Core Erlang 1.0
language specification

Richard Carlsson Björn Gustavsson Erik Johansson
Thomas Lindgren Sven-Olof Nyström Mikael Pettersson
Robert Virding

November 14, 2000

Abstract

We describe a core language for the concurrent functional language Erlang, aptly named "Core Erlang", presenting its grammar and informal static and dynamic semantics relative to Erlang. We also discuss built-in functions and other open issues, and sketch a syntax tree representation.

1 Motivation

Core Erlang is an intermediate representation of Erlang, intended to lie at a level between source code and the intermediate code typically found in compilers.

During its evolution, the syntax of Erlang has become somewhat complicated, making it difficult to develop programs that operate on the source. Such programs might be new parsers, optimisers that transform source code, and various instrumentations on the source code, for example profilers and debuggers.

Core Erlang should meet the following goals:

- Core Erlang should be as regular as possible, to facilitate the development of code-walking tools.
- Core Erlang should have a clear and simple semantics.
- Core Erlang should be straight-forward to translate to every intermediate code used in any Erlang implementation; similarly, it should be straightforward to translate from Erlang programs to equivalent Core Erlang programs.
- There should exist a well-defined textual representation of Core Erlang, with a simple and preferably unambiguous grammar, making it easy to construct tools for printing and reading programs. This representation should be possible to use for communication between tools using different internal representations of Core Erlang.

- The textual representation should be easy for humans to read and edit, in case the developer wants to check what the ERLANG source looks like in CORE ERLANG form, inspect – or modify – the results of code transformations, or maybe write some CORE ERLANG code by hand.

These goals force CORE ERLANG to a fairly high level of abstraction. It is not, for example, possible to break down the `receive` construct into operations that operate on the mailbox, since no such (useful) translation would be compatible with all ERLANG implementations.

Section 2 discusses lexical analysis and parsing. Section 3 gives the grammar for the language and section 4 the static semantics. Section 5 describes evaluation of programs and expressions. Section 6 discusses issues that may need further specification. Appendix A contains a quick reference to the language and appendix B lists character escape sequences. Appendix C shows a simple syntax tree representation.

2 Lexical analysis and parsing

We discuss the lexical processing of a CORE ERLANG program in terms of operations on a sequence of Unicode [4] characters (of which both ASCII and Latin-1 [3], ISO/IEC 8859-1, are subsets), such that there are no Unicode escapes (written `\uXXXX`, where each `X` is a hexadecimal digit) in the sequence. Note, though, that it is *not* required that tools handling CORE ERLANG source programs use Unicode for input and output; like ERLANG, no part of the written language *per se* requires characters outside the 7-bit ASCII subset. However, in order to support STANDARD ERLANG [2], tools must be able to handle Unicode encodings of CORE ERLANG character, string and atom literals.

We assume that the translation from the sequence of characters into a sequence of tokens, suitable for parsing according to the grammar of the following section, is straightforward, being very similar to that of ERLANG. In CORE ERLANG, atom literals are always single-quoted, to avoid any possible confusion with keywords. Comments on any source code line in CORE ERLANG, like in ERLANG, begin with the leftmost percent character '`%`', (`\u0025`) on that line that does not occur as part of an atom, string or character literal, and continue up to (but not including) the first line terminator following that character. Comments are ignored by the tokenisation, in effect only recognising the line terminator.

A line terminator is defined as the longest sequence of input characters consisting of exactly one ASCII `CR` (`\u000d`), one ASCII `LF` (`\u000a`), or one `CR` followed by one `LF`. Line terminators are generally treated as whitespace, except in atom and string literals where line terminators are not allowed. CORE ERLANG does not distinguish between periods ('`.`' characters) that are followed by whitespace (called *FullStop* tokens in the ERLANG Reference Manual [1]) and those that are not (*i. e.*, ordinary separator periods).

The tokenisation should concatenate all adjacent string literals (these may be separated by any number of whitespace characters, line terminators and comments). Thus, the text "`"Hey" "Ho"`" denotes the same string literal as `"HeyHo"`. This allows strings to be split over several lines.

3 Grammar

This section describes the basic grammar of Core Erlang programs. An effort has been made to reduce the language as far as it is possible – and practical – while maintaining readability and preserving most of the lexical conventions of Erlang. For instance, it can be noted that the syntactic distinction between variables and function names is not really necessary, but makes the connection between exported function names and calls to locally bound functions more obvious to the eye than if "plain" Erlang-style variables were used for all bindings.

3.1 Notation

Literals are described using ordinary regular expressions, where ? stands for "zero or one occurrence of", + for "one or more repetitions of", and ellipsis (...) indicates repetition over a range of characters; no other symbols are used except parentheses and the standard | for alternative choices and $*$ for zero or more repetitions.

For some widely used grammar rules we use abbreviations, such as i for *Integer* and v for *AnnotatedVariable*. The abbreviations are given within parentheses by the corresponding rules.

To further keep the presentation compact, we use ellipsis notation with indices ("$x_1 \ldots x_n$") instead of giving explicit recursive rules for (possibly empty) sequences.

3.2 Lexical definitions

sign	::=	+ | -
digit	::=	0 | 1 | ... | 9
uppercase	::=	A | ... | Z | \u00c0 | ... | \u00d6 | \u00d8 | ... | \u00de
lowercase	::=	a | ... | z | \u00df | ... | \u00f6 | \u00f8 | ... | \u00ff
inputchar	::=	any character except CR and LF
control	::=	\u0000 | ... | \u001f
space	::=	\u0020
namechar	::=	*uppercase* | *lowercase* | *digit* | @ | _
escape	::=	\ (*octal* | (^ *ctrlchar*) | *escapechar*)
octaldigit	::=	0 | 1 | ... | 7
octal	::=	*octaldigit* (*octaldigit* *octaldigit*?)?
ctrlchar	::=	\u0040 | ... | \u005f
escapechar	::=	b | d | e | f | n | r | s | t | v | " | ' | \

An escape sequence \ *octal* denotes the character whose Unicode value is given by the octal numeral. An escape sequence \ ^ *ctrlchar* denotes the character whose Unicode value is 64 less than that of the *ctrlchar*. The meanings of escape sequences \ *escapechar* are defined in appendix B.

3.3 Terminals

Integer (i):

$sign?\ digit^{+}$

Float:

$sign?\ digit^{+}$ `.` $digit^{+}$ $((\texttt{E} \mid \texttt{e})\ sign?\ digit^{+})?$

Atom (a):

`'` $((inputchar$ except $control$ and `\` and `'` $) \mid escape)^{*}$ `'`

Char:

`$` $((inputchar$ except $control$ and $space$ and `\` $) \mid escape)$

String:

`"` $((inputchar$ except $control$ and `\` and `"` $) \mid escape)^{*}$ `"`

VariableName:

$(uppercase \mid (\_\ namechar))\ namechar^{*}$

Note that a single underscore character '_' is not a valid *VariableName*.

3.4 Non-terminals

AnnotatedModule:

Module

`(` *Module* `-|` `[` c_1 `,` ... `,` c_n `]` `)` $(n \geq 0)$

Module:

`module` *a* *ModuleHeader* *ModuleBody* `end`

ModuleHeader:

Exports *Attributes*

Exports:

`[` $FunctionName_1$ `,` ... `,` $FunctionName_n$ `]` $(n \geq 0)$

FunctionName (a/i):

a `/` *i*

where *a* is called the *identifier*, and *i* the *arity*.

Attributes:

`attributes` `[` $ModuleAttribute_1$ `,` ... `,` $ModuleAttribute_n$ `]` $(n \geq 0)$

ModuleAttribute:

a `=` c

where a is called the *key*, and c the *value* of the attribute.

ModuleBody:

$\mathit{FunctionDefinition}_1 \cdots \mathit{FunctionDefinition}_n$ $(n \geq 0)$

FunctionDefinition:

AnnotatedFunctionName `=` *AnnotatedFun*

AnnotatedFunctionName:

FunctionName

`(` *FunctionName* `-|` `[` c_1 `,` $\ldots$ `,` c_n `]` `)` $(n \geq 0)$

AnnotatedFun:

Fun

`(` *Fun* `-|` `[` c_1 `,` $\ldots$ `,` c_n `]` `)` $(n \geq 0)$

Constant (c):

AtomicLiteral

$\{$ c_1 `,` $\ldots$ `,` c_n $\}$ $(n \geq 0)$

`[` c_1 `,` $\ldots$ `,` c_n `]` $(n \geq 1)$

`[` c_1 `,` $\ldots$ `,` c_{n-1} `|` c_n `]` $(n \geq 2)$

AtomicLiteral:

Integer

Float

Atom

Nil

Char

String

Nil:

`[ ]`

AnnotatedVariable (v):

VariableName

`(` *VariableName* `-|` `[` c_1 `,` $\ldots$ `,` c_n `]` `)` $(n \geq 0)$

AnnotatedPattern (p):

v[1]

Pattern

`(` *Pattern* `-|` `[` c_1 `,` $\ldots$ `,` c_n `]` `)` $(n \geq 0)$

Pattern:

AtomicLiteral

`{` p_1 `,` $\ldots$ `,` p_n `}` $(n \geq 0)$

`[` p_1 `,` $\ldots$ `,` p_n `]` $(n \geq 1)$

`[` p_1 `,` $\ldots$ `,` p_{n-1} `|` p_n `]` $(n \geq 2)$

v `=` p

where the last form v `=` p is called an *alias pattern*.

Expression (e):

AnnotatedValueList

AnnotatedSingleExpression

AnnotatedValueList:

ValueList

`(` *ValueList* `-|` `[` c_1 `,` $\ldots$ `,` c_n `]` `)` $(n \geq 0)$

ValueList:

`<` *AnnotatedSingleExpression*$_1$ `,` $\ldots$ `,`
AnnotatedSingleExpression$_n$ `>` $(n \geq 0)$

AnnotatedSingleExpression:

SingleExpression

`(` *SingleExpression* `-|` `[` c_1 `,` $\ldots$ `,` c_n `]` `)` $(n \geq 0)$

SingleExpression:

AtomicLiteral

VariableName

FunctionName

Tuple

List

Let

[1]The separation of variables from other patterns is necessary to keep the grammar LALR(1)

Case

Fun

Letrec

Application

InterModuleCall

PrimOpCall

Try

Receive

Protected

Sequencing

Catch

Tuple:

{ e_1 , ... , e_n } $\quad (n \geq 0)$

Note that this includes the 0-tuple { } and 1-tuples $\{x\}$.

List:

`[` e_1 , ... , e_n `]` $\quad (n \geq 1)$

`[` e_1 , ... , e_{n-1} `|` e_n `]` $\quad (n \geq 2)$

Let:

`let` *Variables* `=` e_1 `in` e_2

Variables:

v

`<` v_1 , ... , v_n `>` $\quad (n \geq 0)$

Case:

`case` e `of` *AnnotatedClause*$_1$ $\cdots$ *AnnotatedClause*$_n$ `end` $\quad (n \geq 1)$

AnnotatedClause:

Clause

`(` *Clause* `-|` `[` c_1 , ... , c_n `]` `)` $\quad (n \geq 0)$

Clause:

Patterns *Guard* `->` e

Patterns:

p

`<` p_1 `,` $\ldots$ `,` p_n `>` $(n \geq 0)$

Guard:

`when` e

Fun:

`fun (` v_1 `,` $\ldots$ `,` v_n `) ->` e $(n \geq 0)$

Note that there is no `end` keyword terminating the expression.

Letrec:

`letrec` $\mathit{FunctionDefinition}_1 \cdots \mathit{FunctionDefinition}_n$ `in` e $(n \geq 0)$

Application:

`apply` e_0 `(` e_1 `,` $\ldots$ `,` e_n `)` $(n \geq 0)$

InterModuleCall:

`call` e'_1 `:` e'_2 `(` e_1 `,` $\ldots$ `,` e_n `)` $(n \geq 0)$

PrimOpCall:

`primop` a `(` e_1 `,` $\ldots$ `,` e_n `)` $(n \geq 0)$

Try:

`try` e_1 `catch (`v_1`,` v_2`) ->` e_2

Receive:

`receive` $\mathit{AnnotatedClause}_1 \cdots \mathit{AnnotatedClause}_n$ *Timeout* `end` $(n \geq 0)$

Timeout:

`after` e_1 `->` e_2

where e_1 is called the *expiry expression* and e_2 the *expiry body.*

Protected:

`protected` e

Sequencing:

`do` e_1 e_2

Catch:

`catch` e

4 Static semantics

4.1 Annotations

An annotation `(` $\bullet$ `-|` `[`c_1`,` ..., c_n`]` `)` associates a list of constant literals c_1, ..., c_n with the enclosed phrase $\bullet$. Annotations are always optional; leaving out an annotation is equivalent to specifying an empty annotation list. The interpretation of annotations on program phrases is implementation-dependent.

4.2 Module definitions

The general form of a module definition is:

$$\begin{array}{l}
\texttt{module}\ a\ \texttt{[}\mathit{FunctionName}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{FunctionName}_n\texttt{]}\\
\quad\texttt{attributes}\ \texttt{[}\mathit{ModuleAttribute}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{ModuleAttribute}_{n'}\texttt{]}\\
\quad\mathit{FunctionDefinition}_1\ \cdots\ \mathit{FunctionDefinition}_{n''}\\
\texttt{end}
\end{array}$$

(*cf.* p. 4), where the atom a is the name of the module.

For each *FunctionName* a/i listed in the *Exports* declaration, it is a compile-time error if the function name a/i does not occur on the left-hand side of a *FunctionDefinition* in the corresponding *ModuleBody*.

For each *ModuleAttribute* a `=` c listed in the *Attributes* declaration, it is a compile-time error if there exists more than one *ModuleAttribute* with the same key a in the list. The interpretation of module attributes is implementation-dependent.

Each *FunctionDefinition* in the *ModuleBody* associates a *FunctionName* a_k/i_k with a *Fun* f_k. It is a compile-time error if the number of parameters of the right-hand side f_k does not equal the left-hand side arity i_k. The scope of each such function definition is the whole of the corresponding *Module*; see evaluation of *InterModuleCall* expressions (p. 14) for details. It is a compile-time error if the same function name a/i occurs on the left-hand side of two function definitions D_j, D_k, $j \neq k$, in a *ModuleBody* $D_1 \cdots D_n$. (*Cf. Letrec* expressions, p. 10.) A function name thus defined in the module body is said to be *exported* by the module if and only if it is also in the *Exports* declaration of the module.

4.3 Atomic literals

A *String* (p. 4) is defined as shorthand for the corresponding list of *Char* literals (*cf. List*, p. 7). *E. g.*, `"Hi!"` denotes the list `[$H, $i, $!]`. Also recall that the tokenisation process will concatenate all adjacent *String* literals that are separated only by whitespace and/or comments.

4.4 Lists

For lists in general, the following equivalences are defined:

$$\texttt{[}\ x_1\texttt{,}\ \ldots\texttt{,}\ x_n\ \texttt{]} \quad ::= \quad \texttt{[}\ x_1\texttt{,}\ \ldots\texttt{,}\ x_n\ \texttt{|}\ \texttt{[]}\ \texttt{]}$$

for $n \geq 1$, and

$$\texttt{[}\ x_1\texttt{,}\ \ldots\texttt{,}\ x_{n-1}\ \texttt{|}\ x_n\ \texttt{]} \quad ::= \quad \texttt{[}\ x_1\texttt{,}\ \ldots\texttt{,}\ x_{n-2}\ \texttt{|}\ \texttt{[}\ x_{n-1}\ \texttt{|}\ x_n\ \texttt{]}\ \texttt{]}$$

for $n \geq 3$. Thus, every list (p. 7) can be equivalently written on a unique normal form using only the *list constructor* primitive[2] `[ • | • ]`, and the constant literal `[]` (*Nil*). This also applies to lists in constants (p. 5) and patterns (p. 6).

4.5 Expressions

- For a *VariableName* or *FunctionName* expression, it is a compile-time error if the occurrence is not within the scope of a corresponding binding. A *VariableName* can only be bound in a *Let*, a *Clause*, a *Fun*, or a *Try*, while a *FunctionName* can only be bound in a *FunctionDefinition* of a *Module* or a *Letrec*.

- In *Fun* (p. 8) expressions `fun (`v_1`, ...,` v_n`) ->` e, and in *Let* (p. 7) expressions `let <`v_1`, ...,` v_n`> =` e_1 `in` e_2, no variable name may occur more than once in $v_1, \ldots, v_n$. Likewise, in *Try* (p. 8) expressions `try` e_1 `catch (`v_1`,` v_2`) ->` e_2, v_1 and v_2 must be distinct variables. A *Let* expression `let <`v`> =` e_1 `in` e_2 may equivalently be written `let` v `=` e_1 `in` e_2.

- In a *Letrec* (p. 8) expression `letrec` $D_1 \cdots D_n$ `in` e, it is a compile-time error if the same *FunctionName* a/i occurs on the left-hand side of two function definitions D_j, D_k, $j \neq k$, in $D_1 \cdots D_n$.

- In a *Case* (p. 7) expression `case` e `of` $\mathit{Clause}_1 \cdots \mathit{Clause}_n$ `end`, it is a compile-time error if not all clauses of the expression have the same number of patterns (*cf.* section 4.6).

- In a *Receive* (p. 8) expression, on the general form:

  `receive` $\mathit{Clause}_1 \cdots \mathit{Clause}_n$ `after` e_1 `->` e_2 `end`

  it is a compile-time error if some clause of the expression does not have exactly one pattern (*cf.* section 4.6).

- A *Protected* (p. 8) expression `protected` e may only occur as part of a clause guard (*cf.* p. 16).

4.6 Clauses and patterns

A *Clause* (p. 7) has the general form `<`p_1`, ...,` p_n`> when` e_1 `->` e_2, where e_1 is known as the *guard* and e_2 as the *body* of the clause. e_2 is any expression, whereas e_1 is a restricted expression that must be valid as a Core Erlang clause guard (see section 5.6 for details). If n is 1, the clause can equivalently be written p_1 `when` e_1 `->` e_2.

Each p_i is a *Pattern* (p. 6) consisting of variables, atomic literals, tuple and list constructors, and alias patterns. *No variable name may occur more than once in the patterns* $p_1, \ldots, p_n$ *of a clause.* Pattern matching is described in section 5.4.

[2] Usually called *cons*.

5 Dynamic semantics

Core Erlang is a higher-order functional language operating on the same data types as Erlang. As in Erlang, functions are identified by the pair of the identifier *and* the arity. However, while in Erlang a function call evaluates to a single value, in Core Erlang the result of evaluating an expression is an *ordered sequence*, written `<`x_1`, ..., `x_n`>`, of zero, one or more values x_i. A sequence is not in itself a value; thus it is not possible to create sequences of sequences. For simplicity we denote any single-value sequence `<`x`>` by x where no confusion can ensue. If an expression e always evaluates to a sequence of values `<`x_1`, ..., `x_n`>`, then we define the *degree* of e to be the length n of this sequence.

An *environment* ρ is a mapping from names to Erlang values; *e. g.*, $\rho =$ [$v \mapsto$ `'true'`] maps the single variable name v to the atom `'true'`. We write $\rho_1\rho_2$ to denote the extension of ρ_1 by the elements of ρ_2, such that if $v \mapsto x$ is in ρ_1 and $v \mapsto y$ is in ρ_2, then only the latter is in $\rho_1\rho_2$. To simplify the presentation, in the context of environments all names are assumed to be without annotations.

5.1 Programs and processes

A Core Erlang *program* consists of an unordered set of definitions of distinctly named modules (*cf. Module*, p. 4). Execution of a program is performed by evaluating an initial expression `call` a_1`:`a_2`(`x_1`, ..., `x_n`)`, where a_1 and a_2 are atoms and $x_1, \ldots, x_n$ are any values (*cf. InterModuleCall*, p. 14), in an empty environment. The program execution ends when the evaluation of the initial call is completed, either normally, yielding a final result (the interpretation of which is implementation-dependent), or abruptly, by causing an exception to be raised that is not caught by a *Try* expression (*cf.* p. 15) in the program.

Each particular instance of a program execution is associated with some specific *process*. We define a process to be an object with a unique *identity* and a mutable *state*. The state of a process is assumed to contain a *mailbox* object, but otherwise its details are implementation-dependent. A mailbox is an ordered sequence of values, such that its contents may be inspected, a value may be appended to the sequence, and any value (at any position) may be removed from the sequence; no other operations are allowed. The state of a process, including the mailbox, may be mutated at any point during its lifetime, as a side effect of program execution or by other causes; this is also implementation-dependent.

The set of module definitions constituting the program is mutable, and at any time, module definitions may be added, removed or replaced,[3] maintaining the invariant that each module definition in the set is distinctly named. A definition m with name a in the set at any time, is generally referred to as the *latest version* of a at that time.

[3]Variously known as "dynamic code replacement", "run-time code replacement", and "hot code loading".

5.2 Evaluation of expressions

Argument evaluation in CORE ERLANG is *strict*, *i. e.*, all arguments to an operator are completely evaluated before the evaluation of the operator begins; furthermore, the evaluation order of arguments is *always undefined*, unless otherwise stated (notably in *Let* expressions, *Case* expressions, *Receive* expressions and *Try* expressions). The degree of any expression used as argument to another is unless otherwise stated expected to be 1 (one); if the degree of an expression does not match its use, the behaviour is undefined.

Every expression is assumed to be evaluated in a given environment ρ, mapping all free variables and function names in the expression to ERLANG values.

Expression evaluation can either terminate normally, yielding a sequence of values, or *abruptly*, by raising an exception. An *exception* is an object having two components called the *tag* and the *value*; both components must be ERLANG values. Except for *Try* expressions (see p. 15), *Protected* expressions (p. 16) and *Catch* expressions (p. 17), if the evaluation of an immediate subexpression e' of some expression e terminates abruptly with exception x, then evaluation of e also terminates abruptly with exception x.

ValueList:

> `<`s_1`,  ...,  `s_n`>`
>
> where each s_i is a *SingleExpression*, which must have degree 1.
>
> This evaluates to the sequence `<`x_1`,  ...,  `x_n`>` where for $i \in [1,n]$, s_i evaluates to x_i. The degree of the *ValueList* expression is thus n.

AtomicLiteral:

> This evaluates to the ERLANG value denoted by the literal. *Nil* (p. 5) denotes the empty list, which is a unique constant whose type is distinct from all other constants; it is thus not *e. g.* an atom. *Char* literals (p. 4) may be interpreted as denoting integer values representing character codes, but implementations may instead support a distinct character type.

VariableName:

> This evaluates to the value to which the *VariableName* v is bound in the environment ρ, that is, the value of $\rho(v)$.

FunctionName:

> a/i
>
> This evaluates to the *closure* to which the *FunctionName* a/i is bound in the environment ρ, that is, the value of $\rho(a/i)$. See also *Application* (p. 14) and *Fun* (p. 13).

Tuple:

> `{`e_1`,  ...,  `e_n`}`
>
> This evaluates to the ERLANG n-tuple $\{x_1,\ \ldots,\ x_n\}$, where for $i \in [1,n]$, e_i evaluates to x_i. Note that a 1-tuple $\{x\}$ is distinct from the value x, and that the 0-tuple $\{\}$ is a unique value.

List:

> `[`e_1 `|` e_2`]`

This evaluates to the ERLANG list constructor `[`x_1 `|` x_2`]`, where for $i \in [1, 2]$, e_i evaluates to x_i. See section 4.4 for details on list notation.

Let:

> `let <`v_1`, ..., `v_n`> = `e_1` in `e_2

e_1 is evaluated in the environment ρ, yielding a sequence `<`x_1`, ..., `x_n`>` of values. e_2 is then evaluated in the environment $\rho[v_1 \mapsto x_1, \ldots, v_n \mapsto x_n]$. e_1 must be completely evaluated before evaluation of e_2 begins, unless interleaving their evaluation yields no observable difference. The result is that of e_2 if evaluation of both expressions completes normally.

Note that if for all $i \in [1, n]$, v_i is not used in e_2, the expression is effectively a sequencing operator (*cf. Sequencing*, p. 17), evaluating e_1 before e_2 but discarding its value.

If e_1 does not have degree n, the behaviour is undefined.

Case:

> `case `e` of `P_1` when `g_1` -> `$b_1 \cdots P_n$` when `g_n` -> `b_n` end`

where each P_i, $i \in [1,n]$, is a sequence `<`p_{i1}`, ..., `p_{ik}`>` of patterns, for some fixed k (*cf.* p. 10).

The switch expression e is first evaluated in the environment ρ. If this succeeds, yielding a sequence `<`x_1`, ..., `x_k`>` of values, that sequence is then tried against the clauses of the *Case* in environment ρ as described in section 5.5.

If clause selection succeeds with selected clause j and mapping ρ' as result, the body b_j is evaluated in the environment $\rho\rho'$, and the result of that evaluation is the result of the *Case* expression.

If no clause can be selected, or if e does not have degree k, the behaviour is undefined.

Fun:

> `fun (`v_1`, ..., `v_n`) -> `e

This evaluates to the *closure*[4] defined by abstracting the expression e with respect to the parameters $v_1, \ldots, v_n$ in the environment ρ; see also *Application* (p. 14).

Letrec:

> `letrec `a_1`/`i_1` = `$f_1 \cdots a_n$`/`i_n` = `f_n` in `e

[4]A closure is defined as the pair consisting of: a) the program code of the function, and b) the environment in which it should be evaluated.

where for $k \in [1, n]$, each a_k/i_k is a *FunctionName* and each f_k a *Fun*.

The result of evaluating the *Letrec* in environment ρ is the result of evaluating expression e in the environment ρ', which is the smallest environment such that:

- for each x in the domain of ρ, *except* $x \in \{a_1/i_1, \ldots, a_n/i_n\}$, $\rho'(x)$ is equal to $\rho(x)$
- for each $a_k/i_k \in \{a_1/i_1, \ldots, a_n/i_n\}$, $\rho'(a_k/i_k)$ is equal to the result of evaluating the corresponding *Fun* expression f_k in the environment ρ' itself.

(Note that this definition of ρ' is circular; however, also note that only *Fun* expressions can be bound by a *Letrec*.)

Application:

`apply` e_0`(`e_1`, ...,` e_n`)`

where e_0 evaluates to a closure f (*cf. Fun*, p. 13).

All of $e_0, e_1, \ldots, e_n$ are evaluated in the environment ρ. Assume that $e_1, \ldots, e_n$ evaluate to values $x_1, \ldots, x_n$, respectively, and that f is the result of evaluating an expression `fun (`v_1`, ...,` v_k`) ->` e' in an environment ρ'. Evaluation of the application is then performed by evaluating e' in the environment $\rho'[v_1 \mapsto x_1, \ldots, v_n \mapsto x_n]$, if $n = k$.

If e_0 does not evaluate to a closure, or if the number n of arguments in the application is not equal to the arity k of f, the behaviour is implementation-dependent.

If the code defining the function of the closure is no longer available[5] at the time of evaluation of the application, the behaviour is implementation-dependent.

InterModuleCall:

`call` e'_1`:`e'_2`(`e_1`, ...,` e_n`)`

where e'_1 and e'_2 evaluate to atoms a_1 and a_2, respectively.

All of e'_1, e'_2 and $e_1, \ldots, e_n$ are evaluated in the environment ρ. Let m be the latest version of the module named by a_1 at the time of evaluation of the *InterModuleCall* expression. If the *ModuleBody* $D_1 \cdots D_k$ of m (*cf.* p. 5) contains a *FunctionDefinition* defining the name a_2/n, and a_2/n is also in the *Exports* declaration of m, then let the closure f be the result of evaluating the expression `letrec` $D_1 \cdots D_k$ `in` a_2/n, in the empty environment.[6] The *InterModuleCall* expression is then equivalent to an *Application* `apply` f`(`e_1`, ...,` e_n`)`.

[5]An implementation could use a garbage collection scheme to safely remove unused code. Another strategy, used by current ERLANG implementations, is to force the removal of code which has been superseded twice by a newer version. This so-called *purging* of code might however be unsafe, unless extra runtime checks are done.

[6]The domain of the environment of such a closure is simply the function names defined by the module, and it is therefore not necessary to represent the closure explicitly.

If a_2/n is not defined and exported by m, the behaviour of the inter-module call expression is implementation-dependent.

If e'_1 and e'_2 do not both evaluate to atoms, the behaviour is implementation-dependent.

PrimOpCall:

> `primop` a`(`e_1`, ...,` e_n`)`

$e_1, \ldots, e_n$ are evaluated in the environment ρ to values $x_1, \ldots, x_n$, respectively. The primitive operation to be performed is identified by the name a and the number n of arguments (its arity).

Evaluation of a *PrimOpCall* is always implementation-dependent and may depend on the values $x_1, \ldots, x_n$, the state of the associated process (*e. g.* the mailbox), or the external state (*i. e.*, the world). The evaluation may have side effects, and may complete abruptly by raising an exception (*cf.* *Try*, below).

Try:

> `try` e_1 `catch (`v_1`,` v_2`) ->` e_2

e_1 is evaluated in the environment ρ, and if that evaluation completes normally, its result is also the result of the *Try* expression; otherwise, if evaluation of e_1 completes abruptly with exception x, e_2 is evaluated in the environment $\rho[v_1 \mapsto t, v_2 \mapsto u]$, where t is the *tag* and u the *value* of x (*cf.* p. 12), and the result of that evaluation becomes the result of the *Try* expression.

CORE ERLANG defines no specific way of raising exceptions, but given a primitive operation named *e. g.* `raise`, of arity 2, with the effect of always terminating abruptly by raising an exception whose corresponding tag and value are the actual parameters to the call, we could define:

```
 erlang:exit(R)  ::=  primop 'raise'('EXIT', R)
erlang:throw(R)  ::=  primop 'raise'('THROW', R)
```

for the ERLANG built-in standard functions `exit/1` and `throw/1`.

Receive:

> `receive <`p_1`> when` g_1 `->` $b_1 \cdots$ `<`p_n`> when` g_n `->` b_n
> `after` e_1 `->` e_2 `end`

Evaluation of a *Receive* is divided into stages, as follows:

1. First, the expiry expression e_1 is evaluated to a value t in the environment ρ. t must be either a nonnegative integer or the atom `'infinity'`, otherwise the behaviour is implementation-dependent.
2. Next, each value in the mailbox (of the associated process), in first-to-last order, is tried one at a time against the clauses `<`p_1`> when` g_1 `->` $b_1 \cdots$ `<`p_n`> when` g_n `->` b_n in the environment ρ, as described in section 5.5, until one of the following occurs:

- If for some value M_k at position k in the mailbox and some $i \in [1, n]$, clause selection succeeds yielding a selected clause i and a mapping ρ', then the element at position k is first deleted from the mailbox, and expression b_i is evaluated in environment $\rho\rho'$ to yield the value of the *Receive*.
- If there are no remaining values to be tried in the mailbox, then either if t is the integer `0` (zero), or t is a positive integer and t or more milliseconds have passed since the transition from stage 1 to stage 2 was made, the expiry body e_2 is evaluated in environment ρ to yield the value of the *Receive*; otherwise stage 3 is entered.

3. The evaluation of the *Receive* is at this point suspended, and may be resumed when either or both of the following has occurred:

   - One or more values have been appended to the mailbox.
   - t or more milliseconds have passed since the transition from stage 1 to stage 2 was made, when t is a positive integer.

   The evaluation then again enters stage 2, where this time only those values in the mailbox (if any) should be tried that have not been tried since the latest transition from stage 1 to stage 2 was made. (Note that any subsequent *Receive* will thus start over from the first value in the mailbox, and not continue where any previous *Receive* finished.)

A Receive may never be evaluated as part of the evaluation of a clause guard of another Receive. The removal of a message from the mailbox is a *side effect*, and this is not allowed in a guard. Even more importantly, two *Receive* expressions being evaluated in a nested fashion using the same mailbox could want to select and remove the same message, and it is not obvious how such conflicts could be resolved in a consistent way useful to the programmer. Another, lesser complication would be that the evaluation would have to be able to track nested timeouts to any depth.

Because the timeout limit t (when t is a positive integer) is *soft*, *i. e.*, a lower bound only, an implementation is free to allow any number of values to be appended to the mailbox while evaluation is suspended in stage 3, even after the timeout limit has expired. However, implementations should in general attempt to detect timeouts as soon as possible.

It can be noted that it is quite possible for an implementation to signal timeouts by simply appending a unique value, associated with a particular active *Receive*, to the corresponding mailbox, causing the second wake-up condition of stage 3 to be subsumed by the first. However, unselected timeout messages will then need to be garbage collected from the mailboxes in order to prevent cases of unbounded growth.

Protected:

```
protected e
```

A *Protected* expression may only occur as part of a clause guard. e is evaluated in the environment ρ, and if that evaluation completes normally, its result is also the result of the *Protected* expression; otherwise, if evaluation of e completes abruptly, the *Protected* expression evaluates to the constant 'false'.

Outside of clause guards, this behaviour can be simulated by the expression

```
try e catch (v1, v2) -> 'false'
```

where v_1 and v_2 are distinct variables (*cf.* *Try*, p. 15).

5.3 Standard syntactic sugar

This section describes CORE ERLANG expressions that are defined in terms of the primitives that have been described above, but which are nevertheless included in the language for convenience (usually referred to as "syntactic sugar".)

Sequencing:

```
do e1 e2
```

This is equivalent to `let <`v_1`, ..., `v_n`> = `e_1` in `e_2, where n is the degree of e_1, and the variables $v_1, \ldots, v_n$ do not occur free in e_2. Thus, e_1 is evaluated before e_2, but its result is not used (*cf.* *Let*, p. 13).

Catch:

```
catch e
```

This is equivalent to

```
try e
catch (v1, v2) ->
    case v1 of
      'THROW' when 'true' -> v2
      v3 when 'true' -> {v1, v2}
    end
```

where v_1 and v_2 are distinct variables (*cf.* *Try*, p. 15), and v_3 is not the same variable as v_2. This encodes the behaviour of ERLANG `catch` expressions.

5.4 Pattern matching

Pattern matching recursively matches the structures of a sequence of values $x_1, \ldots, x_n$ against a corresponding sequence of patterns (*cf.* p. 6) $p_1, \ldots, p_n$, either succeeding, yielding as result a mapping from the variables in $p_1, \ldots, p_n$ to subterms of $x_1, \ldots, x_n$, or otherwise failing. No variable name may occur more than once in the sequence of patterns.

- A sequence of patterns $p_1, \ldots, p_n$ matches a sequence of values $x_1, \ldots, x_n$, yielding the mapping $\rho_1 \cdots \rho_n$, if and only if for all $i \in [1, n]$, p_i matches x_i yielding the mapping ρ_i.

- An *AtomicLiteral* pattern p matches a value x, yielding the empty mapping $[\,]$, if and only if p denotes x.
- A *VariableName* pattern p *always* matches a value x, yielding the mapping $[p \mapsto x]$.
- A tuple pattern `{`p_1`,` ...`,` p_n`}` matches a value `{`x_1`,` ...`,` x_n`}`, yielding the mapping $\rho_1 \cdots \rho_n$, if and only if for all $i \in [1, n]$, p_i matches x_i yielding the mapping ρ_i.
- A list constructor pattern `[`p_1 `|` p_2`]` matches a value `[`x_1 `|` x_2`]`, yielding the mapping $\rho_1\rho_2$, if and only if for $i \in [1, 2]$, p_i matches x_i yielding the mapping ρ_i.
- An alias pattern v `=` p matches a value x, yielding the mapping $\rho[v \mapsto x]$, if and only if p matches x yielding the mapping ρ.

5.5 Clause selection

First, recall that in Core Erlang, a variable may occur at most once in the patterns of a single clause, and note that pattern variables are always binding occurrences; variables cannot be repeated or imported in patterns, as they can in Erlang.

Given a sequence $x_1, \ldots, x_k$ of switch values and an environment ρ, a sequence of clauses

$$P_1 \texttt{ when } g_1 \texttt{ -> } b_1 \cdots P_n \texttt{ when } g_n \texttt{ -> } b_n$$

where each P_i is a sequence `<`p_{i1}`,` ...`,` p_{ik}`>` of patterns, is tried in left-to-right order as follows:

If the pattern sequence $p_{i1}, \ldots, p_{ik}$ is matched successfully against $x_1, \ldots, x_k$, yielding a mapping $\rho' = [v_1 \mapsto x'_1, \ldots, v_m \mapsto x'_m]$, where $v_1, \ldots, v_m$ are exactly the variables occurring in $p_{i1}, \ldots, p_{ik}$, each bound in ρ' to some subterm x'_j of $x_1, \ldots, x_k$ as the result of the pattern matching (*cf.* section. 5.4 for details), then the expression `protected` g_i is evaluated in the environment $\rho\rho'$. If the result is `'true'`, clause selection succeeds, yielding the selected clause i and mapping ρ' as result. If the result is `'false'`, the next clause in order is tried; if no clause remains, clause selection fails.

5.6 Clause guards

A Core Erlang clause guard must not have observable side effects and should evaluate in bounded (preferably constant or linear) time. Because all guards are evaluated within a `protected` expression, if the evaluation of a guard does not complete normally, the raised exception is implicitly caught and discarded, and the value `'false'` is used for the result, thus failing the clause quietly.[7] If a clause guard evaluates to a value other than `'true'` or `'false'`, the behaviour is undefined.

At present, Core Erlang clause guards and all their subexpressions are restricted to *Protected* together with the following subset of expressions:

[7]It is the explicit intention that a compiler might utilise a more efficient error handling mechanism in the restricted case of guard evaluation.

- *AtomicLiteral*
- *VariableName* and *FunctionName*
- *Tuple*
- *List*
- *Let* (and thus also *Sequencing*)
- *PrimOpCall*
- *InterModuleCall*

where for any *PrimOpCall* `primop` a`(`e_1`, ..., `e_n`)`, the primitive operation a/n must not have observable side effects, and for any *InterModuleCall* `call` e'_1`:`e'_2`(`e_1`, ..., `e_n`)`, e'_1 and e'_2 must be atom literals such that the function named e'_2/n in module e'_1 is trusted to exist and not have observable side effects.[8] The set of trusted functions and primitive operations is implementation-dependent; in implementations of ERLANG, it typically includes those so called built-in functions (BIFs) that are classified as "guard BIFs", and type tests; see the ERLANG Reference Manual [1] for details.

6 Open issues

This section discusses known issues that may warrant further specification in future versions of this document.

6.1 Source code portability

Because several details of the semantics of CORE ERLANG have been defined as implementation-dependent, it is possible for an implementation to expect a particular behaviour for each of those details. (Typical examples of expected behaviour could be that an exception on a particular form is raised, or that an attempt is made to load missing code.) Therefore, CORE ERLANG code generated by one implementation (*e. g.*, by translation from ERLANG source code) might not be suitable as input to another implementation that makes different assumptions.

At present, there is no canonical translation from ERLANG to CORE ERLANG, which preserves the semantics of the ERLANG program while making as few assumptions as possible about implementation-dependent behaviour in CORE ERLANG. Furthermore, the naming conventions for the ERLANG operators and type tests (*cf.* section 6.3) need to be standardised in order to create a canonical translation.

[8] Not all such "remote" functions must have actual implementations in existing modules, but may instead be aliases for built-in operations known to the compiler.

6.2 Syntax for binary objects

The latest addition to the Erlang language, as of this writing, is a compact syntax for composition and decomposition of binary-type objects. This can easily be expanded into primitive operations during translation from Erlang to Core Erlang, but it is not yet clear whether it is actually preferable that the expansion is performed at that level, or whether a similar syntax should be added to the Core Erlang language in order to facilitate optimisations on such expressions.

6.3 Built-in functions

The Erlang language specifies a large number of so-called built-in functions (BIFs), together with a set of unary and binary operators, and boolean type test operations that are recognized only in clause guards. Most BIFs, but not all, currently belong to the `erlang` module. Some BIFs may be used in clause guard expressions. Some BIFs are recognised by the compiler as if implicitly declared as imported, thus not needing to be qualified by their module names.

BIFs are predefined functions supplied with the implementation, but do not have to be implemented in any particular way – they can be inline-expanded by the compiler, implemented in another language such as C, or be regular Erlang functions. The only requirement on a BIF is that it “must not be redefined during the lifetime of a node” [1], which makes it possible for an implementation to “use all information in the description of the BIF to make execution efficient”.

All BIFs have a “home module”, making it possible to dynamically call also those BIFs that are not implemented as regular Erlang functions by their module and function names. The Erlang operators and type tests, however, do not at present have corresponding documented home module and function names, but this will be added in future releases.

The Core Erlang representation of an explicit call to an Erlang BIF that is not a regular Erlang function can therefore be either of `call` a_1`:`a_2`(`...`)`, where a_1 and a_2 are atom literals, or `primop` a`(`...`)`, where a is an atom literal. In the former case, it is then assumed that the compiler will recognise the call as an alias for a built-in operation and eventually generate appropriate code for making the call, possibly by first rewriting it as a `primop` call. The names and semantics of `primop` operations are however always implementation-dependent, and it can be expected that programs operating on Core Erlang code will be more portable if the form `call` a_1`:`a_2`(`...`)` is used and retained for as long as possible in the compilation process.

In order to extend the portability of programs that operate on Core Erlang code, it will be necessary to parameterise information about built-in functions. Because the Erlang language keeps evolving, and because different Erlang implementations may not have the exact same sets of predefined functions, it is generally not a good idea to hard-code assumptions about BIFs. Instead, such information should in as much as possible be moved to separate modules, so that when porting a Core Erlang analysis or transformation from one Erlang implementation to another, only these modules need rewriting. It is then possible that a set of standard modules for BIF information could be agreed on, which could be assumed to be supplied by every Erlang implementation.

References

[1] J. Barklund, R. Virding, ERLANG *4.7.3 Reference Manual*, draft version 0.7. June 1999.

[2] J. Barklund, R. Virding, *Specification of the* STANDARD ERLANG *programming language*, draft version 0.7. June 1999.

[3] ISO/IEC, *Information processing – 8-bit single-byte coded graphic character sets*, 1987. Reference number ISO 8879:1987.

[4] The Unicode Consortium, *The Unicode Standard, Version 2.0.* Addison-Wesley, Reading, Mass., 1996.

A Quick reference

This section gives an informal overview of the elements of the language.

A.1 Comments

Example:

```
% This is a comment; it ends just before the line break.
```

A.2 Constant literals

Examples:

Integers: `8`, `+17`, `299792458`, `-4711`

Floating-point numbers: `0.0`, `2.7182818`, `-3.14`, `+1.2E-6`, `-1.23e12`, `1.0e+9`

Atoms: `'foo'`, `'Bar'`, `'foo bar'`, `''`, `'%#\010@\n!'`, `'_hello_world'`

Character literals: `$A`, `$$`, `$\n`, `$\s`, `$\\`, `$\12`, `$\101`, `$\^A`

Strings: `"Hello, world!"`, `"Two\nlines"`, `""`, `"Ring\^G" "My\7" "Bell\007!"`

A.3 Variables

Examples:

`X`, `Bar`, `Value_2`, `One2Three`, `Stay@home`, `_hello_world`

A.4 Keywords

after	apply	attributes	call	case
catch	do	end	fun	in
let	letrec	module	of	primop
protected	receive	try	when	

A.5 Separators

(	)	{	}	[	]	<	>
,	:	|	/	=	->	-|	

A.6 Annotations

`( • -| [` $const_1$ `,` ..., $const_n$ `] )`

A.7 Programs and expressions

$$
\begin{array}{rcl}
\mathit{module} & ::= & \texttt{module}\ \mathit{Atom}\ \texttt{[}\mathit{fname}_{i_1}\texttt{,}\ \ldots\texttt{,}\ \mathit{fname}_{i_k}\texttt{]} \\
 & & \texttt{attributes [}\mathit{Atom}_1\ \texttt{=}\ \mathit{const}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{Atom}_m\ \texttt{=}\ \mathit{const}_m\ \texttt{]} \\
 & & \mathit{fname}_1\ \texttt{=}\ \mathit{fun}_1 \cdots \mathit{fname}_n\ \texttt{=}\ \mathit{fun}_n\ \texttt{end} \\
\mathit{fname} & ::= & \mathit{Atom}\ \texttt{/}\ \mathit{Integer} \\
\mathit{const} & ::= & \mathit{lit} \mid \texttt{[}\mathit{const}_1\ \texttt{|}\ \mathit{const}_2\texttt{]} \mid \texttt{\{}\mathit{const}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{const}_n\texttt{\}} \\
\mathit{lit} & ::= & \mathit{Integer} \mid \mathit{Float} \mid \mathit{Atom} \\
 & \mid & \mathit{Char} \mid \mathit{String} \mid \texttt{[]} \\
\mathit{fun} & ::= & \texttt{fun (}\mathit{var}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{var}_n\texttt{) ->}\ \mathit{vals} \\
\mathit{var} & ::= & \mathit{VariableName} \\
\mathit{vals} & ::= & \mathit{expr} \mid \texttt{<}\mathit{expr}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{expr}_n\texttt{>} \\
\mathit{expr} & ::= & \mathit{var} \mid \mathit{fname} \mid \mathit{lit} \mid \mathit{fun} \\
 & \mid & \texttt{[}\mathit{vals}_1\ \texttt{|}\ \mathit{vals}_2\texttt{]} \mid \texttt{\{}\mathit{vals}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{vals}_n\texttt{\}} \\
 & \mid & \texttt{let}\ \mathit{vars}\ \texttt{=}\ \mathit{vals}_1\ \texttt{in}\ \mathit{vals}_2 \\
 & \mid & \texttt{case}\ \mathit{vals}\ \texttt{of}\ \mathit{clause}_1 \cdots \mathit{clause}_n\ \texttt{end} \\
 & \mid & \texttt{letrec}\ \mathit{fname}_1\ \texttt{=}\ \mathit{fun}_1 \cdots \mathit{fname}_n\ \texttt{=}\ \mathit{fun}_n\ \texttt{in}\ \mathit{vals} \\
 & \mid & \texttt{apply}\ \mathit{vals}_0\texttt{(}\mathit{vals}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{vals}_n\texttt{)} \\
 & \mid & \texttt{call}\ \mathit{vals}'_1\texttt{:}\mathit{vals}'_2\texttt{(}\mathit{vals}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{vals}_n\texttt{)} \\
 & \mid & \texttt{primop}\ \mathit{Atom}\texttt{(}\mathit{vals}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{vals}_n\texttt{)} \\
 & \mid & \texttt{try}\ \mathit{vals}_1\ \texttt{catch (}\mathit{var}_1\texttt{,}\ \mathit{var}_2\texttt{) ->}\ \mathit{vals}_2 \\
 & \mid & \texttt{receive}\ \mathit{clause}_1 \cdots \mathit{clause}_n\ \texttt{after}\ \mathit{vals}_1\ \texttt{->}\ \mathit{vals}_2\ \texttt{end} \\
 & \mid & \texttt{protected}\ \mathit{vals} \\
 & \mid & \texttt{do}\ \mathit{vals}_1\ \mathit{vals}_2 \\
 & \mid & \texttt{catch}\ \mathit{vals} \\
\mathit{vars} & ::= & \mathit{var} \mid \texttt{<}\mathit{var}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{var}_n\texttt{>} \\
\mathit{clause} & ::= & \mathit{pats}\ \texttt{when}\ \mathit{vals}_1\ \texttt{->}\ \mathit{vals}_2 \\
\mathit{pats} & ::= & \mathit{pat} \mid \texttt{<}\mathit{pat}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{pat}_n\texttt{>} \\
\mathit{pat} & ::= & \mathit{var} \mid \mathit{lit} \mid \texttt{[}\mathit{pat}_1\ \texttt{|}\ \mathit{pat}_2\texttt{]} \mid \texttt{\{}\mathit{pat}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{pat}_n\texttt{\}} \\
 & \mid & \mathit{var}\,\texttt{=}\,\mathit{pat}
\end{array}
$$

B Escape sequences

This table shows the Unicode character values for the escape sequences defined by Core Erlang; they are the same as in Erlang.

`\b`	`\u0008`	(backspace, `BS`)
`\d`	`\u007f`	(delete, `DEL`)
`\e`	`\u001b`	(escape, `ESC`)
`\f`	`\u000c`	(form feed, `FF`)
`\n`	`\u000a`	(linefeed, `LF`)
`\r`	`\u000d`	(carriage return, `CR`)
`\s`	`\u0020`	(space, `SPA`)
`\t`	`\u0009`	(horizontal tab, `HT`)
`\v`	`\u000b`	(vertical tab, `VT`)
`\"`	`\u0022`	(double quote, `"`)
`\'`	`\u0027`	(apostrophe/single quote, `'`)
`\\`	`\u005c`	(backslash, `\`)

C Syntax tree representation

The following schema describes a representation of CORE ERLANG syntax trees, suitable for most general purposes. For brevity, define x^i to mean an ordered sequence $(x_1, \ldots, x_i)$, for $i \geq 0$, and x^* to mean any sequence in $\bigcup_{i=0}^{\infty}\{x^i\}$.

$$
\begin{array}{rcl}
\mathit{module} & ::= & \texttt{module}\,(\texttt{atom}\,a)\ \mathit{fname}^*\ \mathit{attr}^*\ \mathit{def}^{\,*} \\
\mathit{fname} & ::= & \texttt{fname}\,a\ i \\
\mathit{attr} & ::= & (\texttt{atom}\,a,\ \mathit{const}) \\
\mathit{const} & ::= & \mathit{lit}\ \mathit{const}^* \\
\mathit{lit} & ::= & \texttt{int}\,i \ \mid\ \texttt{float}\,f \\
 & \mid & \texttt{atom}\,a \ \mid\ \texttt{char}\,c \\
 & \mid & \texttt{nil} \ \mid\ \texttt{cons} \ \mid\ \texttt{tuple} \\
\mathit{def} & ::= & (\mathit{fname},\ \mathit{fun}) \\
\mathit{fun} & ::= & \texttt{fun}\,v^*\ w \\
v & ::= & \texttt{var}\,s \\
w & ::= & e \ \mid\ \texttt{values}\,e^* \\
e & ::= & v \ \mid\ \mathit{fname} \ \mid\ \mathit{lit}\ w^* \ \mid\ \mathit{fun} \\
 & \mid & \texttt{let}\,v^*\ w\ w \\
 & \mid & \texttt{case}\,w\ \mathit{clause}^* \\
 & \mid & \texttt{letrec}\,\mathit{def}^{\,*}\ w \\
 & \mid & \texttt{apply}\,w\ w^* \\
 & \mid & \texttt{call}\,w\ w\ w^* \\
 & \mid & \texttt{primop}\,a\ w^* \\
 & \mid & \texttt{try}\,w\ v^*\ w \\
 & \mid & \texttt{receive}\,\mathit{clause}^*\ w\ w \\
 & \mid & \texttt{protected}\,w \\
 & \mid & \texttt{do}\,w\ w \\
 & \mid & \texttt{catch}\,w \\
\mathit{clause} & ::= & \texttt{clause}\,\mathit{pat}^*\ w\ w \\
\mathit{pat} & ::= & v \ \mid\ \mathit{lit}\ \mathit{pat}^* \ \mid\ \texttt{alias}\,v\ \mathit{pat}
\end{array}
$$

where a stands for an atom, i for an integer, f for a floating-point number, c for a character and s for a string.

Implicitly, each constructor above should also have an additional field for an associated list of annotations $(\texttt{atom}\,a,\ \mathit{const})$.

Index