lfe_guide(7)

Robert Virding

2008-2020

NAME

lfe_guide - Lisp Flavoured Erlang User Guide

SYNOPSIS

Note: {{ … }} is used to denote optional syntax.

LITERALS AND SPECIAL SYNTACTIC RULES

Integers

Integers can be written in various forms and number bases:

- Regular decimal notation:

```
1234 -123 0
```

- Binary notation:

```
#b0 #b10101 #b-1100
```

- Binary notation (alternative form):

```
#*0 #*10101 #*-1100
```

- Octal notation:

```
#o377 #o-111
```

- Explicitly decimal notation:

```
#d1234 #d-123 #d0
```

- Hexadecimal notation:

```
#xc0ffe #x-01
```

- Notation with explicit base (up to 36):

```
#2r1010 #8r377 #36rhelloworld
```

- Character notation (the value is the Unicode code point of the character):

```
#\a #\$ #\ä
```

- Character notation with the value in hexadecimal:

```
#\x1f42d;
```

In all these forms, the case of the indicating letter is not significant, i.e. `#b1010` and `#B1010`

are identical as are `#16rf00` and `#16Rf00`.

Similarly, the case is not significant for digits beyond 9 (i.e. ‘a’, ‘b’, ‘c’, … for number bases larger than 10), e.g. `#xabcd` is the same as `#xABCD` and can even be mixed in the same number, e.g. `#36rHelloWorld` is valid and the same number as `#36Rhelloworld` and `#36rHELLOWORLD`.

The character notation using hexadecimal code representation (`#\x...;`) is basically the same thing as the regular hexadecimal notation `#x...` except that it conveys to the reader that a character is intended and that it does a sanity check on the value (e.g. negative numbers and value outside the Unicode range are not permitted).

Floating point numbers

There is only one type of floating point numbers and the literals are written in the usual way, e.g. these are all valid floating point numbers:

```
1.0 +1.0 -1.0 1.0e10 1.111e-10
```

The one thing to watch out for is that you cannot omit the part before or after the decimal point if it is zero. E.g. the following are not valid forms: `100.` or `.125`.

Strings

There are two forms of strings: list strings and binary strings.

List Strings

List strings are just lists of integers (where the values have to be from a certain set of numbers that are considered valid characters) but they have their own syntax for literals (which will also be used for integer lists as an output representation if the list contents looks like it is meant to be a string): “any text between double quotes where " and other special characters like `\n` can be escaped”.

As a special case you can also write out the character number in the form `\xHHH;` (where “HHH” is an integer in hexadecimal notation), e.g. `"\x61;\x62;\x63;"` is a complicated way of writing `"abc"`. This can be convenient when writing Unicode letters not easily typeable or viewable with regular fonts. E.g. `"Cat: \\x1f639;"` might be easier to type (and view on output devices without a Unicode font) then typing the actual Unicode letter.

Binary Strings

Binary strings are just like list strings but they are represented differently in the virtual machine. The simple syntax is `#"..."`, e.g. `#"This is a binary string \n with some \"escaped\" and quoted (\\x1f639;) characters"`

You can also use the general format for creating binaries (`#B(...)`, described below), e.g. `#B("a")`, `#"a"`, and `#B(97)` are all the same binary string.

Character Escaping

Certain control characters can be more readably included by using their escaped name:

Escaped name	Character
\b	Backspace
\t	Tab
\n	Newline
\v	Vertical tab
\f	Form Feed
\r	Carriage Return
\e	Escape
\s	Space
\d	Delete

Alternatively you can also use the hexadecimal character encoding, e.g. `"a\nb"` and

`"a\x0a;b"` are the same string.

Triple-quoted Strings

LFE's triple-quote strings are modelled on those in Erlang so we will use their description of them.

Strings, both list strings and binary strings, can also be written as triple-quoted strings, which can be indented over multiple lines to follow the indentation of the surrounding code. They are also verbatim, that is, they do not allow escape sequences, and thereby do not need double quote characters to be escaped.

Example, with verbatim double quote characters:

```
"""
  Line "1"
  Line "2"
  """
```

That is equivalent to the normal single quoted string (which also allows newlines):

```
"Line \"1\"
Line \"2\""
```

The opening and the closing lines have the delimiters: the """ characters. The lines between them are the content lines. The newline on the opening line is not regarded as string content, nor is the newline on the last content line.

The indentation is defined by the white space character sequence preceding the delimiter on the closing line. That character sequence is stripped from all content lines. There can only be white space before the delimiter on the closing line, or else it is regarded as a content line.

The opening line is not allowed to have any characters other than white space after the delimiter, and all content lines must start with the defined indentation character sequence, otherwise the string has a syntax error.

Here is a larger example:

```
"""
      First line starting with two spaces
    Not escaped: "\t \r \xFF" and """

    """
```

That corresponds to the normal string:

```
"  First line starting with two spaces
Not escaped: \"\\t \\r \\xFF\" and \"\"\"
"
```

Binary strings can also be written as triple-quoted strings:

```
#"""
  Line "1"
  Line "2"
  """
```

which correspnds to the binary:

```
#"Line \"1\"\nLine \"2\""
```

Binaries

We have already seen binary strings, but the `#B(...)` syntax can be used to create binaries with any contents. Unless the contents is a simple integer you need to annotate it with a type and/or size.

Example invocations are that show the various annotations:

```
> #B(42 (42 (size 16)) (42 (size 32)))
#B(42 0 42 0 0 0 42)
> #B(-42 111 (-42 (size 16)) 111 (-42 (size 32)))
#B(-42 111 (-42 (size 16)) 111 (-42 (size 32)))
> #B((42 (size 32) big-endian) (42 (size 32) little-endian))
#B(0 0 0 42 42 0 0 0)
> #B((1.23 float) (1.23 (size 32) float) (1.23 (size 64) float))
#B(63 243 174 20 122 225 71 174 63 157 112 164 63 243 174 20
   122 225 71 174)
> #B((#"a" binary) (#"b" binary))
#"ab"
```

Learn more about “segments” of binary data e.g. in “ Learn You Some Erlang ” http://learnyousomeerlang.com/starting-out-for-real#bit-syntax.

Lists

Lists are formed either as `( ... )` or `[ ... ]` where the optional elements of the list are separated by some form or whitespace. For example:

```
()
(the empty list)
(foo bar baz)
(foo
 bar
 baz)
```

Tuples

Tuples are written as `#(value1 value2 ...)`. The empty tuple `#()` is also valid.

Maps

Maps are written as `#M(key1 value1 key2 value2 ...)` The empty map is also valid and written as `#M()`.

Structs

Structs are written as `#S(struct-name key1 value1 key2 value2 ...)`.

Note that structs cannot be created with the literal syntax, the `(struct mod-name ...)` form must be used.

Symbols

Things that cannot be parsed as any of the above are usually considered as a symbol.

Simple examples are `foo`, `Foo`, `foo-bar`, `:foo`. But also somewhat surprisingly `123foo` and `1.23e4extra` (but note that illegal digits don’t make a number a symbol when using the explicit number base notation, e.g. `#b10foo` gives an error).

Symbol names can contain a surprising breadth or characters, basically all of the latin-1 character set without control character, whitespace, the various brackets, double quotes and semicolon.

Of these, only `|`, `\'`, `'`, `,`, and `#` may not be the first character of the symbol’s name (but they *are* allowed as subsequent letters).

I.e. these are all legal symbols: `foo`, `foo`, `µ#`, `±1`, `451°F`.

Symbols can be explicitly constructed by wrapping their name in vertical bars, e.g. `|foo|`, `|symbol name with spaces|`. In this case the name can contain any character of in the range from 0 to 255 (or even none, i.e. `||` is a valid symbol). The vertical bar in the symbol name needs to be escaped: `|symbol with a vertical bar \| in its name|`

(similarly you will obviously have to escape the escape character as well).

Comments

Comments come in two forms: line comments and block comments.

Line comments start with a semicolon (`;`) and finish with the end of the line.

Block comments are written as `#| comment text |#` where the comment text may span multiple lines but my not contain another block comment, i.e. it may not contain the character sequence `#|`.

Evaluation While Reading

`#.(... some expression ...)`. E.g. `#.(+ 1 1)` will evaluate the `(+ 1 1)` while it reads the expression and then be effectively `2`.

Supported forms

Core forms

```
(quote e)
(cons head tail)
(car e)
(cdr e)
(list e ... )
(tuple e ... )
(tref tuple index)
(tset tuple index val)
(binary seg ... )
(map key val ...)
(map-size map) (msiz m)
(map-get map key) (mref m k)
(map-set map key val ...) (mset m k v ...)
(map-update map key val ...) (mupd m k v ...)
(map-remove map key ...) (mrem m k k ...)
(lambda (arg ...) ...)
(match-lambda
  ((arg ... ) {{(when e ...)}} ...)          - Matches clauses
  ... )
(function func-name arity)                   - Function reference
(function mod-name func-name arity)
(let ((pat {{(when e ...)}} e)
      ...)
  ... )
(let-function ((name lambda|match-lambda)    - Local functions
               ... )
  ... )
(letrec-function ((name lambda|match-lambda)  - Local functions
                  ... )
  ... )
(let-macro ((name lambda-match-lambda)       - Local macros
            ...)
  ...)
(progn ... )
(if test true-expr {{false-expr}})
(case e
  (pat {{(when e ...)}} ...)
  ... ))
(maybe
  ...
  {{else ...}})
(receive
  (pat {{(when e ...)}} ... )
  ...
  (after timeout ... ))
```

```
(catch ... )
(try
  e
  {{(case ((pat {{(when e ...)}} ... )
          ... ))}}
  {{(catch
     ((tuple type value stacktrace)|_ {{(when e ...)}}
                           - Must be tuple of length 3 or just _!
      ... )
     ... )}}
  {{(after ... )}})
(funcall func arg ... )
(call mod func arg ... )    - Call to Mod:Func(Arg, ... )

(define-record name fields)
(record name field val ...)
(is-record record name)
(record-index name field)
(record-field record name field)
(record-update record name field val ...)

(define-struct fields)
(struct mod-name field val ...)
(is-struct struct)
(is-struct struct name)
(struct-field struct name field)
(struct-update struct name field val ...)

(define-module name meta-data attributes)
(extend-module meta-data attributes)

(define-function name meta-data lambda|match-lambda)
(define-macro name meta-data lambda|match-lambda)

(define-type type definition)
(define-opaque-type type definition)
(define-function-spec func spec)
```

Basic macro forms

```
(: mod func arg ... ) =>
        (call 'mod 'func arg ... )
(mod:func arg ... ) =>
        (call 'mod 'func arg ... )
(? {{timeout {{default}} }})
(++ ... )
(-- ... )
(list* ... )
(let* (... ) ... )
(flet ((name (arg ...) {{doc-string}} ...)
       ...)
  ...)
(flet* (...) ... )
(fletrec ((name (arg ...) {{doc-string}} ...)
          ...)
  ...)
(cond (test body ...)
      ...
      ((?= pat expr) ...)
      ...
      (else ...))
(andalso ... )
(orelse ... )
(fun func arity)
(fun mod func arity)
(lc (qual ...) expr)
(list-comp (qual ...) expr)
(bc (qual ...) bitstringexpr)
(binary-comp (qual ...) bitstringexpr)
(ets-ms ...)
(trace-ms ...)
```

Common Lisp inspired macros

```
(defun name (arg ...) {{doc-string}} ...)
(defun name
  {{doc-string}}
  ((argpat ...) ...)
  ...)
(defmacro name (arg ...) {{doc-string}} ...)
(defmacro name arg {{doc-string}} ...)
(defmacro name
  {{doc-string}}
  ((argpat ...) ...)
  ...)
(defsyntax name
  (pat exp)
  ...)
(macrolet ((name (arg ...) {{doc-string}} ...)
           ...)
  ...)
(syntaxlet ((name (pat exp) ...)
            ...)
  ...)
(prog1 ...)
(prog2 ...)
(defmodule name ...)
(defrecord name ...)
(defstruct ...)
```

Patterns

Written as normal data expressions where symbols are variables and use quote to match explicit values. Binaries and tuples have special syntax.

```
{ok,X}                  -> (tuple 'ok x)
error                   -> 'error
{yes,[X|Xs]}            -> (tuple 'yes (cons x xs))
<<34,U:16,F/float>>     -> (binary 34 (u (size 16)) (f float))
[P|Ps]=All              -> (= (cons p ps) all)
```

Repeated variables are supported in patterns and there is an automatic comparison of values.

_ as the “don’t care” variable is supported. This means that the symbol _, which is a perfectly valid symbol, can never be bound through pattern matching.

Aliases are defined with the `(= pattern1 pattern2)` pattern. As in Erlang patterns they can be used anywhere in a pattern.

CAVEAT The lint pass of the compiler checks for aliases and if they are possible to match. If not an error is flagged. This is not the best way. Instead there should be a warning and the offending clause removed, but later passes of the compiler can’t handle this yet.

Guards

Wherever a pattern occurs (in let, case, receive, lc, etc.) it can be followed by an optional guard which has the form `(when test ...)`. Guard tests are the same as in vanilla Erlang and can contain the following guard expressions:

```
(quote e)
(cons gexpr gexpr)
(car gexpr)
(cdr gexpr)
(list gexpr ...)
(tuple gexpr ...)
(tref gexpr gexpr)
(binary ...)
(record ...)               - Also the macro versions
(is-record ...)
(record-field ...)
(record-index ...)
(map ...)
(msiz ...) (map-size ...)
(mref ...) (map-get ...)
(mset ...) (map-set ...)
(mupd ...) (map-update ...)
(type-test e)              - Type tests
(guard-bif ...)            - Guard BIFs, arithmetic,
                             boolean and comparison operators
```

An empty guard, `(when)`, always succeeds as there is no test which fails. This simplifies writing macros which handle guards.

Comments in Function Definitions

Inside functions defined with defun LFE permits optional comment strings in the Common Lisp style after the argument list. So we can have:

```
(defun max (x y)
  "The max function."
  (if (>= x y) x y))
```

Optional comments are also allowed in match style functions after the function name and before the clauses:

```
(defun max
  "The max function."
  ((x y) (when (>= x y)) x)
  ((x y) y))
```

This is also possible in a similar style in local functions defined by flet and fletrec:

```
(defun foo (x y)
  "The max function."
  (flet ((m (a b)
           "Local comment."
           (if (>= a b) a b)))
    (m x y)))
```

Variable Binding and Scoping

Variables are lexically scoped and bound by `lambda`, `match-lambda` and `let` forms. All variables which are bound within these forms shadow variables bound outside but other variables occurring in the bodies of these forms will be imported from the surrounding environments.No variables are exported out of the form. So for example the following function:

```
(defun foo (x y z)
  (let ((x (zip y)))
    (zap x z))
  (zop x y))
```

The variable `y` in the call `(zip y)` comes from the function arguments. However, the `x` bound in the `let` will shadow the `x` from the arguments so in the call `(zap x z)` the `x` is bound in the `let` while the `z` comes from the function arguments. In the final `(zop x y)` both `x` and `y` come from the function arguments as the `let` does not export `x`.

Function Binding and Scoping

Functions are lexically scoped and bound by the top-level `defun` and by the macros `flet` and `fletrec`. LFE is a Lisp-2 so functions and variables have separate namespaces and when searching for function both name and arity are used. This means that when calling a function which has been bound to a variable using `(funcall func-var arg ...)` is required to call `lambda`/`match-lambda` bound to a variable or used as a value.

Unqualified functions shadow as stated above which results in the following order within a module, outermost to innermost:

- Predefined Erlang BIFs
- Predefined LFE BIFs
- Imports
- Top-level defines
- Flet/fletrec
- Core forms, these can never be shadowed

This means that it is perfectly legal to shadow BIFs by imports, BIFs/imports by top-level functions and BIFs/imports/top-level by `fletrec`s. In this respect there is nothing special about BIFs, they just behave as predefined imported functions, a whopping big `(import (from erlang ...))`. EXCEPT that we know about guard BIFs and expression BIFs. If you want a private version of `spawn` then define it, there will be no warnings.

CAVEAT This does not hold for the supported core forms. These can be shadowed by imports or redefined but the compiler will *always* use the core meaning and never an alternative. Silently!

Module definition

The basic forms for defining a module and extending its metadata and attributes are:

```
(define-module name meta-data attributes)
(extend-module meta-data attributes)
```

The valid meta data is `(type typedef ...)`, `(opaque typedef ...)`, `(spec function-spec ...)` and `(record record-def ...)`. Each can take multiple definitions in one meta form.

Attributes declarations have the syntax `(attribute value-1 ...)` where the attribute value is a list off the values in the declaration

To simplify defining modules there is a predefined macro:

```
(defmodule name
  "This is the module documentation."
  (export (f 2) (g 1) ... )
  (export all)                          ;Export all functions
  (import (from mod (f1 2) (f2 1) ... )
          (rename mod ((g1 2) m-g1) ((g2 1) m-g2) ... ))
  (module-alias (really-long-module-name rlmn) ...)
  (attr-1 value-1 value-2)
  (meta meta-data ...)
  ... )
```

We can have multiple export and import attributes within module declaration. The `(export all)` attribute is allowed together with other export attributes and overrides them. Other attributes which are not recognized by the compiler are allowed and are simply passed on to the module and can be accessed with the `module_info/0-1` functions.

In the `import` attribute the `(from mod (f1 2) ...)` means that the call `(f1 'everything 42)` will be converted by the compiler to `(mod:f1 'everything 42))` while the `(rename mod ((g2 2) m-g1) ...)` means that the call `(m-g1 'everything 42)` will be converted to `(mod:g1 'everything 42)`. The `rename` form can be used as compact way of indicating the imported function's module. Note that when importing a module

- the compiler does no checking on that module at all

- in the `rename` above the functions `g1/2` and `g2/1` aren't automatically imported, only the "renamed" functions.
- we do not really see in the code that we are calling a function in another module

In the `module-alias` attribute the `(really-long-module-name rlmn)` declaration means that the call `(lrmn:foo 'everything 42)` will be converted by the compiler to `(really-long-module-name:foo 'everything 42)`. This is often used to write short module names in the code when calling functions in modules with long names. It is in many ways better than using `import` as it does not hide that we are calling a function in another module.

Macros

Macro calls are expanded in both body and patterns. This can be very useful to have both make and match macros, but be careful with names.

A macro is function of two arguments which is a called with a list of the arguments to the macro call and the current macro environment. It can be either a lambda or a match-lambda. The basic forms for defining macros are:

```
(define-macro name meta-data lambda|match-lambda)
(let-macro ((name lambda|match-lambda)
  ...)
```

Macros are definitely NOT hygienic in any form. However, variable scoping and variable immutability remove most of the things that can cause unhygienic macros. It can be done but you are not going to do it by mistake. The only real issue is if you happen to be using a variable which has the same name as one which the macro generates, that can cause problems. The work around for this is to give variables created in the macro expansion really weird names like `| - foo - |` which no one in their right mind would use.

To simplify writing macros there are a number of predefined macros:

```
(defmacro name (arg ...) ...)
(defmacro name arg ...)
(defmacro name ((argpat ...) body) ...)
```

Defmacro can be used for defining simple macros or sequences of matches depending on whether the arguments are a simple list of symbols or can be interpreted as a list of pattern/body pairs. In the second case when the argument is just a symbol it will be bound to the whole argument list. For example:

```
(defmacro double (a) `(+ ,a ,a))
(defmacro my-list args `(list ,@args))
(defmacro andalso
  ((list e) `,e)
  ((cons e es) `(if ,e (andalso ,@es) 'false))
  (() `'true))
```

The macro definitions in a macrolet obey the same rules as defmacro.

The macro functions created by defmacro and macrolet automatically add the second argument with the current macro environment with the name `$ENV`. This allows explicit expansion of macros inside the macro and also manipulation of the macro environment. No changes to the environment are exported outside the macro.

User defined macros shadow the predefined macros so it is possible to redefine the built-in macro definitions. However, see the caveat below!

Yes, we have the backquote. It is implemented as a macro so it is expanded at macro expansion time.

Local functions that are only available at compile time and can be called by macros are defined using eval-when-compile:

```
(defmacro foo (x)
  ...
  (foo-helper m n)
  ...)

(eval-when-compile
  (defun foo-helper (a b)
    ...)

  )
```

There can be many eval-when-compile forms. Functions defined within an `eval-when-compile` are mutually recursive but they can only call other local functions defined in an earlier `eval-when-compile` and macros defined earlier in the file. Functions defined in `eval-when-compile` which are called by macros can defined after the macro but must be defined before the macro is used.

Scheme's syntax rules are an easy way to define macros where the body is just a simple expansion. The are implemented the the module `scm` and are supported with `scm:define-syntax` and `scm:let-syntax` and the equivalent `scm:defsyntax` and `scm:syntaxlet`. Note that the patterns are only the arguments to the macro call and do not contain the macro name. So using them we would get:

```
(scm:defsyntax andalso
  (() 'true)
  ((e) e)
  ((e . es) (case e ('true (andalso . es)) ('false 'false))))
```

There is an include file "include/scm.lfe" which defines macros so the names don't have to be prefixed with `scm:`.

CAVEAT While it is perfectly legal to define a Core form as a macro these will silently be ignored by the compiler.

Comments in Macro Definitions

Inside macros defined with defmacro LFE permits optional comment strings in the Common Lisp style after the argument list. So we can have:

```
(defmacro double (a)
  "Double macro."
  `(+ ,a ,a))
```

Optional comments are also allowed in match style macros after the macro name and before the clauses:

```
(defmacro my-list args
  "List of arguments."
  `(list ,@args))

(defmacro andalso
  "The andalso form."
  ((list e) `,e)
  ((cons e es) `(if ,e (andalso ,@es) 'false))
  (() `'true))
```

This is also possible in a similar style in local functions defined by macrolet:

```
(defun foo (x y)
  "The max function."
  (macrolet ((m (a b)
               "Poor macro definition."
               `(if (>= ,a ,b) ,a ,b)))
    (m x y)))
```

Records

Records are tuples with the record name as first element and the rest of the fields in order

exactly like “normal” Erlang records. As with Erlang records the default default value is the atom ‘undefined’.

The basic forms for defining a record, creating, accessing and updating it are:

```
(define-record name (field | (field) |
                     (field default-value) |
                     (field default-value type) ...))
(record name field value field value ...)
(is-record record name)
(record-index name field)
(record-field record name field)
(record-update record name field value field value ...)
```

Note that the list of field/value pairs when making or updating a record is a flat list.

Note that the old `make-record` form has been deprecated and is replaced by `record` which better matches other constructors like `tuple` and `map`. It still exists but should not be used.

We will explain these forms with a simple example. To define a record we do:

```
(define-record person
               ((name "")
                (address "" (string))
                (age)))
```

which defines a record `person` with the fields `name` (default value `""`), `address` (default value `""` and type `(string)`) and `age`. To make an instance of a `person` record we do:

```
(record person name "Robert" age 54)
```

The `record` form is also used to define a pattern.

We can get the value of the `address` field in a person record and set it by doing (the variable `robert` references a `person` record):

```
(record-field robert person address)
(record-update robert person address "my home" age 55)
```

Note that we must include the name of the record when accessing it and there is no need to quote the record and field names as these are always literal atoms.

To simplify defining and using records there is a predefined macro:

```
(defrecord name
  (field) | field
  (field default-value)
  (field default-value type)
  ... )
```

This will create access macros for record creation and accessing and updating fields. The `make-`, `match-` and `update-` forms takes optional argument pairs field-name value to get non-default values. E.g. for

```
(defrecord person
  (name "")
  (address "" (string))
  (age))
```

the following will be generated:

```
(make-person {{field value}} ... )
(match-person {{field value}} ... )
(is-person r)
(fields-person)
(update-person r {{field value}} ... )
(person-name r)
(person-name)
(update-person-name r name)
(person-age r)
(person-age)
(update-person-age r age)
(person-address r)
(person-address)
(update-person-address r address)
```

- `(make-person name "Robert" age 54)` - Will create a new person record with the name field set to “Robert”, the age field set to 54 and the address field set to the default ““.
- `(match-person name name age 55)` - Will match a person with age 55 and bind the variable name to the name field of the record. Can use any variable name here.
- `(is-person john)` - Test if john is a person record.
- `(person-address john)` - Return the address field of the person record john.
- `(person-address)` - Return the index of the address field of a person record.
- `(update-person-address john "back street")` - Updates the address field of the person record john to “back street”.
- `(update-person john age 35 address "front street")` - In the person record john update the age field to 35 and the address field to “front street”.
- `(fields-person)` - Returns a list of fields for the record. This is useful for when using LFE with Mnesia, as the record field names don’t have to be provided manually in the create_table call.
- `(size-person)` - Returns the size of the record tuple.

Note that the older now deprecated `set-` forms are still generated.

Structs

Structs in LFE are the same as Elixir structs and have been defined in the same way so to be truly compatible. This means that you can use structs defined in Elixr from LFE and structs defined in LFE from Elixir.

```
(define-struct (field | (field) |
                (field default-value) |
                (field default-value type) ...))
(struct name field value field value ...)
(is-struct struct)
(is-struct struct name)
(struct-field struct name field)
(struct-update struct name field value field value ...)
```

We will explain these forms with a simple example. To define a struct we do:

```
(define-struct ((name "")
                (address "" (string))
                (age)))
```

which defines a struct with the name of the current module with the fields `name` (default value ""), `address` (default value "" and type `(string)`) and `age`. To make an instance of struct we do:

```
(struct mod-name name "Robert" age 54)
```

The `struct` form is also used to define a pattern.

We can get the value of the `address` field in the struct and set it by doing (the variable `robert` references a struct):

```
(struct-field robert mod-name address)
(struct-update robert mod-name address "my home" age 55)
```

Note that a struct automatically gets the name of the module in which it is defined so that there can only be one struct defined in a module. This mirrors how structs are implemented in Elixir.

Note that we must include the name of the struct when accessing it and there is no need to quote the struct and field names as these are always literal atoms.

Binaries/bitstrings

A binary is

```
(binary seg ... )
```

where `seg` is

```
    byte
    string
    (val integer | float | binary | bitstring | bytes | bits |
         utf8 | utf-8 | utf16 | utf-16 | utf32 | utf-32
         (size n) (unit n)
         big-endian | little-endian | native-endian
         big | little | native
         signed | unsigned)
```

`val` can also be a string in which case the specifiers will be applied to every character in the string. As strings are just lists of integers these are also valid here. In a binary constant all literal forms are allowed on input but they will always be written as bytes.

Maps

A map is created with:

```
(map key value ... )
```

To access maps there are the following forms:

- `(map-size map)` - Return the size of a map.
- `(map-get map key)` - Return the value associated with the key in the map.
- `(map-set map key val ... )` - Set the keys in the map to values. This form can be used to update the values of existing keys and to add new keys.
- `(map-update map key val ... )` - Update the keys in the map to values. Note that this form requires all the keys to already exist in the map.
- `(map-remove map key ... )` - Remove the keys in the map.

There are also alternate short forms `msiz`, `mref`, `mset`, `mupd` and `mrem` based on the Maclisp array reference forms. They take the same arguments as their longer alternatives.

Core forms

Cond

The LFE `cond` is similar to `if` and tests in `cond` are Erlang tests in that they should return either `true` or `false`. If no test succeeds then the `cond` does not generate an exception but just returns `false`. There is a simple catch-all "test" `else` which must last and can be used to

handle the case when all tests fail.

Cond has been extended with the extra test `(?= pat expr)` which tests if the result of `expr` matches the pattern `pat`. If so it binds the variables in `pat` which can be used in the `cond` test body expression. A optional guard is allowed here. An example:

```
(cond ((foo x) ...)
      ((?= (cons x xs) (when (is_atom x)) (bar y))
       (fubar xs (baz x)))
      ((?= (tuple 'ok x) (baz y))
       (zipit x))
      ...
      (else 'yay))
```

Maybe

LFE has an Erlang compatible `maybe`. It has the same features as the Erlang `maybe` with the `?=` operator and `else`. An example:

```
(maybe
  (foo g)
  (?= `#(ok ,a) (a g))
  (bar g)
  ;; (== 'true (>= a 0))
  (?= `#(ok ,b) (b g))
  (+ a b)
  else
  ('error 1 #(got error))
  ('wrong 2 #(got wrong))
  )
```

The maybe body can include `?=` forms which behave in the same way as in the Erlang maybe. As LFE cannot bind variables in the same way as in Erlang we allow `let` in the body to bind variables. These variables are only local in the let body so this body is “lifted” upto the top level maybe body and so can contain `?=` forms as well.

List/binary comprehensions

List/binary comprehensions are supported as macros. The syntax for list comprehensions is:

```
(lc (qual  ...) expr)
(list-comp (qual  ...) expr)
```

where the last expr is used to generate the elements of the list.

The syntax for binary comprehensions is:

```
(bc (qual  ...) bitstringexpr )
(binary-comp (qual  ...) bitstringexpr)
```

where the final expr is a bitstring expression and is used to generate the elements of the binary.

The supported qualifiers, in both list/binary comprehensions are:

```
(<- pat {{guard}} list-expr)        - Extract elements from list
(<= bin-pat {{guard}} binary-expr)  - Extract elements from binary
expr                                - Normal boolean test
```

Some examples:

```
(lc ((<- v (when (> v 5)) l1)
     (== (rem v 2) 0))
  v)
```

returns a list of all the even elements of the list `l1` which are greater than 5.

```
(bc ((<= (binary (f float (size 32))) b1)
     (> f 10.0))
  (progn
    (: io fwrite "~p\n" (list f))
    (binary (f float (size 64)))))
```

returns a binary of floats of size 64 bits which are from the binary b1 where they are of size 32 bits and larger than 10.0. The returned numbers are first printed.

This could also be written using a guard for the test:

```
(bc ((<= (binary (f float (size 32))) (when (> f 10.0)) b1))
  (progn
    (: io fwrite "~p\n" (list f))
    (binary (f float (size 64)))))
```

ETS and Mnesia

LFE also supports match specifications and Query List Comprehensions. The syntax for a match specification is the same as for match-lambdas:

```
(ets-ms
  ((arg ... ) {{(when e ...)}} ...)             - Matches clauses
  ... )
```

For example:

```
(ets:select db (ets-ms
                 ([(tuple _ a b)] (when (> a 3)) (tuple 'ok b))))
```

It is a macro which creates the match specification structure which is used in `ets:select` and `mnesia:select`. For tracing instead of the `ets-ms` macro there is the `trace-ms` macro which is also used in conjunction with the `dbg` module. The same restrictions as to what can be done apply as for vanilla match specifications:

- There is only a limited number of BIFs which are allowed
- There are some special functions only for use with dbg
- For ets/mnesia it takes a single parameter which must a tuple or a variable
- For dbg it takes a single parameter which must a list or a variable

N.B. the current macro neither knows nor cares whether it is being used in ets/mnesia or in dbg. It is up to the user to get this right.

Macros, especially record macros, can freely be used inside match specs.

CAVEAT Some things which are known not to work in the current version are andalso, orelse and record updates.

Query List Comprehensions

LFE supports QLCs for mnesia through the qlc macro. It has the same structure as a list comprehension and generates a Query Handle in the same way as with `qlc:q([...])`. The handle can be used together with all the combination functions in the module qlc.

For example:

```
(qlc (lc ((<- (tuple k v) (: ets table e2)) (== k i)) v)
     {{Option}})
```

Macros, especially record macros, can freely be used inside query list comprehensions.

CAVEAT Some things which are known not to work in the current version are nested QLCs and let/case/recieve which shadow variables.

Predefined LFE functions

The following more or less standard lisp functions are predefined:

```
(<arith_op> expr ...)
(<comp_op> expr ...)
```

The standard arithmetic operators, + - * /, and comparison operators, > >= < =< == /= =:= =/= , can take multiple arguments the same as their standard lisp counterparts. This is still experimental and implemented using macros. They do, however, behave like normal functions and evaluate ALL their arguments before doing the arithmetic/comparisons operations.

```
(acons key value list)
(pairlis keys values {{list}})
(assoc key list)
(assoc-if test list)
(assoc-if-not test list)
(rassoc value list)
(rassoc-if test list)
(rassoc-if-not test list)
```

The standard association list functions.

```
(subst new old tree)
(subst-if new test tree)
(subst-if-not new test tree)
(sublis alist tree)
```

The standard substitution functions.

```
(macroexpand-1 expr {{environment}})
```

If Expr is a macro call, does one round of expansion, otherwise returns Expr.

```
(macroexpand expr {{environment}})
```

Returns the expansion returned by calling macroexpand-1 repeatedly, starting with Expr, until the result is no longer a macro call.

```
(macroexpand-all expr {{environment}})
```

Returns the expansion from the expression where all macro calls have been expanded with macroexpand.

NOTE that when no explicit environment is given the macroexpand functions then only the default built-in macros will be expanded. Inside macros and in the shell the variable $ENV is bound to the current macro environment.

```
(eval expr {{environment}})
```

Evaluate the expression expr. Note that only the pre-defined lisp functions, erlang BIFs and exported functions can be called. Also no local variables can be accessed. To access local variables the expr to be evaluated can be wrapped in a let defining these.

For example if the data we wish to evaluate is in the variable expr and it assumes there is a local variable “foo” which it needs to access then we could evaluate it by calling:

```
(eval `(let ((foo ,foo)) ,expr))
```

Supplemental Common Lisp Functions

LFE provides the module cl which contains the following functions which closely mirror functions defined in the Common Lisp Hyperspec. Note that the following functions use zero-based indices, like Common Lisp (unlike Erlang, which start at index ‘1’). A major difference between the LFE versions and the Common Lisp versions of these functions is that the boolean values are the LFE `'true` and `'false`. Otherwise the definitions closely follow the CL definitions and won’t be documented here.

```
cl:make-lfe-bool cl-value
cl:make-cl-bool lfe-bool

cl:mapcar  function  list
cl:maplist  function  list
```

```
cl:mapc  function  list
cl:mapl  function  list

cl:symbol-plist  symbol
cl:symbol-name  symbol
cl:get  symbol  pname
cl:get  symbol  pname  default
cl:getl  symbol  pname-list
cl:putprop  symbol  value  pname
cl:remprop  symbol  pname

cl:getf  plist  pname
cl:getf  plist  pname  default
cl:putf  plist  value  pname  ; This does not exist in CL
cl:remf  plist  pname
cl:get-properties  plist  pname-list

cl:elt  index  sequence
cl:length  sequence
cl:reverse  sequence
cl:some  predicate  sequence
cl:every  predicate  sequence
cl:notany  predicate  sequence
cl:notevery  predicate  sequence
cl:reduce  function  sequence
cl:reduce  function  sequence  'initial-value  x
cl:reduce  function  sequence  'from-end  'true
cl:reduce  function  sequence  'initial-value  x  'from-end  'true

cl:remove  item  sequence
cl:remove-if  predicate  sequence
cl:remove-if-not  predicate  sequence
cl:remove-duplicates  sequence

cl:find  item  sequence
cl:find-if  predicate  sequence
cl:find-if-not  predicate  sequence
cl:find-duplicates  sequence
cl:position  item  sequence
cl:position-if  predicate  sequence
cl:position-if-not  predicate  sequence
cl:position-duplicates  sequence
cl:count  item  sequence
cl:count-if  predicate  sequence
cl:count-if-not  predicate  sequence
cl:count-duplicates  sequence

cl:car  list
cl:first  list
cl:cdr  list
cl:rest  list
cl:nth  index  list
cl:nthcdr  index  list
cl:last  list
cl:butlast  list

cl:subst  new  old  tree
cl:subst-if  new  test  tree
cl:subst-if-not  new  test  tree
cl:sublis  alist  tree

cl:member  item  list
cl:member-if  predicate  list
cl:member-if-not  predicate  list
cl:adjoin  item  list
cl:union  list  list
cl:intersection  list  list
cl:set-difference  list  list
cl:set-exclusive-or  list  list
cl:subsetp  list  list

cl:acons  key  data  alist
cl:pairlis  list  list
cl:pairlis  list  list  alist
```

```
cl:pairlis  list  list  alist
cl:assoc  key  alist
cl:assoc-if  predicate  alost
cl:assoc-if-not  predicate  alost
cl:rassoc  key  alist
cl:rassoc-if  predicate  alost
cl:rassoc-if-not  predicate  alost

cl:type-of  object
cl:coerce  object  type
```

Furthermore, there is an include file which developers may which to utilize in their LFE programs: `(include-lib "lfe/include/cl.lfe")`. Currently this offers Common Lisp predicates, but may include other useful macros and functions in the future. The provided predicate macros wrap the various `is_*` Erlang functions; since these are expanded at compile time, they are usable in guards. The include the following:

```
(alivep x)
(atomp x)
(binaryp x)
(bitstringp x)
(boolp x) and (booleanp x)
(builtinp x)
(consp x)
(floatp x)
(funcp x) and (functionp x)
(intp x) and (integerp x)
(listp x)
(mapp x)
(numberp x)
(pidp x)
(process-alive-p x)
(recordp x tag)
(recordp x tag size)
(refp x) and (referencep x)
(tuplep x)
(vectorp x)
```

Non-predicate macros in `lfe/include/cl.lfe` include:

```
(dolist ...)
(vector ...)
```

Supplemental Clojure Functions

From LFE's earliest days, it's Lisp-cousin Clojure (created around the same time) has inspired LFE developers to create similar, BEAM-versions of those functions. These were collected in a separate library and then expanded upon, until eventually becoming part of the LFE standard library.

Function definition macros:

```
(clj:defn ...)
(clj:defn- ...)
(clj:fn ...)
```

Threading macros:

```
(clj:-> ...)
(clj:->> ...)
(clj:as-> ...)
(clj:cond-> ...)
(clj:cond->> ...)
(clj:some-> ...)
(clj:some->> ...)
(clj:doto ...)
```

Conditional macros:

```
(clj:if-let ...)
(clj:iff-let ...)
(clj:condp ...)
(clj:if-not ...)
(clj:iff-not ...)
(clj:when-not ...)
(clj:not= ...)
```

Predicate macros:

```
(clj:atom? x)
(clj:binary? x)
(clj:bitstring? x)
(clj:bool? x)
(clj:boolean? x)
(clj:even? x)
(clj:false? x)
(clj:falsy? x)
(clj:float? x)
(clj:func? x)
(clj:function? x)
(clj:identical? x)
(clj:int? x)
(clj:integer? x)
(clj:map? x)
(clj:neg? x)
(clj:nil? x)
(clj:number? x)
(clj:odd? x)
(clj:pos? x)
(clj:record? x)
(clj:reference? x)
(clj:true? x)
(clj:tuple? x)
(clj:undef? x)
(clj:undefined? x)
(clj:zero? x)
```

Other:

```
(clj:str x)
(clj:lazy-seq x)
(clj:conj ...)
(clj:if ...)
```

Most of the above mentioned macros are available in the `clj` include file, the use of which allows developers to forego the `clj:` prefix in calls:

```
(include-lib "lfe/include/clj.lfe")
```

Notes

- NYI - Not Yet Implemented
- N.B. - Nota bene (note well)

SEE ALSO

lfe(1), **lfescript(1)**, **lfe_cl(3)**