list of binaries.

Evaluation

Suppose that v_1 is a list `[`B_1`,...,`B_k`]`, such that each B_i, $1 \leq i \leq k$, is a binary consisting of the bytes $b_{i,1}, \ldots, b_{i,n_i}$. A binary consisting of the bytes $b_{1,1}, \ldots, b_{1,n_1}, \ldots, b_{k,1}, \ldots, b_{k,n_k}$ is returned.

The time for computing the result should be $O(\max(\sum_{i=1}^{k} n_i, k))$.

13.4.5 `list_to_binary/1`

Given a list of bytes, a binary consisting of the same bytes in the same order is returned.

Type

`list_to_binary([int()]) -> bin().`

Exits

`list_to_binary/1` exits with cause `badarg` if v_1 is not a list of bytes.

Evaluation

Suppose that v_1 is a list of bytes `[`I_1`,...,`I_k`]`. A binary consisting of the bytes $I_1, \ldots, I_k$ is returned.

The time for computing the result should be $O(k)$.

13.4.6 `size/1`

See §13.5.4.

13.4.7 `split_binary/2`

A binary is split into two binaries, where the number of elements in the first binary is given.

Type

`split_binary(bin(),int()) -> {bin(),bin()}.`

Exits

`split_binary/2` exits with cause `badarg` if v_1 is not a binary or v_2 is not an integer. It also exits with cause `badarg` if the number represented by v_2 is out of range (see below).

Evaluation

Suppose that v_1 is a binary consisting of the bytes $I_1, \ldots, I_k$ and that $i = \Re^{-1}[v_2]$. The evaluation depends on i and k:

- If $i < 0$ or $i > k$, then `split_binary/2` exits with cause `badarg`.
- Otherwise a 2-tuple of binaries $\{B_l, B_r\}$ is returned, where B_l consists of the bytes $I_1, \ldots, I_i$ and B_r consists of the bytes $I_{i+1}, \ldots, I_k$. (If $i = 0$ then B_l is an empty binary and B_r equals v_1, while if $i = k$ then B_l equals v_1 and B_r is an empty binary.)

The time for computing the answer should be $O(1)$.

13.4.8 `term_to_binary/1`

Given a term, a binary having elements that represent the term in the external format (§D) is returned.

Type

```
term_to_binary(term()) -> bin().
```

Exits

`term_to_binary/1` always completes normally.

Evaluation

Let the representation in the external format (§D) of v_1 be the sequence of bytes $I_1, \ldots, I_k$. A binary consisting of these bytes, in that order, is returned. The time for computing the answer should be $O(k)$.

13.5 Builtin functions on tuples

Tuples are described in §4.8.

13.5.1 `element/2`

One element of a tuple is returned. `element/2` is a guard BIF.

Type

```
element(int(),tuple()) -> term().
```

Exits

`element/2` exits with cause `badarg` if v_1 is not an integer or v_2 is not a tuple. It also exits with cause `badarg` if the number represented by v_1 is out of range (see below).

Evaluation

Suppose that $i = \Re^{-1}[v_1]$ and that v_2 is a tuple with elements $T_1, \ldots, T_k$. The evaluation depends on i and k:

- If $i < 1$ or $i > k$, then `element/2` exits with cause `badarg`.
- Otherwise, T_i is returned.

The time for computing the result should be $O(1)$.

13.5.2 `list_to_tuple/1`

A tuple with the same elements as a given list is returned.

Type

```
list_to_tuple([term()]) -> tuple().
```

Exits

`list_to_tuple/1` exits with cause `badarg` if v_1 is not a list.

Evaluation

Let v_1 be a list with elements $T_1, \ldots, T_k$. A tuple `{`T_1`,...,`T_k`}` is returned. The time for computing the answer should be $O(k)$.

13.5.3 `setelement/3`

A tuple is computed that differs from a given tuple in exactly one element.

Type

```
setelement(int(),tuple(),term()) -> tuple().
```

Exits

`setelement/3` exits with cause `badarg` if v_1 is not an integer or v_2 is not a tuple. It also exits with cause `badarg` if the number represented by v_1 is out of range (see below).

Evaluation

Suppose that $i = \Re^{-1}[v_1]$ and that v_2 with elements $T_1, \ldots, T_k$. The evaluation depends on i and k:

- If $i < 1$ or $i > k$, then `setelement/3` exits with cause `badarg`.
- Otherwise, a tuple `{`T_1`,...,`T_{i-1}`,`v_3`,`T_{i+1}`,`T_k`}` is returned. (That is, a tuple that is exactly like v_2 except that the element at position i is v_3.)

The time for computing the answer should be $O(k)$. (This is not a destructive operation so the tuple given as argument must not be observably affected.)

13.5.4 `size/1`

The number of elements of a binary or a tuple is returned. `size/1` is a guard BIF.

Type

```
size(bin()) -> int() ;
size(tuple()) -> int().
```

Exits

`size/1` exits with cause `badarg` if v_1 is neither a binary, nor a tuple.

Evaluation

The integer $\Re[k]$ is returned, where k is the number of elements in the binary or tuple v_1. The time for computing the answer should be $O(1)$.

13.5.5 `tuple_to_list/1`

A list with the same elements as a given tuple is returned.

Type

```
tuple_to_list(tuple()) -> [term()].
```

Exits

`tuple_to_list/1` exits with cause `badarg` if v_1 is not a tuple.

Evaluation

Let v_1 be a tuple with elements $T_1, \ldots, T_k$. A list `[`T_1`,...,`T_k`]` is returned. The time for computing the result should be $O(k)$.

13.6 Builtin functions on lists and conses

Lists and conses are described in §4.9.

13.6.1 `hd/1`

The head of a cons, e.g., the first element of a list, is returned. `hd/1` is a guard BIF.

Type

If cons is used as intended, the type is

`hd([T]) -> T.`

If cons is used as a general pairing operator, the type is instead

`hd([T|_]) -> T.`

Exits

`hd/1` exits with cause `badarg` if v_1 is not a cons.

Evaluation

The head of the cons v_1 is returned. The time for computing the result should be $O(1)$.

13.6.2 `length/1`

The number of elements of a list is returned. `length/1` is a guard BIF.

Type

`length([term()]) -> int().`

Exits

`length/1` exits with cause `badarg` if v_1 is not a list.

Evaluation

The integer $\Re[k]$ is returned, where k is the number of elements in the list. The time for computing the answer should be $O(1)$.

13.6.3 `tl/1`

The tail of a cons, e.g., all but the first element of a list, is returned. `tl/1` is a guard BIF.

Type

If cons is used as intended, the type is

`tl([T]) -> [T].`

If cons is used as a general pairing operator, the type is instead

`tl([_|T]) -> T.`

Exits

`tl/1` exits with cause `badarg` if v_1 is not a cons.

Evaluation

The tail of the cons v_1 is returned. The time for computing the result should be $O(1)$.

13.7 Builtin functions for modules

The syntax of modules is described in §8.1 and their dynamics in §9.

The BIFs in this section are not designed to be used directly in applications. Rather, they are provided for implementing more high-level interface to dynamic loading and replacement of modules.[1]

13.7.1 `erlang:check_process_code/2`

A check is made whether a particular process is using a given module (§9.1).

Type

`erlang:check_process_code(pid(),atom()) -> bool().`

Exits

`erlang:check_process_code/2` exits with cause `badarg` if v_1 is not a pid or v_2 is not an atom.[2]

Evaluation

Let N be `node[`v_1`]`.

- If module_table[N] contains a row with v_2 as key and R as value, `old_version[`R`]` is not `none` and a reference to `old_version[`R`]` is found in `stack_trace[`P`]`, then the BIF returns `true` (cf. §9.7).

[1] Cf. the `code` module of OTP [6, pp. 158–167].

[2] ERLANG 4.7.3 actually allows v_2 to be anything and always returns `false` if v_2 is not an atom.

- Otherwise, it returns `false`.

13.7.2 `erlang:delete_module/1`

The current version of a module is changed to be the old version (§9.1).

Type

`erlang:delete_module(atom()) -> atom().`

Exits

`erlang:delete_module/1` exits with cause `badarg` if v_1 is not an atom. Moreover, it will exit with cause `badarg` if there is already an old version of the module (see below)

Evaluation

Let *P* be the process calling `delete_module/1` and let *N* be `node[`*P*`]`.

- If there is no row with key v_1 in `module_table[`*N*`]`, then `undefined` is returned.[3] Otherwise, let *R* be the value for v_1 in `module_table[`*N*`]`.
- If `old_version[`*R*`]` is not `none`, the BIF exits with cause `badarg`.
- Otherwise, if `current_version[`*R*`]` is not `none`, it is made the old version of module v_1 as described in §9.5 and `true` is returned.
- Otherwise, no action is taken and `true` is returned.

13.7.3 `erlang:load_module/2`

A compiled module is loaded as the current version of the module (§9.1). If there is already a current version of the module, it is made the old version of the module.

Type

`erlang:load_module(atom(),bin()) -> atom().`

Exits

`erlang:load_module/2` exits with cause `badarg` if v_1 is not an atom or v_2 is not a binary.

[3]This is contrary to the reasonable intuition that the BIF should return `undefined` if, and only if, `module_loaded/1` returns `false` for the module.

Evaluation

Let P be the process calling `load_module/2` and let N be `node[`P`]`.

- If the binary v_2 does not contain compiled code for a module named v_1, the BIF returns a tuple `{error, badfile}`.
- Otherwise, the following is done:

  First, if `module_table[`N`]` contains no row with v_1 as key, a row is added with v_1 as key and R as value, such that `old_version[`R`]` and `current_version[`R`]]` are both `none`.

  Otherwise, let R be the value of the row with v_1 as key; if neither of `old_version[`R`]` and `current_version[`R`]]` is `none`, the BIF returns a tuple `{error, not_purged}`.

  Next, if `current_version[`R`]` is not `none`, it is made the old version of module v_1 on node N as described in §9.5.

  Finally, the binary v_2 is made the current version of module v_1 on node N as described in §9.4.

13.7.4 `erlang:preloaded/0`

A list is returned of the names of all modules that were loaded as part of starting the current node.

Type

`erlang:preloaded() -> [atom()].`

Exits

`erlang:preloaded/0` always completes normally.

Evaluation

The BIF returns a list representing the value of `preloaded[node[`P`]]`, where P is the process calling the BIF.

13.7.5 `erlang:purge_module/1`

The old version of a module is purged (§9.1).

Type

`erlang:purge_module(atom()) -> true.`

Exits

`erlang:purge_module/1` exits with cause `badarg` if v_1 is not an atom. Moreover, it will exit with cause `badarg` if there is no old version of the module (see below)

Evaluation

Let `P` be the process calling `purge_module/1` and let `N` be `node[P]`.

- If there is a row in `module_table[N]` with v_1 as key and some `R` as value, and `old_version[R]` is not `none`, then `old_version[R]` is purged as described in §9.6 and `true` is returned.
- Otherwise, the BIF exits with cause `badarg`.

13.7.6 `erlang:module_loaded/1`

It is found out whether there is a current version (§9.1) of some module.

Type

```
erlang:module_loaded(atom()) -> bool().
```

Exits

`erlang:module_loaded/1` exits with cause `badarg` if v_1 is not an atom.

Evaluation

Let `P` be the process calling `module_loaded/1` and let `N` be `node[P]`.

- If there is a row in `module_table[N]` with v_1 as key and some `R` as value, and `current_version[R]` is not `none`, then the BIF returns `true`.
- Otherwise, the BIF returns `false`.

13.8 Builtin functions for functions and processes

13.8.1 `apply/2`

A given function is applied to a sequence of arguments.

Type

```
apply({atom(),atom()},[term()]) -> term() ;
apply(function(),[term()]) -> term()
```

Exits

`apply/2` exits with cause `badarg` if v_1 is neither a 2-tuple of atoms, nor a function, or v_2 is not a list. `apply/2` may also exit with the reasons described in §6.7. (In addition, the function being applied may complete abnormally with any reason.)

Evaluation

The evaluation depends on the type of the first argument:

- If v_1 is a 2-tuple of atoms *`Mod`* and *`Fun`*, then evaluation proceeds as described by case 1 in §6.7 with the atoms *`Mod`* and *`Fun`* specifying the module name and function symbol, respectively, and the list v_2 specifying the values of the arguments (and the arity).
- If v_1 is a function, then evaluation proceeds as described by case 3 or 4 in §6.7 with v_1 being the function and the list v_2 specifying the values of the arguments.

13.8.2 `apply/3`

Given a module name, a function symbol and a list of arguments, a function is looked up and applied to the arguments.

Type

```
apply(atom(),atom(),[term()]) -> term().
```

Exits

`apply/3` exits with cause `badarg` if v_1 is not an atom, v_2 is not an atom, or v_3 is not a list. `apply/3` may also exit with the reasons described in §6.7. (In addition, the function being applied may complete abnormally with any reason.)

Evaluation

The evaluation of `apply/3` is described by case 1 in §6.7 with the atoms v_1 and v_2 specifying the module name and function symbol, respectively, and the list v_3 specifying the values of the arguments.

13.8.3 `exit/1`

The BIF always exits with the argument as reason. Unless the application is governed by a `catch` expression (§6.9), the process calling the BIF will exit.

Type

`exit(term()) -> _.`

Exits

`exit/1` always exits, see below.

Evaluation

The evaluation of `exit/1` exits with reason v_1.

13.8.4 `exit/2`

An exit signal with the given reason is sent to a process or port. Reception of the exit signal will cause the receiving process to complete abruptly unless it traps exit signals or the reason is the atom `normal`, cf. §10.4.3. If the reason is the atom `kill`, the receiving process will always complete abruptly.

Type

```
exit(pid(),term()) -> true ;
exit(port(),term()) -> true.
```

Exits

`exit/2` exits with cause `badarg` if v_1 is neither a PID, nor a port.

Evaluation

An exit signal with v_2 as reason is sent to the process or port identified by v_1, as described in §10.4.2. The result is always the atom `true`.

13.8.5 `group_leader/0`

The BIF returns the group leader (§10.8) of the calling process.

Type

`group_leader() -> pid().`

Exits

`group_leader/0` always completes normally.

Evaluation

The result is `group_leader[`P`]`, where P is the process calling the BIF.

13.8.6 `group_leader/2`

The BIF changes the group leader of a process.

Type

`group_leader(pid(),pid()) -> true`.

Exits

`group_leader/2` exits with cause `badarg` if v_1 or v_2 is not a PID.

Evaluation

A group leader signal (§10.6) is sent to v_2 with v_1 as additional data. When the signal is received by v_2, `group_leader[`v_2`]` will be set to v_1 (§10.6.3).

The result is always the atom `true`.

13.8.7 `link/1`

A request to add a link between the process calling the BIF and some process or port is submitted.

Type

`link(pid()) -> true` ;
`link(port()) -> true`.

Exits

`link/1` exits with cause `badarg` if v_1 is not a PID or a port.

Evaluation

Let P be the process evaluating the application of `link/1`.

- If $P \neq v_1$ and v_1 is not in `linked[`P`]`, then v_1 is added to `linked[`P`]` and a *link* signal with P as sender is dispatched to process v_1 (§10.3, §10.6).
- Otherwise, nothing is done.

The result is always the atom `true`.

13.8.8 `list_to_pid/1`

A list of characters is converted to a PID.

Type

```
list_to_pid([int()]) -> pid().
```

Exits

`list_to_pid/1` exits with cause `badarg` if v_1 is not a list of characters that represents a PID.

Evaluation

The result of a BIF call of `list_to_pid/1` is a PID *P* such that the value of `pid_to_list(P)` (cf. §13.8.9) on the same node equals v_1.

13.8.9 `pid_to_list/1`

A PID is converted to a list of characters.

Type

```
list_to_pid(pid()) -> [int()].
```

Exits

`list_to_pid/1` exits with cause `badarg` if v_1 is not a PID.

Evaluation

The result of a BIF call of `list_to_pid/1` is some list of characters. It is guaranteed that for any PID *P*, the value of an expression `list_to_pid(pid_to_list(P))` equals *P*.

13.8.10 `process_flag/2`

The value of a process flag is read and updated.

Type

```
process_flag(trap_exit,bool()) -> bool() ;
process_flag(error_handler,atom()) -> atom() ;
process_flag(priority,atom()) -> atom().
```

Exits

`process_flag/2` exits with cause `badarg` if v_1 and v_2 is not one of the following combinations:

- The atom `trap_exit` and a Boolean atom.

- The atom `error_handler` and a module name (i.e., an atom).
- The atom `priority` and a priority atom, i.e., one of `normal`, `high` and `low` §10.7).

Evaluation

The action depends on v_1:

- If v_1 is `trap_exit`, then `trap_exit[`P`]` is set to v_2 and the previous value of `trap_exit[`P`]` is returned.
- If v_1 is `error_handler`, then `error_handler[`P`]` is set to v_2 and the previous value of `error_handler[`P`]` is returned.
- If v_1 is `priority`, then `priority[`P`]` is set to v_2 and the previous value of `priority[`P`]` is returned.

13.8.11 `process_info/1`

Information about various properties of a process is returned.

Type

`process_info(pid()) -> term().`

Exits

`process_info/1` exits with cause `badarg` if v_1 is not a PID.

Evaluation

The BIF returns information about the process v_1.

- If that process is not alive, the result is the atom `undefined`.
- Otherwise, the result is a list of 2-tuples, each of which is the same as the result of an application of the BIF `process_info/2` (§13.8.12) to v_1 and a distinct atom in the left column of Table 13.2. The list should include all properties for which `process_info/2` gives meaningful information.

13.8.12 `process_info/2`

Information about some property of a process is returned, as described in Table 13.2.

Second argument	Information returned about the process
`current_function`	The most recently entered function.
`dictionary`	The process dictionary.
`error_handler`	The error handler module.
`group_leader`	The group leader of the process.
`heap_size`	The current heap size.
`initial_call`	The initial application of the process.
`links`	The processes and ports to which the process is linked.
`memory`	The total amount of memory occupied by the process.
`message_queue_len`	The number of unprocessed messages in the queue.
`messages`	The unprocessed messages in the queue.
`priority`	The scheduling priority of the process.
`reductions`	The current number of reductions.
`registered_name`	The registered name of the process, if any.
`stack_size`	The current stack size.
`status`	The scheduling status: waiting, runnable or running.
`trap_exit`	Whether exit signals are trapped.

Table 13.2: Alternatives for the BIF `process_info/2`.

Type

`process_info(pid(),atom()) -> term().`

Exits

`process_info/2` exits with cause `badarg` if v_1 is not a PID or v_2 is not one of the atoms in the left column of Table 13.2.

Evaluation

- If process v_1 is not alive, the result is the atom `undefined`.
- Otherwise, the BIF returns information about the process v_1 and the result is always a 2-tuple where the first element is v_2:
  - ∗ If v_2 is `current_function`, then return `{current_function, current_function[`v_1`]}`.
  - ∗ If v_2 is `dictionary`, then return `{dictionary,`*`Dict`*`}`, where *`Dict`* is an association list representing the contents of `dictionary[`v_1`]` (cf. the BIF `get/0` [§13.9.3]).
  - ∗ If v_2 is `error_handler`, then return `{error_handler,error_handler[`v_1`]}`.

- If v_2 is `group_leader`, then return `{group_leader,group_leader[`v_1`]}`.
- If v_2 is `heap_size`, then return `{heap_size,heap_size[`v_1`]}`.
- If v_2 is `initial_call`, then return `{initial_call,initial_call[`v_1`]}`.
- If v_2 is `links`, then return `{links,`*`Lst`*`}`, where *`Lst`* is a list representing the value of `linked[`v_1`]`.
- If v_2 is `memory`, then return `{memory,memory_in_use[`v_1`]}`.
- If v_2 is `message_queue_len`, then return `{message_queue_len,` $\Re[l]$`}`, where l is the length of `message_queue[`v_1`]`.
- If v_2 is `messages`, then return `{messages,`*`Lst`*`}`, where *`Lst`* is a list representing the value of `message_queue[`v_1`]` (i.e., a list of the messages in the queue, in the same order).
- If v_2 is `priority`, then return `{priority,priority[`v_1`]}`.
- If v_2 is `reductions`, then return `{reductions,reductions[`v_1`]}`.
- If v_2 is `registered_name`, then return `{registered_name,registered_name[`v_1`]}`.
- If v_2 is `stack_size`, then return `{stack_size,`$\Re[s]$`}`, where s is a measure of the size of `stack_trace[`v_1`]`.
- If v_2 is `status`, then return `{status,status[`v_1`]}`.
- If v_2 is `trap_exit`, then return `{trap_exit,trap_exit[`v_1`]}`.

Let *P* be the process calling the BIF. The behaviour with respect to signals (§10.6) should be as if the result was obtained by *P* sending an *info request* signal to process v_1 with v_2 as additional information, and process v_1 responding with a message to *P* containing the result.

13.8.13 `processes/0`

See §13.10.9.

13.8.14 `self/0`

The PID of the process calling the BIF is returned. `self/0` is a guard BIF.

Type

```
self() -> pid().
```

Exits

`self/0` always completes normally.

Evaluation

The PID of the process calling the BIF is returned.

13.8.15 `spawn/3`

A new process is spawned on the same node.

Type

```
spawn(atom(),atom(),[term()]) -> pid().
```

Exits

`spawn/3` exits with cause `badarg` if v_1 or v_2 is not an atom, or if v_3 is not a list.

Evaluation

Let the elements of v_3 be $T_1, \ldots, T_k$. A new process is spawned on the same node as the process calling the BIF (§10.1). The initial call of the process is v_1:v_2(T_1,...,T_k). The PID of the newly spawned process is returned.

13.8.16 `spawn/4`

A new process is spawned on a particular node.

Type

```
spawn(atom(),atom(),atom(),[term()]) -> pid().
```

Exits

`spawn/4` exits with cause `badarg` if v_1, v_2 or v_3 is not an atom, or if v_4 is not a list.

Evaluation

Let the elements of v_4 be $T_1, \ldots, T_k$. A new process is spawned on node v_1 (§10.1). The initial call of the process is v_2:v_3(T_1,...,T_k). The PID of the newly spawned process is returned.

13.8.17 `spawn_link/3`

A new process, initially linked to its creator, is spawned on the same node.

Type

```
spawn_link(atom(),atom(),[term()]) -> pid().
```

Exits

`spawn_link/3` exits with cause `badarg` if v_1 or v_2 is not an atom, or if v_3 is not a list.

Evaluation

The BIF does exactly the same thing as `spawn/3` (§13.8.15), except that when the BIF returns, the newly spawned process is linked with the process calling `spawn_link/3`.

13.8.18 `spawn_link/4`

A new process, initially linked to its creator, is spawned on a particular node.

Type

`spawn_link(atom(),atom(),atom(),[term()]) -> pid()`.

Exits

`spawn/4` exits with cause `badarg` if v_1, v_2 or v_3 is not an atom, or if v_4 is not a list.

Evaluation

The BIF does exactly the same thing as `spawn/4` (§13.8.16), except that when the BIF returns, the newly spawned process is linked with the process calling `spawn_link/4`.

13.8.19 `unlink/1`

A request to remove any link between the process calling the BIF and some process or port is submitted.

Type

`unlink(pid()) -> true` ;
`unlink(port()) -> true`.

Exits

`unlink/1` exits with cause `badarg` if v_1 is not a PID or a port.

Evaluation

Let P be the process evaluating the application of `unlink/1`.

- If $P \neq v_1$ and v_1 is in `linked[P]`, then v_1 is removed from `linked[P]` and an *unlink* signal with P as sender is dispatched to process v_1.
- Otherwise, nothing is done.

The result is always the atom `true`.

13.9 Builtin functions for process dictionaries

As described in §10.9.2, each process P has associated with it a table `dictionary[P]` (§2.4).

13.9.1 `erase/0`

The process calling the BIF has every row of its dictionary removed but a representation of the previous contents is returned.

Type

```
erase() -> [{term(),term()}].
```

Exits

`erase/0` always completes normally.

Evaluation

Let d be the value of `dictionary[P]`, where P is the process calling `erase/0` and let *`lst`* be an association list representing the contents of d (§2.4). The effect of the call is to remove every row of d and the result is *`lst`*.

The operation must be atomic.

13.9.2 `erase/1`

In the dictionary of the process calling the BIF, the value recorded for a certain key (if any) is erased, but the previously recorded value (or `undefined`) is returned.

Type

```
erase(term()) -> term().
```

Exits

`erase/1` always completes normally.

Evaluation

Let d be the value of `dictionary[P]`, where `P` is the process calling `erase/1`. There are two cases depending on whether v_1 is a key in d or not:

- If d contains a row with v_1 as key and some term T_1 as value, then that row is removed from d and `t` is t_1.
- Otherwise, d is unchanged and `t` is the atom `undefined`.

The result is `t`.

The operation must be atomic.

13.9.3 `get/0`

A representation of the dictionary of the process calling the BIF is returned.

Type

```
get() -> [{term(),term()}].
```

Exits

`get/0` always completes normally.

Evaluation

Let d be the value of `dictionary[P]`, where `P` is the process calling `get/0`. The result of calling `get/0` is a list representing d (§2.4).

13.9.4 `get/1`

The value recorded for a certain key in the dictionary of the process calling the BIF is returned (or `undefined` is returned if there is no value recorded).

Type

```
get(term()) -> term().
```

Exits

`get/1` always completes normally.

Evaluation

Let d be the value of `dictionary[P]`, where `P` is the process calling `get/1`. There are two cases depending on whether v_1 is a key in d or not:

- If d contains a row with v_1 as key and some term T_1 as value, then `t` is t_1.

- Otherwise, `t` is the atom `undefined`.

The result is `t`.

13.9.5 `get_keys/1`

A list of all keys in the dictionary of the process calling the BIF that have a certain value is returned.

Type

```
get_keys(term()) -> [term()].
```

Exits

`get_keys/1` always completes normally.

Evaluation

Let d be the value of `dictionary[P]`, where `P` is the process calling `get/0`. The result of calling `get_keys/1` is a list without duplicates that contains each key of d for which the value is v_1. The elements of the resulting list may be in any order.

13.9.6 `put/2`

In the dictionary of the process calling the BIF, a value is set for some key, replacing any previous value, which is returned (or `undefined` is returned if there was no value recorded previously).

Type

```
put(term(),term()) -> term().
```

Exits

`put/2` always completes normally.

Evaluation

Let d be the value of `dictionary[P]`, where `P` is the process calling `put/2`. There are two cases depending on whether v_1 is a key in d or not:

- If d contains a row with v_1 as key and some term t_1 as value, then that row is replaced with one having v_1 as key and v_2 as value; `t` is t_1.
- Otherwise, a row with key v_1 and value v_2 is added to d and `t` is `undefined`.

The result is t.

The operation must be atomic.

13.10 Builtin functions for nodes

13.10.1 erlang:disconnect_node/1

Friendship is terminated with a given node.

Type

`disconnect_node(term()) -> atom().`

Exits

`disconnect_node/1` always completes normally.

Evaluation

Let P be the process calling `disconnect_node/1`.

- If `node[`P`]` is not communicating, then `ignored` is returned.
- Otherwise, if v_1 is not an atom or v_1 is not in `friends[node[`P`]]`, `false` is returned.
- Otherwise the same things happen as when node `node[`P`]` has lost contact with node v_1, as described in §11.3 and §11.6, and then `true` is returned.

13.10.2 erlang:get_cookie/0

The magic cookie of the current node is returned.

Type

`get_cookie() -> atom().`

Exits

`get_cookie/0` always completes normally.

Evaluation

The BIF returns `magic_cookie[node[`P`]]`, where P is the process calling `get_cookie/0`.

13.10.3 `erlang:halt/0`

The current node is terminated.

Type

`halt() -> _.`

Exits

`halt/0` never completes abnormally.

Evaluation

The node `node[P]`, where `P` is the process calling `halt/0`, is terminated immediately. The BIF thus never returns.

13.10.4 `is_alive/0`

The BIF tests whether the current node is communicating or not.

Type

`is_alive() -> bool().`

Exits

`is_alive/0` always completes normally.

Evaluation

The BIF returns the value of `communicating[node[P]]`, where `P` is the process calling the BIF.

13.10.5 `monitor_node/2`

Calling this BIF has as effect to increase or decrease the number of messages that the current process will receive as notification that friendship between the current node and a certain (other) node has ceased.

Type

`monitor_node(atom(),bool()) -> true.`

Exits

`monitor_node/2` exits with cause `badarg` if v_1 is not an atom or if v_2 is not a Boolean atom. On an isolated node, it will also exit with cause `badarg` if the node to be monitored is not the current node.

Evaluation

Let P be the process calling `monitor_node/2`.

- If `alive[node[`P`]]` is `false` and v_1 is not `node[`P`]`, then the BIF exits with cause `badarg`.
- If v_1 is `node[`P`]`, do nothing.
- If v_1 is `true`, do the following:
  1. If there is not already a row in `monitored_nodes[node[`P`]]` with v_1 as key, then add one with an empty table as value.
  2. Let t be the value for v_1 in `monitored_nodes[node[`P`]]`. If there is no row with P as key in that table, add such a row with value 1. Otherwise, add 1 to the value for P in t.
- Otherwise (v_1 is `false`):
  - ∗ If there is no row with v_1 as key in `monitored_nodes[node[`P`]]`, then do nothing.
  - ∗ Otherwise, let t be the value for v_1. If there is no entry for P in t, do nothing.
  - ∗ Otherwise, if the row for P in t has value 1, then remove the row for P.
  - ∗ Otherwise, subtract 1 from the value for P in t.

The result is always `true`.

13.10.6 `node/0`

This BIF returns (the name of) the node on which the calling process resides. `node/0` is a guard BIF.

Type

`node() -> atom().`

Exits

`node/0` always completes normally.

Evaluation

The result is `node[`P`]`, where P is the process calling `node/0` (§10.9.1).

13.10.7 `node/1`

This BIF returns (the name of) the node on which a given ref, PID or port was created. `node/1` is a guard BIF.

Type

```
node(ref()) -> atom() ;
node(pid()) -> atom() ;
node(port()) -> atom().
```

Exits

`node/1` exits with cause `badarg` if v_1 is not a ref, nor a PID or a port.

Evaluation

The result is `node[`v_1`]` (§10.9.1, §12.10.1).

13.10.8 `nodes/0`

A list is returned of all friends of the node on which the BIF is called.

Type

```
nodes() -> [atom()].
```

Exits

`nodes/0` always completes normally.

Evaluation

The BIF returns a list representing the value of `friends[node[`P`]]`, where P is the process calling the BIF.

13.10.9 `processes/0`

A list is returned of the PIDs of all live processes on the node on which the BIF is called.

Type

```
processes() -> [pid()].
```

Exits

`processes/0` always completes normally.

Evaluation

The BIF returns a list representing the value of `processes[node[`P`]]`, where P is the process calling the BIF.

13.10.10 `erlang:set_cookie/2`

The magic cookie of the current node and/or the presumed magic cookie of another node is set.

Type

```
set_cookie(atom(),atom()) -> bool().
```

Exits

`set_cookie/2` exits with cause `badarg` if v_1 or v_2 is not an atom, or if the atom v_1 could not be the name of a node (i.e., it does not contain exactly one '`@`' character, cf. §11.1).

Evaluation

There are two cases, depending on the value of v_1. Let P be the process calling `set_cookie/2` and let N be `node[`P`]`.

- If v_1 equals N, then
  1. Set `magic_cookie[`N`]` to v_2.
  2. For each pair in `magic_cookies[`N`]` where the presumed magic cookie is `nocookie`, change the presumed magic cookie to v_2.
- If v_1 does not equal N, then delete any pair for v_1 in `magic_cookies[`N`]` and add a pair (v_1, v_2) to `magic_cookies[`N`]`.

13.10.11 `set_node/2`

Calling this BIF has as effect to make the node on which the calling process resides a communicating node, provided that certain preconditions are met.

Type

```
set_node(???,???)  -> ???.
```

Exits

???

Evaluation

???

13.10.12 `erlang:set_node/3`

Ugga mugga.

Type

```
set_node(???,???,???)  -> ???.
```

Exits

`set_node/3` ...

Evaluation

Yumm yumm!

13.10.13 `statistics/1`

Information about the current state of the node is returned.

Type

```
statistics(garbage_collection) -> {int(),int(),int()}
statistics(reductions) -> {int(),int()}
statistics(runtime) -> {int(),int()}
statistics(run_queue) -> int()
statistics(wall_clock) -> {int(),int()}.
```

Exits

`statistics/1` exits with cause `badarg` if v_1 is not an atom or if it is not one of the atoms `runtime`, `wall_clock` or `reductions`.

Evaluation

Let *`N`* be the node on which the BIF is called. The evaluation depends on v_1:

- `garbage_collection`: return the result of `current_gc[`*`N`*`]` (§11.7.2), i.e., a 3-tuple `{`*`NumberOfGCs`*`,`*`WordsReclaimed`*`,0}` of integers where *`NumberOfGCs`* is the total number of garbage collection operations that have been carried out by processes on the node and *`WordsReclaimed`* is the total number of memory words reclaimed by such operations, and the third integer is always 0.

- `reductions`: return the result of `current_reductions[`*`N`*`]` (§11.7.2), i.e., a 2-tuple `{`*`TotalReductions`*`,`*`RecentReductions`*`}` where *`TotalReductions`* is an Erlang integer representing the number of function calls made on the node and *RecentReductions* is the same but only since the last call `statistics(reductions)`.

- `run_queue`: return $\Re[i]$, where i is the number of processes on the node that have `runnable` or `running` status.

- `runtime`: return the result of `current_runtime[`*`N`*`]` (§11.7.2), i.e., a 2-tuple `{`*`TotalRuntime`*`,`*`RecentRuntime`*`}` where *`TotalRuntime`* is an Erlang integer representing the total time spent running processes on the node and *RecentRunTime* is the same but only since the last call `statistics(runtime)`.

- `wall_clock`: return the result of `current_wall_clock[`*`N`*`]` (§11.7.2), i.e., a 2-tuple `{`*`TotalWallClock`*`,`*`RecentWallClock`*`}` where *`TotalWallClock`* is an Erlang integer representing the time which has passed since the node was started and *RecentWallClock* is the same but only since the last call `statistics(wall_clock)`.

13.11 Builtin functions for process registries

The process registry of a node is described in §11.5.

13.11.1 `register/2`

A name is registered for a process on a node.

Type

`register(atom(),pid()) -> true`.

Exits

`register/2` exits with cause `badarg` if v_1 is not an atom or if v_2 is not a PID. `register/2` may also exit with cause `badarg` if some process is already registered under the name, there is already a name registered for the process, the process resides on a different node than the one on which the BIF is called, or the process has completed (see below).

Evaluation

Let *`N`* be the node on which the BIF is called.

- If `node[`v_2`]` is not N or `registry[`N`]` already contains a process with name v_1 or `registry[`N`]` already contains a name for process v_2, or process v_2 has completed, then the BIF exits with cause `badarg`.
- Otherwise, the name v_1 is added for v_2 in `registry[`N`]`. The BIF always returns the atom `true`.

The operation should be atomic to ensure the integrity of the registry.

13.11.2 `registered/0`

A list of all names in the registry on the current node is returned.

Type

`registered() -> [atom()].`

Exits

`registered/0` always completes normally.

Evaluation

Let N be the node on which the BIF is called. A list without duplicates of all names in `registry[`N`]` is returned. The order of the atoms in the list is not defined. The operation should be atomic to ensure the integrity of the registry.

13.11.3 `unregister/1`

Any registration for a name is removed.

Type

`unregister(atom()) -> true.`

Exits

`unregister/1` exits with cause `badarg` if v_1 is not an atom.

Evaluation

Let N be the node on which the BIF is called.

- If `registry[`N`]` does not contains the name v_1, then the BIF has no effect.
- Otherwise, the association for the name v_1 is removed from `registry[`N`]`.

In either case, the atom `true` is returned. The operation should be atomic to ensure the integrity of the registry.

13.11.4 `whereis/1`

The PID of a process registered under the given name is returned, if any.

Type

`whereis(atom()) -> term().`

Exits

`whereis/1` exits with cause `badarg` if v_1 is not an atom.

Evaluation

Let *N* be the node on which the BIF is called.

- If `registry[`*N*`]` does not contains the name v_1, then the BIF returns the atom `undefined`.
- Otherwise, the PID that is the value for v_1 in `registry[`*N*`]` is returned.

The operation should be atomic to ensure the integrity of the registry.

13.12 Builtin functions for I/O and ports

13.12.1 `open_port/2`

A new port is opened to a recently opened driver or recently spawned external process.

Type

`open_port(term(),[term()]) -> port().`

Exits

`open_port/2` exits with cause `badarg` if v_1 is not one of the permitted alternatives or v_2 contains an invalid option (§12.4).

Evaluation

The BIF `open_port/2` is described fully in §12.4.

13.12.2 `port_close/1`

A port is closed.

Type

`port_close(port()]) -> true`.

Exits

`port_close/1` exits with cause `badarg` if v_1 is not a port. It may also exit with cause `badarg` if v_1 is already closed.

Evaluation

If v_1 is already closed, exit with cause `badarg`. Otherwise, close port v_1.

13.12.3 `port_info/1`

Information about various properties of a port is returned.

Type

`port_info(port()) -> term()`.

Exits

`port_info/1` exits with cause `badarg` if v_1 is not a port.

Evaluation

The BIF returns information about the port v_1.

- If that port is not open, the result is the atom `undefined`.

- Otherwise, the result is a list of 2-tuples, each of which is the same as the result of an application of the BIF `port_info/2` (§13.12.4) to v_1 and a distinct atom in the left column of Table 13.3. The list should include all properties for which `port_info/2` gives meaningful information.

13.12.4 `port_info/2`

Information about some property of a port is returned, as described in Table 13.3.

Type

`port_info(pid(),atom()) -> term()`.

Exits

`port_info/2` exits with cause `badarg` if v_1 is not a port or v_2 is not one of the atoms in the left column of Table 13.3.

Evaluation

- If port v_1 is closed, the result is the atom `undefined`.
- Otherwise, the BIF returns information about the port v_1 and the result is always a 2-tuple where the first element is v_2:
  - ∗ If v_2 is `id`, then return `{id,ID[`v_1`]}`.
  - ∗ If v_2 is `connected`, then return `{connected,owner[`v_1`]}`.
  - ∗ If v_2 is `input`, then return `{input,count_in[`v_1`]}`.
  - ∗ If v_2 is `links`, then return `{links,`*Lst*`}`, where *Lst* is a list representing the value of `linked[`v_1`]`.
  - ∗ If v_2 is `name`, then return `{name,command[`v_1`]}`.
  - ∗ If v_2 is `output`, then return `{output,count_out[`v_1`]}`.

Let *P* be the process calling the BIF. The behaviour with respect to signals (§10.6) should be as if the result was obtained by *P* sending an *info request* signal to port v_1 with v_2 as additional information, and port v_1 responding with a message to *P* containing the result.

13.12.5 `ports/0`

A list is returned of all open ports on the node on which the BIF is called.

Type

```
ports() -> [port()].
```

Second argument	Information returned about the port
`id`	The ID of the port.
`connected`	The process owning the port.
`input`	The number of bytes read from the port.
`links`	The processes to which the port is linked.
`name`	The driver or external process to which the port is opened.
`output`	The number of bytes written to the port.

Table 13.3: Alternatives for the BIF `port_info/2`.

Exits

`ports/0` always completes normally.

Evaluation

The BIF returns a list representing the value of `ports[node[`*`P`*`]]`, where *`P`* is the process calling the BIF.

13.13 Miscellaneous builtin functions

13.13.1 `date/0`

The local date when the BIF is called is returned as a triple.

Type

`date() -> {int(),int(),int()}`.

Exits

`date/0` always completes normally.

Evaluation

Let the current local year, month and day at the time of evaluation be y, m and d. Let *Month* be a function mapping January to 1, February to 2, ... , December to 12.

A triple `{`*`Year`*`,`*`Month`*`,`*`Day`*`}` is returned, where *`Year`* is the ERLANG integer $\Re^{-1}[y]$, *`Month`* is the ERLANG integer $\Re^{-1}[\mathit{Month}(m)]$, and *`Day`* is the ERLANG integer $\Re^{-1}[d]$.

Examples

- `date()` evaluated on the 29th of June, 1996, would return `{1996,6,29}`.
- `date()` evaluated on the 1st of January, 2000, would return `{2000,1,1}`.

13.13.2 `erlang:hash/2`

A hash value in a specified range for an arbitrary term is returned.

Type

`hash(term(),fixnum()) -> int()`.

Exits

`hash/2` exits with cause `badarg` if v_2 is not a nonnegative integer.

Evaluation

The BIF returns $\Re[\mathit{Hash}(v_1, \Re^{-1}[v_2]) + 1]$, where the function *Hash* is as defined in §C. That is, the BIF maps each ERLANG term to an integer in the range $[1, v_2]$. The function *Hash* is defined in such a way as to be portable across nodes and independent of time. (It is obviously not invertible.)

13.13.3 `make_ref/0`

A ref is returned that is different from all refs created previously on the node and that is different from all refs created on other nodes.

Type

```
make_ref() -> ref().
```

Exits

`make_ref/0` always completes normally.

Evaluation

Let N be the node on which `make_ref/0` is called. The BIF invokes the operation `next_ref[`N`]` (§11.7.2) and the result is a new ref that is returned. This also modifies the state `ref_state[`N`]` so future invocations of `next_ref[`N`]` will produce different refs.

13.13.4 `now/0`

A 3-tuple of integers is returned that is guaranteed to be different for each invocation on a node.

Type

```
now() -> {int(),int(),int()}.
```

Exits

`now/0` always completes normally.

Evaluation

Two calls of the BIF `now()` by processes residing on the same node can never return the same term.[4]

13.13.5 `throw/1`

A value is thrown.

Type

`throw(term()) -> _`.

Exits

`throw/1` always completes abruptly with reason `{'THROW',`v_1`}`.

13.13.6 `time/0`

The local time of day when the BIF is called is returned as a triple.

Type

`time() -> {int(),int(),int()}`.

Exits

`time/0` always completes normally.

Evaluation

Let the current local hour, minute and second at the time of evaluation be h, m and s. The hour is on 24-hour format.

A triple `{`*`Hour`*`,`*`Minute`*`,`*`Second`*`}` is returned, where *`Hour`* is the ERLANG integer $\Re^{-1}[h]$, *`Minute`* is the ERLANG integer $\Re^{-1}[m]$, and *`Second`* is the ERLANG integer $\Re^{-1}[s]$.

Examples

- `time()` evaluated at five minutes and forty-two seconds past midnight would return `{0,5,42}`.

- `time()` evaluated at five minutes and forty-two seconds past noon would return `{12,5,42}`.

[4]The name of the BIF comes from the fact that the three integers normally represent the universal time (with microsecond resolution). However, the time might be inaccurate if several calls are made within a microsecond.

- `time()` evaluated at five minutes and forty-two seconds before midnight would return `{23,44,18}`.

13.14 Reserved function names

The function names in Table 13.4 do not name BIFs but are recognized by the compiler and a module must not define any function with one of these names.

The reason may be that the compiler automatically generates a definition of a function with such a name (e.g., `module_info/0` and `module_info/1`), or that applications of function named as such are treated specially (e.g., `record_info/2`).

Function names without a reference in Table 13.4 are reserved because ERLANG 4.7.3 uses them for internal purposes. If a module defines a function with one of the names below, the compiler gives a compile-time error.

Function name	Described
`apply_lambda/2`	
`module_info/0`	§8.5.1
`module_info/1`	§8.5.2
`module_lambdas/4`	
`record/2`	§6.20.1
`record_index/2`	
`record_info/2`	

Table 13.4: Reserved function names

Chapter 14

Libraries

There are some standard libraries that belong to ERLANG 4.7.3. Each library is one ERLANG module:

Further standard libraries are provided with OTP [6].

14.1 The `file` library

TO BE WRITTEN!

14.2 The `io` library

TO BE WRITTEN!

14.3 The `lists` library

TO BE WRITTEN!

14.4 The `math` library

The library contains trigonometric and logarithmic functions. All trigonometric functions work with angles in radians. The functions correspond to

Module name	Contents
`file`	File handling
`io`	Simple input and output
`lists`	List functions
`math`	Logarithmic and arithmetic functions
`string`	String functions

Table 14.1: Standard libraries of ERLANG 4.7.3

the functions in the `math` facilities of ISO C [10, 13].

In the description of a function, v_i refers to the value of the ith argument.

14.4.1 `acos/1`

Computes the *arccosine* function.

Type

`acos(num()) -> float().`

Exits

`acos/1` exits with cause `badarg` if v_1 is not a number. `acos/1` exits with cause `badarith` if v_1 is not in the range $[-1, 1]$.

Evaluation

The function returns $result_F(\arccos(v_1), rnd_F)$. The result is always in the range $[0, \pi]$.

14.4.2 `asin/1`

Computes the *arcsine* function.

Type

`asin(num()) -> float().`

Exits

`asin/1` exits with cause `badarg` if v_1 is not a number. `asin/1` exits with cause `badarith` if v_1 is not in the range $[-1, 1]$.

Evaluation

The function returns $result_F(\arcsin(v_1), rnd_F)$. The result is always in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

14.4.3 `atan/1`

Computes the *arctangent* function.

Type

`atan(num()) -> float().`

Exits

`atan/1` exits with cause `badarg` if v_1 is not a number.

Evaluation

The function returns $result_F(\arctan(v_1), rnd_F)$. The result is always in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

14.4.4 `atan2/2`

Computes the *arctangent* function of a quotient, taking the signs of the arguments into account for determining the quadrant.

Type

`atan2(num(), num()) -> float().`

Exits

`atan2/2` exits with cause `badarg` if v_1 or v_2 is not a number. `atan2/2` may exit with cause `badarith` if v_1 and v_2 are both zero.

Evaluation

The function returns $result_F(\arctan(\frac{v_1}{v_2}), rnd_F)$. The result is always in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

- If v_2 is zero, then the result is $\pi/2$ if v_1 is positive and $-\pi/2$ if v_1 is negative.
- If v_1 and v_2 are both positive, `atan2(`v_1`,`v_2`)` equals `atan(`v_1`/`v_2`)`.
- If v_1 is negative and v_2 is positive, `atan2(`v_1`,`v_2`)` equals `-atan((-`v_1`)/`v_2`)`.
- If v_1 is positive and v_2 is negative, `atan2(`v_1`,`v_2`)` equals `math:pi()-atan(`v_1`/(-`v_2`))`.
- If v_1 and v_2 are both negative, `atan2(`v_1`,`v_2`)` equals `atan(`v_1`/`v_2`)-math:pi()`.

14.4.5 `cos/1`

Computes the *cosine* function.

Type

`cos(num()) -> float().`

Exits

`cos/1` exits with cause `badarg` if v_1 is not a number.

Evaluation

The function returns $\mathit{result}_F(\cos(\mathtt{v}_1), \mathit{rnd}_F)$.

14.4.6 `cosh/1`

Computes the *hyperbolic cosine* function.

Type

`cosh(num()) -> float().`

Exits

`cosh/1` exits with cause `badarg` if $\mathtt{v}_1$ is not a number.

Evaluation

The function returns $\mathit{result}_F(\cosh(\mathtt{v}_1), \mathit{rnd}_F)$.

14.4.7 `exp/1`

Computes exponentiation with base e.

Type

`exp(num()) -> float().`

Exits

`exp/1` exits with cause `badarg` if $\mathtt{v}_1$ is not a number.

Evaluation

The function returns $\mathit{result}_F(e^{\mathtt{v}_1}, \mathit{rnd}_F)$.

14.4.8 `log/1`

Computes logarithm with base e.

Type

`log(num()) -> float().`

Exits

`log/1` exits with cause `badarg` if $\mathtt{v}_1$ is not a number. `log/1` exits with cause `badarith` if $\mathtt{v}_1 \leq 0$.

Evaluation

The function returns $\mathit{result}_F(\log_e(v_1), \mathit{rnd}_F)$.

14.4.9 `log10/1`

Computes logarithm with base 10.

Type

`log10(num()) -> float().`

Exits

`log10/1` exits with cause `badarg` if v_1 is not a number. `log10/1` exits with cause `badarith` if $v_1 \leq 0$.

Evaluation

The function returns $\mathit{result}_F(\log_{10}(v_1), \mathit{rnd}_F)$.

14.4.10 `pi/0`

Returns an approximation of π.

Type

`pi() -> float().`

Exits

`pi/0` always completes normally.

Evaluation

$\mathit{result}_F(\pi, \mathit{rnd}_F)$ is returned.

14.4.11 `pow/2`

Computes exponentiation.

Type

`pow(num(),num()) -> float().`

Exits

`pow/2` exits with cause `badarg` if v_1 or v_2 is not a number. `pow/2` exits with cause `badarith` if $v_1 < 0$ and v_2 is a float for which the fraction part is not zero.

Evaluation

The function returns $v_1^{v_2}, rnd_F)$.

14.4.12 `sin/1`

Computes the *sine* function.

Type

`sin(num()) -> float()`.

Exits

`sin/1` exits with cause `badarg` if v_1 is not a number.

Evaluation

The function returns $result_F(\sin(v_1), rnd_F)$.

14.4.13 `sinh/1`

Computes the *hyperbolic sine* function.

Type

`sinh(num()) -> float()`.

Exits

`sinh/1` exits with cause `badarg` if v_1 is not a number.

Evaluation

The function returns $result_F(\sinh(v_1), rnd_F)$.

14.4.14 `sqrt/1`

Computes square roots.

Type

`sqrt(num()) -> float()`.

Exits

`sqrt/1` exits with cause `badarg` if v_1 is not a number. `sqrt/1` exits with cause `badarith` if $v_1 < 0$.

Evaluation

The function returns $\sqrt{v_1}, rnd_F)$.

14.4.15 `tan/1`

Computes the *tangent* function.

Type

`tan(num()) -> float().`

Exits

`tan/1` exits with cause `badarg` if v_1 is not a number and may exit with `badarith` if $cos(v_1)$ is zero.

Evaluation

The function returns $result_F(\tan(v_1), rnd_F)$.

14.4.16 `tanh/1`

Computes the *hyperbolic tangent* function.

Type

`tanh(num()) -> float().`

Exits

`tanh/1` exits with cause `badarg` if v_1 is not a number.

Evaluation

The function returns $result_F(\tanh(v_1), rnd_F)$.

14.5 The `string` library

TO BE WRITTEN!

Bibliography

[1] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science.* Computer Science Press, New York, 1992.

[2] American National Standards Institute, New York. *Coded character set — 7-bit American National Standard Code for Information Interchange*, 1986. ANSI X3.4 – 1986.

[3] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG.* Prentice Hall, Hemel Hempstead, second edition, 1996.

[4] Jonas Barklund and Robert Virding. ERLANG *4.7.3 Reference Manual.* Ericsson Software Technology AB, Box 1214, S-164 28 Kista, Sweden, 1999.

[5] The Unicode Consortium. *The Unicode Standard, Version 2.0.* Addison-Wesley, Reading, Mass., 1996.

[6] Ericsson Software Technology AB, Box 1214, S-164 28 Kista, Sweden. *Erlang System/OTP 4.5: Development Environment Reference Manual*, second edition, 1997.

[7] Ericsson Software Technology AB, Box 1214, S-164 28 Kista, Sweden. *Erlang System/OTP 4.5: Mnesia Database Management System (Mnesia 1.1)*, second edition, 1997.

[8] D. Goldsmith and M. Davis. Utf-7: A mail-safe transformation format of unicode. Available by anonymous ftp from `ftp://ds.internic.net/rfc/rfc1642.txt`, July 1994.

[9] James Gosling, Bill Joy, and Guy L. Steele Jr. *The Java*$^{\text{TM}}$ *Language Specification.* Addison-Wesley, Reading, Mass., 1996.

[10] Samuel P. Harbison and Guy L. Steele Jr. *C, a Reference Manual.* Prentice-Hall, Englewood Cliffs, N. J., 1995.

[11] IEEE. *Dead rats and rotten mice*, 1066.

[12] ISO/IEC. *Information processing — 8-bit single-byte coded graphic character sets*, 1987. Reference number ISO 8879:1987.

[13] ISO/IEC. *Information technology — Programming languages — C*, 1990. Reference number ISO/IEC 9899:1990.

[14] ISO/IEC. *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic*, first edition, 1994. Reference number ISO/IEC 10967-1:1994(E).

[15] ISO/IEC. *Information technology — Language independent arithmetic — Part 2: Elementary numerical functions. WORKING DRAFT*, first edition, 1995. Reference number ISO/IEC WD 10967-1:1995(E).

[16] ISO/IEC. *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane. Amendment 2: UCS Transformation Format 8 (UTF-8)*, 1996. Reference number ISO/IEC 10646-1: 1993/AMD.2: 1996(E).

[17] Simon Marlow and Philip Wadler. Erltc: A type checker for Erlang. Technical report, University of Glasgow, 1996.

[18] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In Mads Tofte, editor, *1997 ACM SIGPLAN Intl. Conf. on Functional Programming*, New York, N.Y., 1997. ACM.

[19] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML — Revised.* MIT Press, Cambridge, Mass., 1997.

[20] *Revised^5 Report on the Algorithmic Language Scheme*, February 1998.

Appendix A

Summary of Erlang expressions

This is a summary of the expressions of ERLANG. Each group of expressions is annotated with a precedence.

- α^+ means a nonempty comma-separated sequence of α.
- $\alpha^@$ means a nonempty semicolon-separated sequence of α.
- $\alpha \mid \beta$ means either α or β.
- $[\alpha]$ means α or nothing.
- (α) means α.
- E_p means an expression with precedence p or less.
- *A* means an atom.
- *A* means an integer literal.
- *L* means an atomic literal.
- *V* means a variable.
- *P* means a pattern.
- *G* means a guard.

10	catch E_{12}				§6.9
9	P = E_{11}				§6.10
8	E_9 = E_{10}				§6.11
7	E_6 < E_6	E_6 =< E_6	E_6 > E_6	E_6 >= E_6	§6.12
	E_6 =:= E_6	E_6 =/= E_6	E_6 == E_6	E_6 /= E_6	§6.12
6	E_5 ++ E_6	E_5 -- E_6			§6.13
5	E_5 + E_4	E_5 - E_4	E_5 bor E_4	E_5 bxor E_4	§6.14
	E_5 bsl E_4	E_5 bsr E_4			§6.14
	E_9 or E_8	E_9 xor E_8			§6.14
4	E_4 * E_3	E_4 / E_3			§6.15
	E_4 div E_3	E_4 rem E_3			§6.15
	E_4 band E_3				§6.15
	E_8 and E_7				§6.15
3	+ E_2	- E_2	bnot E_2	not E_2	§6.16
2	#A.A	#A{(A=E_{12})$^+$}			§6.17
	E_2#A.A	E_2#A{(A=E_{12})$^+$}			§6.17
1	A(E_{12}^+)	E_0(E_{12}^+)	E_0:E_0(E_{12}^+)		§6.18
0	V				§6.19.1
	L				§6.19.2
	{E_{12}^+}				§6.19.3
	[]	[E_{12}^*|E_{12}]			§6.19.4
	[E_{12}||(P<-E_{12} | E_{12})$^+$]				§6.19.5
	begin E_{12}^+ end				§6.19.6
	if (G -> E_{12}^+)$^@$ end				§6.19.7
	case E_{12} of (P [when G] -> E_{12}^+)$^@$ end				§6.19.8
	receive (P [when G] -> E_{12}^+)$^@$ [after E_{12} -> E_{12}^+] end				§6.19.9
	fun A/I				§6.19.10
	fun ((P^+) [when G] -> E_{12}^+)$^@$ end				§6.19.10
	query [E_{12}||(P<-E_{12} | E_{12})$^+$] end				§6.19.11
	(E_{12})				§6.19.12

Appendix B

Parse trees

In this appendix we define the standard representation of parse trees for ERLANG programs as ERLANG terms. A `parse_transform/1` (§7.6) takes a list of such terms as input and is expected to return a list of such terms. We define the representation in a top-down way by first examining the forms that make up a module declaration and then going through their lexical constituents, etc.

The representation admits representation of many parse trees that would be rejected by the grammar of ERLANG. For example, it allows representation of a module declaration in which forms are not in a permitted order.

We define the representation semi-formally through a function *Rep* that maps ERLANG formulas or sequences of formulas to ERLANG terms.

When we write **L**, we mean an ERLANG integer that may be interpreted by software tools as a line number when reporting errors, etc.

B.1 Module declarations and forms

A module declaration consists of a sequence of forms that are either function declarations or attributes (§8.1).

- If D is a module declaration consisting of the forms $F_1, \ldots, F_k$, then

$$Rep(D) = \texttt{[}Rep(F_1)\texttt{,}\ldots\texttt{,}Rep(F_k)\texttt{]}.$$

- If F is an attribute `-module(`*Mod*`)`, then

$$Rep(F) = \texttt{\{attribute,}\mathbf{L}\texttt{,module,}Mod\texttt{\}}.$$

- If F is an attribute `-export([`Fun_1`/`A_1`,...,`Fun_k`/`A_k`])`, then

$$Rep(F) = \texttt{\{attribute,}\mathbf{L}\texttt{,export,[\{}Fun_1\texttt{,}A_1\texttt{\},...,\{}Fun_k\texttt{,}A_k\texttt{\}]\}}.$$

- If *F* is an attribute `-import(`*Mod*`,[`Fun_1`/`A_1`,...,`Fun_k`/`A_k`])`, then

$$Rep(F) = \texttt{\{attribute,}\mathbf{L}\texttt{,import,\{}Mod\texttt{,[\{}Fun_1\texttt{,}A_1\texttt{\},...,}\ \texttt{\{}Fun_k\texttt{,}A_k\texttt{\}]\}\}}.$$

- If *F* is an attribute `-compile([`T_1`,...,`T_k`])`, then

$$Rep(F) = \texttt{\{attribute,}\mathbf{L}\texttt{,compile,[}T_1\texttt{,...,}T_k\texttt{]\}}.$$

- If *F* is an attribute `-file(`*File*`,`*Line*`)`, then

$$Rep(F) = \texttt{\{attribute,}\mathbf{L}\texttt{,file,\{}Rep(File)\texttt{,}Rep(Line)\texttt{\}\}}.$$

- If *F* is a record declaration `-record(`*Name*`,{`V_1`,...,`V_k`})`, then

$$Rep(F) = \texttt{\{attribute,}\mathbf{L}\texttt{,record,\{}Name\texttt{,[}Rep(V_1)\texttt{,...,}\ Rep(V_k)\texttt{]\}\}}.$$

- If *F* is a wild attribute `-`*A*`(`*T*`)`, then

$$Rep(F) = \texttt{\{attribute,}\mathbf{L}\texttt{,}A\texttt{,}T\texttt{\}}.$$

- If *F* is a function declaration *Name*`(`Ps_1`)` [`when` Gs_1] `->` B_1 `; ...;` *Name*`(`Ps_k`)` [`when` Gs_k] `->` B_k `end`, where $k \geq 1$ and each Ps_i, Gs_i and B_i, $1 \leq i \leq k$, is a pattern sequence, a guard and a body, respectively (and Gs_i is `true` if omitted), and each Ps_i, $1 \leq i \leq k$, has the same length *Arity*, then

$$\begin{aligned}Rep(F) = \texttt{\{function,}&\mathbf{L}\texttt{,}Name\texttt{,}Arity\texttt{,}\\ &\texttt{[\{clause,}\mathbf{L}\texttt{,}Rep(Ps_1)\texttt{,}Rep(Gs_1)\texttt{,}Rep(B_1)\texttt{\},...,}\\ &\ \texttt{\{clause,}\mathbf{L}\texttt{,}Rep(Ps_k)\texttt{,}Rep(Gs_k)\texttt{,}Rep(B_k)\texttt{\}]\}}.\end{aligned}$$

In addition to the representations of forms, the list that represents a module declaration may contain tuples `{error,`*E*`}`, denoting syntactically incorrect forms, and `{eof,`**L**`}`, denoting an end of stream encountered before a complete form had been parsed.

Each field declaration in a record declaration is with or without an explicit default initializer expression (§8.4).

- If *V* is *A*, then

$$Rep(V) = \texttt{\{record\_field,}\mathbf{L}\texttt{,}Rep(A)\texttt{\}}.$$

- If *V* is *A* `=` *E*, then

$$Rep(V) = \texttt{\{record\_field,}\mathbf{L}\texttt{,}Rep(A)\texttt{,}Rep(E)\texttt{\}}.$$

B.2 Atomic literals

There are five kinds of atomic literals, which are represented in the same way in patterns, expressions and guard expressions:

- If L is an integer literal, then
$$Rep(L) = \texttt{\{integer,}\mathbf{L}\texttt{,}L\texttt{\}}.$$
- If L is a float literal, then
$$Rep(L) = \texttt{\{float,}\mathbf{L}\texttt{,}L\texttt{\}}.$$
- If L is a character literal, then
$$Rep(L) = \texttt{\{integer,}\mathbf{L}\texttt{,}L\texttt{\}}.$$
- If L is a string literal consisting of the characters $C_1, \ldots, C_k$, then
$$Rep(L) = \texttt{\{string,}\mathbf{L}\texttt{,[}C_1\texttt{,}\ldots\texttt{,}C_k\texttt{]\}}.$$
- If L is an atom literal, then
$$Rep(L) = \texttt{\{atom,}\mathbf{L}\texttt{,}L\texttt{\}}.$$

B.3 Patterns

For a sequence Ps of patterns $P_1, \ldots, P_k$,
$$Rep(Ps) = \texttt{[}Rep(P_1)\texttt{,}\ldots\texttt{,}Rep(P_k)\texttt{]}.$$

Individual patterns are represented as follows:

- If P is an atomic literal L, then
$$Rep(P) = Rep(L),$$
cf. §B.2.
- If P is a variable pattern V, then
$$Rep(P) = \texttt{\{var,}\mathbf{L}\texttt{,}A\texttt{\}},$$
where A is an atom with a printname that constitutes the same characters as V.
- If P is a universal pattern _, then
$$Rep(P) = \texttt{\{var,}\mathbf{L}\texttt{,'\_'\}}.$$

- If P is a tuple pattern `{`P_1`,...,`P_k`}`, then

$$Rep(P) = \texttt{\{tuple,}\mathbf{L}\texttt{,[}Rep(P_1)\texttt{,...,}Rep(P_k)\texttt{]\}}.$$

- If P is a nil pattern `[]`, then

$$Rep(P) = \texttt{\{nil,}\mathbf{L}\texttt{\}}.$$

- If P is a cons pattern `[`P_h`|`P_t`]`, then

$$Rep(P) = \texttt{\{cons,}\mathbf{L}\texttt{,}Rep(P_h)\texttt{,}Rep(P_t)\texttt{\}}.$$

- If P is a record pattern `#`*Name*`{`*Field*$_1$`=`P_1`,...,`*Field*$_k$`=`P_k`}`, then

$$\begin{aligned}Rep(P) = \texttt{\{record,}&\mathbf{L}\texttt{,}Name\texttt{,}\\ &\texttt{[\{record\_field,}\mathbf{L}\texttt{,}Rep(Field_1)\texttt{,}Rep(P_1)\texttt{\},...,}\\ &\ \texttt{\{record\_field,}\mathbf{L}\texttt{,}Rep(Field_k)\texttt{,}Rep(P_k)\texttt{\}]\}}.\end{aligned}$$

B.4 Expressions

A body B is a sequence of expressions E_1`,` ... `,` E_k, where $k \geq 1$, and

$$Rep(B) = \texttt{[}Rep(E_1)\texttt{,...,}Rep(E_k)\texttt{]}.$$

An expression E is one of the following alternatives:

- If E is `catch` E_0, then

$$Rep(E) = \texttt{\{'catch',}\mathbf{L}\texttt{,}Rep(E_0)\texttt{\}}.$$

- If E is P `=` E_0, then

$$Rep(E) = \texttt{\{match,}\mathbf{L}\texttt{,}Rep(P)\texttt{,}Rep(E_0)\texttt{\}}.$$

- If E is E_1 *Op* E_2, where *Op* is `!`, `or`, `and`, a *RelationalOp*, an *EqualityOp*, a *ListConcOp*, an *AdditionOp*, a *ShiftOp* or a *MultiplicationOp*, then

$$Rep(E) = \texttt{\{op,}\mathbf{L}\texttt{,}Op\texttt{,}Rep(E_1)\texttt{,}Rep(E_2)\texttt{\}}.$$

- If E is *Op* E_0, where *Op* is a *PrefixOp*, then

$$Rep(E) = \texttt{\{op,}\mathbf{L}\texttt{,}Op\texttt{,}Rep(E_0)\texttt{\}}.$$

- If E is `#`*Name*`.`*Field*, then

$$Rep(E) = \texttt{\{record\_index,}\mathbf{L}\texttt{,}Name\texttt{,}Rep(Field)\texttt{\}}.$$

- If $\mathtt{E}$ is $\mathtt{E}_0$`#`*`Name`*`.`*`Field`*, then

$$Rep(\mathtt{E}) = \texttt{\{record\_field,}\mathbf{L}\texttt{,}Rep(\mathtt{E}_0)\texttt{,}\mathit{Name}\texttt{,}Rep(\mathit{Field})\texttt{\}}.$$

- If $\mathtt{E}$ is `#`*`Name`*`{`Field_1`=`$\mathtt{E}_1$`,...,`Field_k`=`$\mathtt{E}_k$`}`, then

$$\begin{aligned} Rep(\mathtt{E}) = \texttt{\{record,}\mathbf{L}\texttt{,}\mathit{Name}\texttt{,} \\ \texttt{[\{record\_field,}\mathbf{L}\texttt{,}Rep(\mathit{Field}_1)\texttt{,}Rep(\mathtt{E}_1)\texttt{\},...,} \\ \texttt{\{record\_field,}\mathbf{L}\texttt{,}Rep(\mathit{Field}_k)\texttt{,}Rep(\mathtt{E}_k)\texttt{\}]\}}. \end{aligned}$$

- If $\mathtt{E}$ is $\mathtt{E}_0$`#`*`Name`*`{`Field_1`=`$\mathtt{E}_1$`,...,`Field_k`=`$\mathtt{E}_k$`}`, then

$$\begin{aligned} Rep(\mathtt{E}) = \texttt{\{record,}\mathbf{L}\texttt{,}Rep(\mathtt{E}_0)\texttt{,}\mathit{Name}\texttt{,} \\ \texttt{[\{record\_field,}\mathbf{L}\texttt{,}Rep(\mathit{Field}_1)\texttt{,}Rep(\mathtt{E}_1)\texttt{\},...,} \\ \texttt{\{record\_field,}\mathbf{L}\texttt{,}Rep(\mathit{Field}_k)\texttt{,}Rep(\mathtt{E}_k)\texttt{\}]\}}. \end{aligned}$$

- If $\mathtt{E}$ is $\mathtt{E}_0$`(`$\mathtt{E}_1$`,...,`$\mathtt{E}_k$`)` (the case where E_0 is an atom is not distinguished), then

$$Rep(\mathtt{E}) = \texttt{\{call,}\mathbf{L}\texttt{,}Rep(\mathtt{E}_0)\texttt{,[}Rep(\mathtt{E}_1)\texttt{,...,}Rep(\mathtt{E}_k)\texttt{]\}}.$$

- If $\mathtt{E}$ is $\mathtt{E}_m$`:`$\mathtt{E}_0$`(`$\mathtt{E}_1$`,...,`$\mathtt{E}_k$`)`, then

$$\begin{aligned} Rep(\mathtt{E}) = \texttt{\{call,}\mathbf{L}\texttt{,\{remote,}\mathbf{L}\texttt{,}Rep(\mathtt{E}_m)\texttt{,}Rep(\mathtt{E}_0)\texttt{\},[}Rep(\mathtt{E}_1)\texttt{,...,} \\ Rep(\mathtt{E}_k)\texttt{]\}}. \end{aligned}$$

- If $\mathtt{E}$ is a variable $\mathtt{V}$, then

$$Rep(\mathtt{E}) = \texttt{\{var,}\mathbf{L}\texttt{,}\mathtt{A}\texttt{\}},$$

  where $\mathtt{A}$ is an atom with a printname that constitutes the same characters as $\mathtt{V}$.

- If $\mathtt{P}$ is an atomic literal $\mathtt{L}$, then

$$Rep(\mathtt{P}) = Rep(\mathtt{L}),$$

  cf. §B.2.

- If $\mathtt{E}$ is a tuple skeleton `{`$\mathtt{E}_1$`,...,`$\mathtt{E}_k$`}`, then

$$Rep(\mathtt{E}) = \texttt{\{tuple,}\mathbf{L}\texttt{,[}Rep(\mathtt{E}_1)\texttt{,...,}Rep(\mathtt{E}_k)\texttt{]\}}.$$

- If $\mathtt{E}$ is `[]`, then

$$Rep(\mathtt{E}) = \texttt{\{nil,}\mathbf{L}\texttt{\}}.$$

- If $\mathtt{E}$ is a cons skeleton $\mathtt{[E_h|E_t]}$, then

$$Rep(\mathtt{E}) = \mathtt{\{cons,}\mathbf{L}\mathtt{,}Rep(\mathtt{E_h})\mathtt{,}Rep(\mathtt{E_t})\mathtt{\}}.$$

- If $\mathtt{E}$ is a list comprehension $\mathtt{[E_0\ ||\ W_1,\ \ldots,\ W_k]}$, where each $\mathtt{W}_i$, $1 \leq i \leq k$, is a generator or a filter, then

$$Rep(\mathtt{E}) = \mathtt{\{lc,}\mathbf{L}\mathtt{,}Rep(\mathtt{E_0})\mathtt{,[}Rep(\mathtt{W_1})\mathtt{,}\ldots\mathtt{,}Rep(\mathtt{W_k})\mathtt{]\}}.$$

- If $\mathtt{E}$ is `begin` $\mathtt{B}$ `end`, where $\mathtt{B}$ is a body, then

$$Rep(\mathtt{E}) = \mathtt{\{block,}\mathbf{L}\mathtt{,}Rep(\mathtt{B})\mathtt{\}}.$$

- If $\mathtt{E}$ is `if` $\mathtt{Gs_1}$ `->` $\mathtt{B_1}$ `; ...;` $\mathtt{Gs_k}$ `->` $\mathtt{B_k}$ `end`, where $k \geq 1$ and each Gs_i and B_i, $1 \leq i \leq k$, is a guard and a body, respectively, then

$$\begin{aligned}Rep(\mathtt{E}) = \mathtt{\{'if',}\mathbf{L}\mathtt{,[\{clause,}\mathbf{L}\mathtt{,[],}Rep(\mathtt{Gs_1})\mathtt{,}Rep(\mathtt{B_1})\mathtt{\},}\ldots\mathtt{,}\\ \mathtt{\{clause,}\mathbf{L}\mathtt{,[],}Rep(\mathtt{Gs_k})\mathtt{,}Rep(\mathtt{B_k})\mathtt{\}]\}}.\end{aligned}$$

- If $\mathtt{E}$ is `case` $\mathtt{E_0}$ `of` $\mathtt{P_1}$ [`when` $\mathtt{Gs_1}$] `->` $\mathtt{B_1}$ `; ...;` $\mathtt{P_k}$ [`when` $\mathtt{Gs_k}$] `->` $\mathtt{B_k}$ `end`, where $\mathtt{E}'$ is an expression, $k \geq 1$ and each P_i, Gs_i and B_i, $1 \leq i \leq k$, is a pattern, a guard and a body, respectively (and Gs_i is `true` if omitted), then

$$\begin{aligned}Rep(\mathtt{E}) = \mathtt{\{'case',}\mathbf{L}\mathtt{,}&Rep(\mathtt{E_0})\mathtt{,}\\ &\mathtt{[\{clause,}\mathbf{L}\mathtt{,[}Rep(\mathtt{P_1})\mathtt{],}Rep(\mathtt{Gs_1})\mathtt{,}Rep(\mathtt{B_1})\mathtt{\},}\ldots\mathtt{,}\\ &\ \mathtt{\{clause,}\mathbf{L}\mathtt{,[}Rep(\mathtt{P_k})\mathtt{],}Rep(\mathtt{Gs_k})\mathtt{,}Rep(\mathtt{B_k})\mathtt{\}]\}}.\end{aligned}$$

- If $\mathtt{E}$ is `receive` $\mathtt{P_1}$ [`when` $\mathtt{Gs_1}$] `->` $\mathtt{B_1}$ `; ...;` $\mathtt{P_k}$ [`when` $\mathtt{Gs_k}$] `->` $\mathtt{B_k}$ `end`, where $k \geq 1$ and each P_i, Gs_i and B_i, $1 \leq i \leq k$, is a pattern, a guard and a body, respectively (and Gs_i is `true` if omitted), then

$$\begin{aligned}Rep(\mathtt{E}) = \mathtt{\{'receive',}\mathbf{L}\mathtt{,}&\\ &\mathtt{[\{clause,}\mathbf{L}\mathtt{,[}Rep(\mathtt{P_1})\mathtt{],}Rep(\mathtt{Gs_1})\mathtt{,}Rep(\mathtt{B_1})\mathtt{\},}\ldots\mathtt{,}\\ &\ \mathtt{\{clause,}\mathbf{L}\mathtt{,[}Rep(\mathtt{P_k})\mathtt{],}Rep(\mathtt{Gs_k})\mathtt{,}Rep(\mathtt{B_k})\mathtt{\}]\}}.\end{aligned}$$

- If $\mathtt{E}$ is `receive` $\mathtt{P_1}$ [`when` $\mathtt{Gs_1}$] `->` $\mathtt{B_1}$ `; ...;` $\mathtt{P_k}$ [`when` $\mathtt{Gs_k}$] `->` $\mathtt{B_k}$ `after` $\mathtt{E}'$ `->` $\mathtt{B_{k+1}}$ `end`, where $k \geq 1$, each P_i, Gs_i and B_i, $1 \leq i \leq k$, is a pattern, a guard and a body, respectively (and Gs_i is `true` if omitted), $\mathtt{E}'$ is an expression and $\mathtt{B}_{k+1}$ is a body, then

$$\begin{aligned}Rep(\mathtt{E}) = \mathtt{\{'receive',}\mathbf{L}\mathtt{,}&\\ &\mathtt{[\{clause,}\mathbf{L}\mathtt{,[}Rep(\mathtt{P_1})\mathtt{],}Rep(\mathtt{Gs_1})\mathtt{,}Rep(\mathtt{B_1})\mathtt{\},}\ldots\mathtt{,}\\ &\ \mathtt{\{clause,}\mathbf{L}\mathtt{,[}Rep(\mathtt{P_k})\mathtt{],}Rep(\mathtt{Gs_k})\mathtt{,}Rep(\mathtt{B_k})\mathtt{\}],}\\ &Rep(\mathtt{E_0})\mathtt{,}Rep(\mathtt{B_{k+1}})\mathtt{\}}.\end{aligned}$$

- If `E` is `fun` *`Name/Arity`*, then

$$Rep(\texttt{E}) = \texttt{\{'fun',}\mathbf{L}\texttt{,\{function,}\mathit{Name}\texttt{,}\mathit{Arity}\texttt{\}\}}.$$

- If `E` is `fun` P_1 [`when` Gs_1] `->` B_1 `; ...;` P_k [`when` Gs_k] `->` B_k `end`, where $k \geq 1$ and each P_i, Gs_i and B_i, $1 \leq i \leq k$, is a pattern, a guard and a body, respectively (and Gs_i is `true` if omitted), then

$$\begin{aligned}Rep(\texttt{E}) = \texttt{\{'fun',}\mathbf{L}\texttt{,\{clauses,}\\ &\texttt{[\{clause,}\mathbf{L}\texttt{,[}Rep(P_1)\texttt{],}Rep(Gs_1)\texttt{,}Rep(B_1)\texttt{\}]\},}\\ &\texttt{...,}\\ &\ \texttt{\{clause,}\mathbf{L}\texttt{,[}Rep(P_k)\texttt{],}Rep(Gs_k)\texttt{,}Rep(B_k)\texttt{\}]\}\}}.\end{aligned}$$

- If `E` is `query [`E_0 `||` W_1`,...,`W_k`] end`, where each W_i, $1 \leq i \leq k$, is a generator or a filter, then

$$Rep(\texttt{E}) = \texttt{\{'query',}\mathbf{L}\texttt{,\{lc,}\mathbf{L}\texttt{,}Rep(E_0)\texttt{,[}Rep(W_1)\texttt{,...,}Rep(W_k)\texttt{]\}\}}.$$

- If `E` is E_0`.`*`Field`*, a Mnesia record access inside a `query`, then

$$Rep(\texttt{E}) = \texttt{\{record\_field,}\mathbf{L}\texttt{,}Rep(E_0)\texttt{,}Rep(\mathit{Field})\texttt{\}}.$$

- If `E` is `(` E' `)`, then

$$Rep(\texttt{E}) = Rep(\texttt{E}'),$$

  i.e., parenthesized expressions cannot be distinguished from their bodies.

When `W` is a generator or a filter (in the body of a list comprehension), then:

- If `W` is a generator `P <- E`, where `P` is a pattern and `E` is an expression, then

$$Rep(\texttt{W}) = \texttt{\{generate,}\mathbf{L}\texttt{,}Rep(\texttt{P})\texttt{,}Rep(\texttt{E})\texttt{\}}.$$

- If `W` is a filter `E`, which is an expression, then

$$Rep(\texttt{W}) = Rep(\texttt{E}).$$

B.5 Guards

A guard $\texttt{Gs}$ is a nonempty sequence of guard tests $G_1, \ldots, G_k$, and

$$Rep(\texttt{Gs}) = \texttt{[}Rep(G_1)\texttt{,}\ldots\texttt{,}Rep(G_k)\texttt{]}.$$

A guard test G is either `true`, an application of a BIF to a sequence of guard expressions (syntactically this includes guard record tests), or a binary operator applied to two guard expressions.

- If G is `true`, then

$$Rep(G) = \texttt{\{atom,}\mathbf{L}\texttt{,true\}}.$$

- If G is an application $A\texttt{(}E_1\texttt{,}\ldots\texttt{,}E_k\texttt{)}$, where A is an atom and $E_1, \ldots, E_k$ are guard expressions, then

$$Rep(G) = \texttt{\{call,}\mathbf{L}\texttt{,\{atom,}\mathbf{L}\texttt{,}A\texttt{\},[}Rep(E_1)\texttt{,}\ldots\texttt{,}Rep(E_k)\texttt{]\}}.$$

- If G is an operator expression $E_1\ Op\ E_2$, where Op is a *RelationalOp* or an *EqualityOp*, and E_1, E_2 are guard expressions, then

$$Rep(G) = \texttt{\{op,}\mathbf{L}\texttt{,}Op\texttt{,}Rep(E_1)\texttt{,}Rep(E_2)\texttt{\}}.$$

All guard expressions are expressions and are represented in the same way as the corresponding expressions, cf. §B.4.

Appendix C

Portable hashing

The function *Hash* defined in this appendix is used as part of the definition of the BIF `erlang:hash/2` (§13.13.2). Given an arbitrary ERLANG term and a positive integer r, it returns an integer in the range $[0, r-1]$. The function has been designed with the aim to make it a good hash function, i.e., that it spreads function values evenly across the range.

C.1 Definitions

We make use of nine constants $C_1, \ldots, C_9$:

$$\begin{aligned}
C_1 &= 268440163 \\
C_2 &= 268439161 \\
C_3 &= 268435459 \\
C_4 &= 268436141 \\
C_5 &= 268438633 \\
C_6 &= 268437017 \\
C_7 &= 268438039 \\
C_8 &= 268437511 \\
C_9 &= 268439627
\end{aligned}$$

We will use a helper function *Foldl*, such that

$$\mathit{Foldl}(F, E, \langle v_1, \ldots, v_k \rangle) = F(v_k, F(v_{k-1}, \ldots F(v_2, F(v_1, E)) \ldots))$$

(Note that $\mathit{Foldl}(F, E, \langle\rangle) = E$, regardless of F.)
$w_1 \otimes w_2$ denotes the bitwise exclusive OR of w_1 and w_2.
All arithmetic operations in this appendix are modulo 2^{32}.

C.2 The hash function

The main function *Hash* is defined as follows:

$$\mathit{Hash}(\texttt{t}, r) = H(\texttt{t}, 0) \bmod r$$

The auxiliary function H is defined by cases.

- If `t` is an atom having a printname with character codes i_1, ..., i_k, where for all j, $1 \leq j \leq k$, $i_j \in [0, 255]$, then

  $$H(\texttt{t}, h) = C_1 * h + \mathit{Foldl}(F, 0, \langle i_1, \ldots, i_k \rangle),$$

  where

  $$\begin{aligned} F(i,h) &= G(16h + i) \\ G(j) &= (j \bmod 2^{28}) \otimes 16(\lfloor j/2^{28} \rfloor). \end{aligned}$$

  (Note that for any application of F, $i \in [0, 255]$ and $h \in [0, 2^{28} - 1]$, and for any application of G, $j \in [0, 2^{28} - 1]$.)

- If `t` is a fixnum, then

  $$H(\texttt{t}, h) = C_2 * h + (\Re^{-1}[t] \bmod 2^{32}).$$

  I'M NOT SURE I GOT THIS RIGHT AND I'D RATHER NOT MENTION FIXNUMS AND BIGNUMS!!!

- If `t` is a bignum where the 32-bit words of its absolute value in little-endian order is w_1, ..., w_k, then

  $$H(\texttt{t}, h) = C * \mathit{Foldl}(F, h, \langle w_1, \ldots, w_k \rangle) + k,$$

  where

  $$\begin{aligned} C &= C_2 && \text{if } \Re^{-1}[t] \geq 0; \\ &= C_3 && \text{if } \Re^{-1}[t] < 0. \\ F(w,h) &= C_2 * h + w \end{aligned}$$

- If `t` is `[]`, then

  $$H(\texttt{t}, h) = C_3 * h + 1.$$

- If `t` is a binary consisting of the bytes $i_1, \ldots, i_k$, then

  $$H(\texttt{t}, h) = C_4 * \mathit{Foldl}(F, h, \langle i_1, \ldots, i_l \rangle) + k,$$

  where

  $$\begin{aligned} l &= \min(k, 15) \\ F(i,h) &= C_1 * h + i. \end{aligned}$$

- If `t` is a PID, then

$$H(\mathtt{t}, h) = C_5 * h + \mathit{MagicPid}(\mathtt{t}).$$

- If `t` is a port or a ref, then

$$H(\mathtt{t}, h) = C_9 * h + \mathit{MagicPortRef}(\mathtt{t}).$$

- If `t` is a float represented by the two unsigned 32-bit quantities w_1 and w_2, then (THIS IS NOT VERY PORTABLE!!!)

$$H(\mathtt{t}, h) = C_6 * h + (w_1 \otimes w_2).$$

- If `t` is a term $\mathtt{[t}_1\mathtt{,} \ldots \mathtt{,t}_k \,|\, \mathtt{t}_{k+1}\mathtt{]}$, then

$$H(\mathtt{t}, h) = C_8 * H(\mathtt{t}_{k+1}, \mathit{Foldl}(H, h, \langle t_1, \ldots, t_k \rangle)).$$

- If `t` is a tuple $\mathtt{\{t}_1\mathtt{,} \ldots \mathtt{,t}_k\mathtt{\}}$, then

$$H(\mathtt{t}, h) = C_9 * \mathit{Foldl}(H, h, \langle t_1, \ldots, t_k \rangle) + k.$$

Appendix D

The external term format

The external term format is a representation of any ERLANG term as a sequence of bytes. It is used as part of the ERLANG distribution protocol but can also be accessed explicitly through the BIFs `binary_to_term/1` (§13.4.3) and `term_to_binary/1` (§13.4.8),in which a sequence of bytes is represented as a binary.

The version of the external term format described here is 4.7. It can be recognized by the sequence of bytes beginning with a byte `131`. Future versions of the external term format that are not compatible with version 4.7 must begin the sequence of bytes differently.

We will describe the external term format as a function $\mathit{TermRep}_{4.7}$ that given an ERLANG term returns a sequence of bytes. We use a mathematical notation rather than ERLANG function declarations, although it would not be difficult to transform our function definitions to an ERLANG program that returns a list of bytes.

D.1 Context

The transformation from a term to a sequence of bytes is context-dependent. We assume that we can access the name of the node on which a term `t` resides.

There is also an *atom table* that has 256 rows with keys 0 to 255, each of which has an ERLANG atom as value. Its purpose is to reduce the communication when terms are being transmitted between two nodes (although it would be possible to use an atom table also, for example, when writing terms to a file). The idea is that every node has one atom table for each of its friends (§11). Two nodes that are friends, say N_1 and N_2, agree to ensure that N_1's atom table for N_2 (i.e., `atom_tables[`N_1`](`N_2`)`, cf. §11.7.2) will have the same contents as N_2's atom table for N_1 (i.e., `atom_tables[`N_2`](`N_1`)`).

When an atom A is to be transmitted from node N_1 to node N_2, the following happens at node N_1. First a hash value I between 0 and 255 is computed for `A` (cf. §C). Then row `I` of N_1's atom table for N_2 is inspected. If that row has `A` as value, then only `I` is transmitted to N_2 because it is assumed that also N_2's atom table for N_1 has `A` as value for `I`. Otherwise row `I` of N_1's atom table for N_2 is updated to contain `A` and the whole printname of `A` is transmitted together with `I` to N_2. Note that sequences of bytes must be transformed back into terms at node N_2 in the same order as they were produced at node N_1, for the atom tables to have the correct contents at all times.

D.2 Definitions

We will use the following auxiliary functions. When we write that something is an error, it means that the transformation should fail.

- *BE*(k, i) (for Big Endian) is the sequence of bytes $\mathtt{b}_1, \ldots, \mathtt{b}_k$ such that *BigEndianValue*$(\langle \mathtt{b}_1, \ldots, \mathtt{b}_k \rangle)$ is i. It is an error if $i \notin [0, 256^k - 1]$.
- *BES*(k, i) (for Big Endian Signed) is the sequence of bytes $\mathtt{b}_1, \ldots, \mathtt{b}_k$ such that *BigEndianSignedValue*$(\langle \mathtt{b}_1, \ldots, \mathtt{b}_k \rangle)$ is i. It is an error if $i \notin [0, 256^k - 1]$.
- *ID*(`t`), where `t` is a ref or a port, is the value of `ID[t]`, which uniquely identifies the ref or port on the node on which it resides. If `t` is a PID, *ID*(`t`) is the XXX least significant bits of `ID[t]`, cf. *PS*(`t`).
- *FloatString*(f) is a sequence of 31 bytes where the first 26 (if f is nonnegative) or 27 (if f is negative) are the character codes of the string produced by the ISO C `printf` facility given the format string `"%.20e"` and the remaining 4 or 5 bytes are `0`.
- *CR*(`t`), where `t` is a ref, PID or port, is the value of `creation[t]` (a nonnegative integer), which distinguishes between different invocations of nodes with the same name. *CR*(`t`) must be in the range $[0, 255]$.
- *LE*(k, i) (for Little Endian) is the unique sequence of bytes $\mathtt{b}_1, \ldots, \mathtt{b}_k$ such that *LittleEndianValue*$(\langle \mathtt{b}_1, \ldots, \mathtt{b}_k \rangle)$ is i. It is an error if $i \notin [0, 256^k - 1]$.
- *Log*(k, i) is the base k logarithm of i rounded towards zero. It is an error if $i \leq 0$.
- *Node*(t) is an atom that names the node on which the term t resides.
- *PS*(`t`), where `t` is a PID is `ID[t]` with the XXX least significant bits removed, cf. *ID*(`t`).
- *SignBit*(i) is `0` if $i \geq 0$ and `1` if $i < 0$.

We write a sequence of bytes $b_1, \ldots, b_k$ as $\langle b_1, \ldots, b_k \rangle$. Juxtaposition denotes concatenation, so

$$\langle b_1, \ldots, b_k \rangle \langle b'_1, \ldots, b'_l \rangle = \langle b_1, \ldots, b_k, b'_1, \ldots, b'_l \rangle.$$

D.3 The transformation

$$\mathit{TermRep}_{4.7}(\mathtt{T}) = \langle \mathtt{131} \rangle \ \mathit{TR}_{4.7}(\mathtt{T})$$

The value of $\mathit{TR}_{4.7}(\mathtt{T})$ is defined by cases. When cases overlap, either case could be used but the more specific case is preferred.

- If $\mathtt{T}$ is an integer in the range $[0, 255]$, then

$$\mathit{TR}_{4.7}(\mathtt{T}) = \langle \mathtt{97} \rangle \ \langle \mathtt{T} \rangle.$$

- If $\mathtt{T}$ is an integer in the range $[-2^{31}, 2^{31} - 1]$ (but typically not in the range $[0, 255]$), then

$$\mathit{TR}_{4.7}(\mathtt{T}) = \langle \mathtt{98} \rangle \ \mathit{BES}(4, \mathtt{T}).$$

- If $\mathtt{T}$ is an integer in the range $[-(256^{255} - 1), 256^{255} - 1]$ (but typically not in the range $[-2^{31}, 2^{31} - 1]$), then

$$\mathit{TR}_{4.7}(\mathtt{T}) = \langle \mathtt{110}, l, \mathit{SignBit}(t) \rangle \ \mathit{LittleEndian}(l, \mathtt{T}),$$

  where $l = \mathit{Log}(256, |\mathtt{T}|) + 1$.

- If $\mathtt{T}$ is an integer in the range $[-(256^{2^{32}-1} - 1), 256^{2^{32}-1} - 1]$ (but typically not in the range $[-(256^{255} - 1), 256^{255} - 1]$), then

$$\mathit{TR}_{4.7}(\mathtt{T}) = \langle \mathtt{111} \rangle \ \mathit{BE}(4, l) \ \langle \mathit{SignBit}(\mathtt{T}) \rangle \ \mathit{LittleEndian}(l, |\mathtt{T}|),$$

  where $l = \mathit{Log}(256, |\mathtt{T}|) + 1$.

- If $\mathtt{T}$ is a float, then

$$\mathit{TR}_{4.7}(\mathtt{T}) = \langle \mathtt{99} \rangle, \mathit{FloatString}(\mathtt{T}).$$

- If $\mathtt{T}$ is an atom where the printname consists of k characters with codes $i_1, \ldots, i_k$, such that $k \leq 255$ and for all j, $1 \leq j \leq k$, $i_1 \in [0, 255]$, then

$$\mathit{TR}_{4.7}(\mathtt{T}) = \langle \mathtt{100}, \mathtt{0}, k, i_1, \ldots, i_k \rangle,$$

or if the atom is also to be stored in row p of the atom table (where $p \in [0,255]$), then

$$TR_{4.7}(\mathtt{T}) = \langle \mathtt{78}, p, \mathtt{0}, k, i_1, \ldots, i_k \rangle,$$

or if the atom is already stored in row p of the atom table (where $p \in [0,255]$), then

$$TR_{4.7}(\mathtt{T}) = \langle \mathtt{67}, p \rangle.$$

- If $\mathtt{T}$ is a ref, then

$$TR_{4.7}(\mathtt{T}) = \langle \mathtt{101} \rangle \; TR_{4.7}(Node(\mathtt{T})) \; BE(4, ID(\mathtt{T})) \; \langle CR(\mathtt{T}) \rangle.$$

- If $\mathtt{T}$ is a port, then

$$TR_{4.7}(\mathtt{T}) = \langle \mathtt{102} \rangle \; TR_{4.7}(Node(\mathtt{T})) \; BE(4, ID(\mathtt{T})), \langle CR(\mathtt{T}) \rangle.$$

- If $\mathtt{T}$ is a PID, then

$$TR_{4.7}(\mathtt{T}) = \langle \mathtt{103} \rangle \; TR_{4.7}(Node(\mathtt{T})) \; BE(4, ID(\mathtt{T})) \; BE(4, PS(\mathtt{T})) \langle CR(\mathtt{T}) \rangle.$$

- If $\mathtt{T}$is a tuple with elements $\mathtt{T}_1, \ldots, \mathtt{T}_k$, where $k \le 255$, then

$$TR_{4.7}(\mathtt{T}) = \langle \mathtt{104}, k \rangle \; TR_{4.7}(\mathtt{T}_1) \; \cdots \; TR_{4.7}(\mathtt{T}_k).$$

- If $\mathtt{T}$ is a tuple with elements $\mathtt{T}_1, \ldots, \mathtt{T}_k$, where $k \le 2^{32} - 1$, then

$$TR_{4.7}(\mathtt{T}) = \langle \mathtt{104} \rangle \; BE(4, k) \; TR_{4.7}(\mathtt{T}_1) \; \cdots \; TR_{4.7}(\mathtt{T}_k).$$

- If $\mathtt{T}$ is `[]`, then

$$TR_{4.7}(\mathtt{T}) = \langle \mathtt{106} \rangle.$$

- If $\mathtt{T}$ is a (typically nonempty) string of Latin-1 characters having codes $i_1, \ldots, i_k$, where $k \le 2^{16} - 1$, then

$$TR_{4.7}(\mathtt{T}) = \langle \mathtt{107} \rangle \; BE(2, k) \; \langle i_1, \ldots, i_k \rangle.$$

- If $\mathtt{T}$ is a term $\mathtt{[T_1,} \; \ldots \; \mathtt{,T_k \,|\, T_{k+1}]}$, where $k \le 2^{32} - 1$, (but typically not a string covered by the previous case) then

$$TR_{4.7}(\mathtt{T}) = \langle \mathtt{108} \rangle \; BE(4, k) \; TR_{4.7}(\mathtt{T}_1) \; \cdots \; TR_{4.7}(\mathtt{T}_{k+1}).$$

- If $\mathtt{T}$ is a binary with elements $i_1, \ldots, i_k$, where $k \le 2^{32} - 1$, then

$$TR_{4.7}(\mathtt{T}) = \langle \mathtt{109} \rangle \; BE(4, k) \; \langle i_1, \ldots, i_k \rangle.$$

- Otherwise, it is an error.

The inverse transformation is obvious.

Appendix E

Grammar

E.1 The lexical grammar

ErlangUppercase:
  the capital ASCII letters `A–Z` (`\101` to `\132`)

ErlangLowercase:
  the small ASCII letters `a–z` (`\141` to `\172`)

ErlangLetter:
  ErlangLowercase
  ErlangUppercase

ErlangDigit:
  the ASCII decimal digits `0–9` (`\060` to `\071`)

LineTerminator:
  the LF character ("linefeed" or "newline")

InputCharacter:
  AsciiInputCharacter but not LF

Input:
  $\textit{InputElements}_{opt}$

InputElements:
  InputElement
  InputElements InputElement

InputElement:
  WhiteSpace
  Comment
  Token

Token:
: *Separator*
: *Keyword*
: *Operator*
: *IntegerLiteral*
: *FloatLiteral*
: *CharLiteral*
: *StringLiteral*
: *AtomLiteral Variable*
: *UniversalPattern*

WhiteSpace:
: *LineTerminator*
: *ControlCharacter*
: the ASCII SP character, also known as “space”

ControlCharacter:
: any ASCII control character (`\000` to `\037`)

Comment:
: `%` *InputCharacters*$_{opt}$ *LineTerminator*

InputCharacters:
: *InputCharacter*
: *InputCharacters InputCharacter*

Separator: one of

```
(    )    {   }   [    ]    .        :
|    ||   ;    ,    ?    ->   #
```

Keyword: one of

```
after    cond     let      when
begin    end      of
case     fun      query
catch    if       receive
```

Operator: one of

```
+    -    *    /    div  rem
or   xor  bor  bxor bsl  bsr  and  band
==   /=   =:=  =/=  <    =<   >    >=
not  bnot ++   --   =    !    <-
```

EscapeSequence:

```
\ b              % \008:  backspace BS
\ d              % \177:  delete DEL
\ e              % \033:  escape ESC
\ f              % \014:  form feed FF
\ n              % \012:  linefeed LF
\ r              % \015:  carriage return CR
\ s              % \040:  space SPC
\ t              % \011:  horizontal tab HT
\ v              % \013:  vertical tab VT
\ \              % \008:  backslash \
ControlEscape    % \000 to \037:  64 less than the char
\ '              % \047:  single quote '
\ "              % \042:  double quote "
OctalEscape      % \000 to \777:  from octal value
```

ControlEscape:
  `\ ^` *ControlName*

ControlName:
  any character between `\100` and `\137`

OctalEscape:
  `\` *OctalDigit*
  `\` *OctalDigit OctalDigit*
  `\` *OctalDigit OctalDigit OctalDigit*

OctalDigit: one of
  `0 1 2 3 4 5 6 7`

IntegerLiteral:
  DecimalLiteral
  ExplicitRadixLiteral

DecimalLiteral:
  Digits[10]

ExplicitRadixLiteral:
  `2 #` *Digits*[2]
  `3 #` *Digits*[3]
  `4 #` *Digits*[4]
  `5 #` *Digits*[5]
  `6 #` *Digits*[6]
  `7 #` *Digits*[7]
  `8 #` *Digits*[8]
  `9 #` *Digits*[9]
  `1 0 #` *Digits*[10]
  `1 1 #` *Digits*[11]
  `1 2 #` *Digits*[12]
  `1 3 #` *Digits*[13]
  `1 4 #` *Digits*[14]
  `1 5 #` *Digits*[15]
  `1 6 #` *Digits*[16]

Digits[*k*]:
 Digit[*k*]
 Digits[*k*] *Digit*[*k*]

Digit[*k*]: one of the first *k* of
 `0 1 2 3 4 5 6 7 8 9 Aa Bb Cc Dd Ee Ff`

FloatLiteral:
 DecimalLiteral `.` *DecimalLiteral* $ExponentPart_{opt}$

ExponentPart:
 ExponentIndicator $Sign_{opt}$ *DecimalLiteral*

Sign: one of
 `+ -`

ExponentIndicator: one of
 `E e`

CharLiteral:
 `$` *CharLiteralChar*

CharLiteralChar:
 InputCharacter
 EscapeSequence

StringLiteral:
 `"` $StringCharacters_{opt}$ `"`

StringCharacters:
 StringCharacter
 StringCharacters *StringCharacter*

StringCharacter:
 InputCharacter but not *ControlCharacter* or \ or `"`
 EscapeSequence

AtomLiteral:
 AtomLiteralChars but not a *Keyword* or *Operator*
 `'` $QuotedCharacters_{opt}$ `'`

AtomLiteralChars:
 ErlangLowercase $NameChars_{opt}$

NameChars:
 NameChar
 NameChars *NameChar*

NameChar:
 ErlangLetter
 ErlangDigit
 `@`
 `_`

QuotedCharacters:
 QuotedCharacter
 QuotedCharacters *QuotedCharacter*

QuotedCharacter:
  InputCharacter but not *ControlCharacter* or \ or '
  EscapeSequence

Variable:
  _ *NameChars*
  ErlangUppercase *NameChars*$_{opt}$

UniversalPattern:
  _

TerminatedTokens:
  TokenSequences$_{opt}$

TokenSequences:
  TokenSequence
  TokenSequences *TokenSequence*

TokenSequence:
  Tokens *FullStop*

Tokens:
  Token
  Tokens *Token*

FullStop:
  . *WhiteSpace*
  . *Comment*

E.2 The main grammar

ModuleDeclaration:
  FileAttributes$_{opt}$ *ModuleAttribute* *HeaderForms*$_{opt}$ *ProgramForms*$_{opt}$

FileAttributes:
  FileAttribute
  FileAttributes *FileAttribute*

ModuleAttribute:
  `- module (` *ModuleName* `)` *FullStop*

ModuleName:
  AtomLiteral

HeaderForms:
  HeaderForm
  HeaderForms *HeaderForm*

HeaderForm:
  HeaderAttribute
  AnywhereAttribute

HeaderAttribute:
  ExportAttribute
  ImportAttribute
  CompileAttribute
  WildAttribute

AnywhereAttribute:
  FileAttribute
  MacroDefinition
  RecordDeclaration

ExportAttribute:
  `-` `export` `(` *FunctionNameList* `)` *FullStop*

FunctionNameList:
  `[` $FunctionNames_{opt}$ `]`

FunctionNames:
  FunctionName
  FunctionNames `,` *FunctionName*

FunctionName:
  FunctionSymbol `/` *Arity*

FunctionSymbol:
  AtomLiteral

Arity:
  IntegerLiteral

ImportAttribute:
  `-` `import` `(` *ModuleName* `,` *FunctionNameList* `)` *FullStop*

CompileAttribute:
  `-` `compile` `(` `[` $Terms_{opt}$ `]` `)` *FullStop*

FileAttribute:
  `-` `file` `(` *StringLiteral* `,` *LineNumeral* `)` *FullStop*

LineNumeral:
  IntegerLiteral

WildAttribute:
  `-` *AtomLiteral* `(` *Term* `)` *FullStop*

ProgramForms:
  FunctionDeclaration
  ProgramForms *FunctionDeclaration*
  ProgramForms *AnywhereAttribute*

FunctionDeclaration:
  FunctionClauses *FullStop*

FunctionClauses:
  FunctionClause
  FunctionClauses `;` *FunctionClause*

FunctionClause:
  FunctionSymbol FunClause

RecordDeclaration:
  `- record (` *RecordType* `,` *RecordDeclTuple* `)` *FullStop*

RecordDeclTuple:
  `{` *RecordFieldDecls*$_{opt}$ `}`

RecordFieldDecls:
  RecordFieldDecl
  RecordFieldDecls `,` *RecordFieldDecl*

RecordFieldDecl:
  RecordFieldName RecordFieldValue$_{opt}$

Pattern:

AtomicLiteral	(§6.19.2)
Variable	(§3.16)
UniversalPattern	(§3.17)
TuplePattern	
RecordPattern	
ListPattern	

TuplePattern:
  `{` *Patterns*$_{opt}$ `}`

ListPattern:
  `[ ]`
  `[` *Patterns ListPatternTail*$_{opt}$ `]`

ListPatternTail:
  `|` *Pattern*

Patterns:
  Pattern
  Patterns `,` *Pattern*

RecordPattern:
  `#` *RecordType RecordPatternTuple*

RecordType:
  AtomLiteral

RecordPatternTuple:
  `{` *RecordFieldPatterns*$_{opt}$ `}`

RecordFieldPatterns:
  RecordFieldPattern
  RecordFieldPatterns `,` *RecordFieldPattern*

RecordFieldPattern:
  RecordFieldName `=` *Pattern*

RecordFieldName:
  AtomLiteral

Body:
    Exprs

Exprs:
    Expr
    Exprs `,` *Expr*

Expr:
    `catch` *Expr*
    MatchExpr

MatchExpr:
    Pattern `=` *MatchExpr*
    SendExpr

SendExpr:
    CompareExpr `!` *SendExpr*
    CompareExpr

CompareExpr:
    ListConcExpr RelationalOp ListConcExpr
    ListConcExpr EqualityOp ListConcExpr
    ListConcExpr

RelationalOp: one of
```
<    =<   >    >=
```

EqualityOp: one of
```
=:=  =/=  ==   /=
```

ListConcExpr:
    AdditionShiftExpr ListConcOp ListConcExpr
    AdditionShiftExpr

ListConcOp: one of
```
++   --
```

AdditionShiftExpr:
    AdditionShiftExpr AdditionOp MultiplicationExpr
    AdditionShiftExpr ShiftOp MultiplicationExpr
    AdditionShiftExpr `or` *MultiplicationExpr*
    AdditionShiftExpr `xor` *MultiplicationExpr*
    MultiplicationExpr

AdditionOp: one of
```
+    -
bor  bxor
```

ShiftOp: one of
```
bsl  bsr
```

MultiplicationExpr:
    MultiplicationExpr MultiplicationOp PrefixOpExpr
    MultiplicationExpr `and` *PrefixOpExpr*
    PrefixOpExpr

MultiplicationOp: one of
`*` `/`
`div` `rem`
`band`

PrefixOpExpr:
PrefixOp RecordExpr
RecordExpr

PrefixOp: one of
`+` `-`
`bnot` `not`

RecordExpr:
$\mathit{RecordExpr}_{opt}$ `#` *RecordType* `.` *RecordFieldName*
$\mathit{RecordExpr}_{opt}$ `#` *RecordType RecordUpdateTuple*
ApplicationExpr

RecordUpdateTuple:
`{` $\mathit{RecordFieldUpdates}_{opt}$ `}`

RecordFieldUpdates:
RecordFieldUpdate
RecordFieldUpdates `,` *RecordFieldUpdate*

RecordFieldUpdate:
RecordFieldName RecordFieldValue

RecordFieldValue:
`=` *Expr*

ApplicationExpr:
PrimaryExpr `(` Exprs_{opt} `)`
PrimaryExpr `:` *PrimaryExpr* `(` Exprs_{opt} `)`
PrimaryExpr

PrimaryExpr:
Variable
AtomicLiteral
TupleSkeleton
ListSkeleton
ListComprehension
BlockExpr
IfExpr
CaseExpr
ReceiveExpr
FunExpr
QueryExpr
ParenthesizedExpr

AtomicLiteral:
    IntegerLiteral
    FloatLiteral
    CharLiteral
    StringLiterals
    AtomLiteral

StringLiterals:
    StringLiteral
    StringLiterals StringLiteral

TupleSkeleton:
    `{` $Exprs_{opt}$ `}`

ListSkeleton:
    `[ ]`
    `[` *Exprs* $ListSkeletonTail_{opt}$ `]`

ListSkeletonTail:
    `|` *Expr*

ListComprehension:
    `[` *Expr* `||` *ListComprehensionExprs* `]`

ListComprehensionExprs:
    ListComprehensionExpr
    ListComprehensionExprs `,` *ListComprehensionExpr*

ListComprehensionExpr:
    Generator
    Filter

Generator:
    Pattern `<-` *Expr*

Filter:
    Expr

BlockExpr:
    `begin` *Body* `end`

IfExpr:
    `if` *IfClauses* `end`

IfClauses:
    IfClause
    IfClauses `;` *IfClause*

IfClause:
    Guard ClauseBody

ClauseBody:
    `->` *Body*

CaseExpr:
    `case` *Expr* `of` *CrClauses* `end`

CrClauses:
    CrClause
    CrClauses `;` *CrClause*

CrClause:
    Pattern $ClauseGuard_{opt}$ *ClauseBody*

ClauseGuard:
    `when` *Guard*

Guard:
    Body

ReceiveExpr:
    `receive` *CrClauses* `end`
    `receive` $CrClauses_{opt}$ `after` *Expr* *ClauseBody* `end`

FunExpr:
    `fun` *FunctionArity*
    `fun` *FunClauses* `end`

FunClauses:
    FunClause
    FunClauses `;` *FunClause*

FunClause:
    `(` $Patterns_{opt}$ `)` $ClauseGuard_{opt}$ *ClauseBody*

QueryExpr:
    `query` *ListComprehension* `end`

RecordExpr:
    RecordExpr `.` *RecordFieldName*

ParenthesizedExpr:
    `(` *Expr* `)`

Guard:
    GuardTest
    Guard `,` *GuardTest*

GuardTest:
    `true`
    GuardRecordTest
    GuardRecognizer
    GuardTermComparison
    ParenthesizedGuardTest

GuardRecordTest:
    `record (` *GuardExpr* `,` *RecordType* `)`

GuardRecognizer:
    RecognizerBIF `(` *GuardExpr* `)`

RecognizerBIF:
    AtomLiteral

GuardTermComparison:
    GuardExpr RelationalOp GuardExpr
    GuardExpr EqualityOp GuardExpr

ParenthesizedGuardTest:
    `(` *GuardTest* `)`

GuardExpr:
    GuardAdditionShiftExpr

GuardAdditionShiftExpr:
    GuardAdditionShiftExpr AdditionOp GuardMultiplicationExpr
    GuardAdditionShiftExpr ShiftOp GuardMultiplicationExpr
    GuardMultiplicationExpr

GuardMultiplicationExpr:
    GuardMultiplicationExpr MultiplicationOp GuardPrefixOpExpr
    GuardPrefixOpExpr

GuardPrefixOpExpr:
    PrefixOp GuardApplicationExpr
    GuardApplicationExpr

GuardApplicationExpr:
    GuardBIF `(` *GuardExprs*$_{opt}$ `)`
    GuardRecordExpr
    GuardPrimaryExpr

GuardBIF:
    AtomLiteral

GuardExprs:
    GuardExpr
    GuardExprs `,` *GuardExpr*

GuardRecordExpr:
    GuardPrimaryExpr$_{opt}$ `#` *AtomLiteral* `.` *AtomLiteral*

GuardPrimaryExpr:
    Variable
    AtomicLiteral
    GuardListSkeleton
    GuardTupleSkeleton
    ParenthesizedGuardExpr

GuardListSkeleton:
    `[]`
    `[` *GuardExprs GuardListSkeletonTail*$_{opt}$ `]`

GuardListSkeletonTail:
    `|` *GuardExpr*

GuardTupleSkeleton:
    `{` *GuardExprs*$_{opt}$ `}`

ParenthesizedGuardExpr:
    `(` *GuardExpr* `)`

E.3 The preprocessor grammar

Directive:

MacroDefinition	(§7.2.1)
MacroUndefinition	(§7.2.2)
IncludeDirective	(§7.3)
IncludeLibDirective	(§7.3)
IfdefDirective	(§7.4)
IfndefDirective	(§7.4)
ElseDirective	(§7.4)
EndifDirective	(§7.4)

MacroDefinition:
  `- define (` *MacroName* $MacroParams_{opt}$ `,` *MacroBody* `)` *FullStop*

MacroName:
  AtomLiteral
  Variable

MacroParams:
  `(` $Variables_{opt}$ `)`

Variables:
  Variable
  Variables `,` *Variable*

MacroBody:
  Tokens

MacroUndefinition:
  `- undef (` *MacroName* `)` *FullStop*

MacroApplication:
  `?` *MacroName*
  `?` *MacroName* `(` $MacroArguments_{opt}$ `)`

MacroArguments:
  MacroArgument
  MacroArguments `,` MacroArgument

MacroArgument:
  BalancedExpr that is not one of `,` or `)`

BalancedExpr:
  `(` *BalancedExprs* `)`
  `[` *BalancedExprs* `]`
  `{` *BalancedExprs* `}`
  `begin` *BalancedExprs* `end`
  `if` *BalancedExprs* `end`
  `case` *BalancedExprs* `end`
  `receive` *BalancedExprs* `end`
  `query` *BalancedExprs* `end`
  OtherToken
  BalancedExpr BalancedExpr

IncludeDirective:
  `- include (` *IncludeFileName* `)` *FullStop*

IncludeLibDirective:
  `- include_lib (` *IncludeFileName* `)` *FullStop*

IncludeFileName:
  OneStringLiteral

IfdefDirective:
  `- ifdef (` *MacroName* `)` *FullStop*

IfndefDirective:
  `- ifndef (` *MacroName* `)` *FullStop*

ElseDirective:
  `- else` *FullStop*

EndifDirective:
  `- endif` *FullStop*

Index