LFE-1

erlang

Reference Guide

Table of Contents

LFE Reference Guide

Introduction

This Gitbook (available here) is a conversion of Robert Virding's LFE User Guide.

Formatting Note

You will see code blocks in this book surrounded by double curly braces, `{{ ... }}`. This typographic convention is used to denote *optional* syntax.

Special syntactic rules

Syntax	Definition
`#b` , `#o` , `#d` , `#x` , and `#23r`	Based integers
`#(e e ... )`	Tuple constants
`#b(e e ... )`	Binary constants, `e` ... are valid literals segments
`#m(k v ... )`	Map constants, `k v` are keys and values
`[ ... ]`	Allowed as alternative to `( ... )`

Supported Core forms

- `(quote e)`
- `(cons head tail)`
- `(car e)`
- `(cdr e)`
- `(list e ... )`
- `(tuple e ... )`
- `(binary seg ... )`
- `(map key val ...)`, `(map-get m k)`, `(map-set m k v ...)`, `(map-update m k v ...)`

```
(lambda (arg ...) ...)
  (match-lambda
    ((arg ... ) {{{when e ...)}} ...) ; Matches clauses
    ... )
```

```
(let ((pat {{{when e ...)}} e)
      ...)
  ... )
```

```
(let-function ((name lambda|match-lambda) ; Only define local
               ... )                      ; functions
  ... )
```

```
(letrec-function ((name lambda|match-lambda) ; Only define local
                  ... )                      ; functions
  ... )
```

```
(let-macro ((name lambda-match-lambda) ; Only define local
            ...)                       ; macros
  ...)
```

- `(progn ... )`
- `(if test true-expr {{false-expr}})`

```
(case e
  (pat {{{when e ...)}} ...)
   ... ))
```

```
(receive
  (pat {{{when e ...)}} ... )
  ...
  (after timeout ... ))
```

- `(catch ... )`

```
(try
  e
  {{{case ((pat {{{when e ...)}} ... )
          ... ))}}
  {{{catch
     ; Next must be tuple of length 3!
```

```
      (((tuple type value ignore) {{(when e ...)}}
       ... )
      ... )}}
   {{(after ... )}})
```

- `(funcall func arg ... )`
- `(call mod func arg ... )` - Call to Erlang `Mod:Func(Arg, ... )`
- `(define-module name declaration ... )`
- `(extend-module declaration ... )` - Define/extend module and declarations.
- `(define-function name lambda|match-lambda)`
- `(define-macro name lambda|match-lambda)` - Define functions/macros at top-level.

Supported macro forms

Form	Notes
`(: mod func arg ... )`	Expands to `(call 'mod 'func arg ... )`
`(mod:func arg ... )`	Expands to `(call 'mod 'func arg ... )`
`(? {{timeout {{default}} }})`	Receive next message, optional timeout and default value
`(++ ... )`	List concatenation (as in Erlang `++`)
`(list* ...)`	
`(let* (...) ... )`	Sequential let's
`(flet ((name (arg ...) ...) ...) ...)`	
`(flet* (...) ... )`	Sequential `flet` 's
`(fletrec ((name (arg ...) ...) ...) ...)`	Define local functions, this will expand to lambda or match-lambda depending on structure as with defun.
`(cond ... )`	The normal cond, with `(?= pat expr)`
`(andalso ... )`	
`(orelse ... )`	
`(fun func arity)`	Erlang `fun func/arity`
`(fun mod func arity)`	Erlang `fun mod:func/arity`
`(lc (qual ...) ...)`	Erlang `[ expr || qual ... ]`
`(bc (qual ...) ...)`	Erlang `<< expr || qual ... >>`
`(match-spec ...)`	Erlang `ets:fun2ms(fun ( ) -> end)`

Common Lisp-inspired macros

Define a top-level function:

```
(defun name (arg ...) ...)
```

Define a toplevel function with pattern-matching arguments; this will expand to lambda or match-lambda depending on structure:

```
(defun name
  ((argpat ...) ...)
  ...)
```

Define a top-level macro:

```
(defmacro name (arg ...) ...)
```

```
(defmacro name arg ...)
```

Define a top-level macro with pattern-matching arguments; this will expand to lambda or match-lambda depending on structure:

```
(defmacro name
  ((argpat ...) ...)
  ...)
```

Define a top-level macro using Scheme inspired syntax-rules format.

```
(defsyntax name
  (pat exp)
  ...)
```

Define local macros in macro or syntax-rule format:

```
(macrolet ((name (arg ...) ...)
           ...)
  ...)
```

```
(syntaxlet ((name (pat exp) ...)
            ...)
  ...)
```

Like their CL counterparts:

```
(prog1 ...)
```

```
(prog2 ...)
```

Define an Erlang LFE module:

```
(defmodule name ...)
```

Define an Erlang LFE record:

```
(defrecord name ...)
```

Older Scheme-inspired macros

```
(define (name arg ...) ...)
```

```
(define name lambda|match-lambda)
```

```
(define-syntax name
  (syntax-rules (pat exp) ...)|(macro (pat body) ...))
```

```
(let-syntax ((name ...)
             ...)
  ...)
```

```
(begin ...)
```

```
(define-record name ...)
```

Patterns

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

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

Repeated variables are *NOT* supported in patterns, there is no automatic comparison of values. It must explicitly be done in a guard.

`_` 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 (`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 ...)`
- `(binary ...)`
- `(progn gtest ...)` - Sequence of guard tests
- `(if gexpr gexpr gexpr)`
- `(type-test e)`
- `(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)))
```

Bindings and Scoping

LFE is a Lisp-2 and has separate namespaces for variables and functions/macros. Both variables and functions/macros are lexically scoped. Variables are bound by `lambda`, `match-lambda` and `let`. Functions are bound by top-level `defun`, `flet` and `fletrec`. Macros are bound by top-level `defmacro` / `defsyntax` and by `macrolet` / `syntaxlet`.

When searching for functions, both name and arity are used. A macro is considered to have any arity and will match all functions with that name. While this is not consistent with either Scheme (or CL) it is simple, usually easy to understand, and fits Erlang quite well. It does, however, require using `(funcall func arg ... )` like CL to call `lambdas` / `match-lambdas` `(funs)` bound to variables.

Core solves this by having separate bindings and special to have only one apply:

- `apply _F (...)`, and
- `apply _F/3 ( a1, a2, a3 )`

Function shadowing

Unqualified functions shadow as stated previously. This results in the following order within a module, outermost to innermost:

- Predefined BIFs (same as in vanilla Erlang)
- Predefined LFE BIFs
- Imports
- Top-level defines
- Flet/fletrec

This means that it is perfectly legal to shadow BIFs by imports, BIFs/imports by top-level functions and BIFs/imports/top-level by `fletrecs`. In this respect there is nothing special about BIfs, they just behave as prefined 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

```
(defmodule name
  (export (f 2) (g 1) ... )
  (export all)
  (import (from mod (f1 2) (f2 1) ... )
          (rename mod ((f1 2) sune) ((f2 1) kurt) ... ))
  (import (prefix mod mod-prefix))              - NYI
  (attr-1 value-1 value-2)
  ... )
```

There can be multiple `export` and `import` declarations within module declaration. The `(export all)` declaration is allowed together with other `export` declarations and overrides them. Other attributes which are not recognised by the compiler are allowed and are simply passed on to the module and can be accessed through `module_info/0` / `module_info/1`.

Parameterized modules

```
(defmodule (name par1 par2 ... )
  ... )
```

Define a parameterized module which behaves the same way as in vanilla Erlang. For now, avoid defining functions `new` and `instance`.

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 argument 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 lambda|match-lambda)
```

```
(let-macro ((name lambda|match-lambda)
  ...)
```

Macros are definitely NOT hygienic in any form.

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 be 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. These are supported with `defsyntax` and `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:

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

NOTE: These are definitely NOT hygienic.

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)))
```

Extended cond

`cond` has been extended with the extra test `(?= pat expr)` which tests if the result of `expr` matches `pat`. If so, it binds the variables in `pat` which can be used in the `cond`. An 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))
      ... )
```

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 `'undefined'`.

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

The record defined above will create access functions/macros for creation and access fields. The `make`, `match` and `set` forms takes optional argument pairs `field-name value` to get non-default values. E.g. for the record:

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

We have:

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

Examples

```
(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.

```
(emp-person age '$1)
```

Create an Ets Match Pattern for record `person` where the `age` field is set to `$1` and all other fields are set to `_`.

```
(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.

```
(set-person-address john '"back street")
```

Sets the `address` field of the `person` record `john` to `"back street"`.

```
(set-person john age 35 address '"front street")
```

In the `person` record john set 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.

Binaries/bitstrings

A binary is

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

where `seg` is:

```
    byte
    string
    (val integer|float|binary|bitstring|bytes|bits
         (size n) (unit n)
         big-endian|little-endian|native-endian|little|native|big
         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:

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

To access maps there are the following forms:

- Return the value associated with key in map:

  ```
  (map-get map key)
  ```

- Set keys in map to values:

  ```
  (map-set map key val ... )
  ```

- Update keys in map to values. Note that this form requires all the keys to exist:

  ```
  (map-update map key val ... )
  ```

NOTE: This syntax for processing maps has stablized, but may change in the future!

There is also an alternate short form map functions:

- `mref`
- `mset`
- `mupd`

These are based on the MACLISP array reference forms. They take the same arguments as their longer alternatives.

List/binary comprehensions

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

```
(lc (qual  ...) expr ... )
```

where the final `expr` is used to generate the elements of the list.

An alias is provided such that the following is also valid:

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

The syntax for binary comprehensions is:

```
(bc (qual  ...) expr ... )
```

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

An alias is provided such that the following is also valid:

```
(binary-comp (qual  ...) expr ... )
```

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

Form	Notes
`(<- pat {{guard}} list-expr)`	Extract elements from a list expression
`(<= bin-pat {{guard}} binary-expr)`	Extract elements from a binary/bits expression
`(?= pat {{guard}} expr)`	Match test and bind variables in `pat`
`expr`	Normal boolean test

Some examples:

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

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

```
(bc ((<= (f float (size 32)) b1)  ; No wrapping, only bitseg needed
     (> f 10.0))
  (io:fwrite "~p\n" (list f))
  (f float (size 64)))            ; No wrapping, only bitseg needed
```

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

NOTE: A word of warning when using guards when extracting elements from a binary. When a match/guard fails for a binary no more attempts will be made to extract data from the binary. This means that even if a value could be extracted

from the binary, if the guard fails, this value will be lost and extraction will cease. This is **NOT** the same as having following boolean test which may remove an element but will not stop extraction. Using a guard is probably not what you want!

Normal vanilla Erlang does the same thing but does not allow guards.

ETS and Mnesia

Apart from `(emp-record ...)` macros for ETS Match Patterns, which are also valid in Mnesia, LFE also supports match specifications and Query List Comprehensions. The syntax for a match specification is the same as for `match-lambdas`:

```
(match-spec
  ((arg ... ) {{(when e ...)}} ...) ;; Matches clauses
  ... )
```

For example:

```
(ets:select db (match-spec
                 (((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`. The same `match-spec` macro can also be used 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

NOTE: 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:

Operators

Arithmetic

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

The standard arithmetic operators, `+` `-` `*` and `/` 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 operations.

Examples:

```
> (- 43 1)
42
> (* 21 2)
42
```

Comparison

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

The standard comparison operators, `>` , `>=` , `<` , `=<` , `==` , `/=` , `=:=` , and `=/=` 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 comparison operations.

Examples:

```
> (> 1 42)
false
> (< 42 43)
true
```

Association list functions

These are the same as found in Common Lisp.

acons

```
(acons key value list)
```

`acons` constructs a new association list by adding the pair `(key . datum)` to the old `list` .

Example:

```
> (acons 'x 'y 'a)
((x . y) . a)
```

pairlis

```
(pairlis keys values {{list}})
```

`pairlis` takes two lists and makes an association list that associates elements of the first list to corresponding elements of the second list.

Example:

```
> (pairlis '(one two) '(1 2) '((three . 3) (four . 19)))
((one . 1) (two . 2) (three . 3) (four . 19))
```

assoc

```
(assoc key list)
```

Searches the association list `list`. The value returned is the first pair in the `list` such that the `car` of the pair equals the `key` passed to `assoc`. or `()` If there is no such pair in the `list`, an empty list `()` is returned.

Examples:

```
> (assoc 'r '((a . b) (c . d) (r . x) (s . y) (r . z)))
(r . x)
> (assoc 'goo '((foo . bar) (zoo . goo)))
()
> (assoc '2 '((1 a b c) (2 b c d) (-7 x y z)))
(2 b c d)
>
```

assoc-if

```
(assoc-if test list)
```

Searches the association list `list`. The value is the first pair in the `list` such that the `car` of the pair satisfies the test, or `()` if there is no such pair in the `list`.

Examples:

```
> (assoc-if #'is_atom/1 '(("a" . "b") (3 . 4) (r . x) (s . y) (r . z)))
(r . x)
> (assoc-if #'is_integer/1 '(("a" . "b") (3 . 4) (r . x) (s . y) (r . z)))
(3 . 4)
> (assoc-if #'is_list/1 '(("a" . "b") (3 . 4) (r . x) (s . y) (r . z)))
("a" 98)
> (assoc-if #'is_float/1 '(("a" . "b") (3 . 4) (r . x) (s . y) (r . z)))
()
```

assoc-if-not

```
(assoc-if-not test list)
```

Searches the association list `list`. The value is the first pair in the `list` such that the `car` of the pair satisfies the test, or `()` if there is no such pair in the `list`.

Examples:

```
> (assoc-if-not #'is_float/1 '(("a" . "b") (3 . 4) (r . x) (s . y) (r . z)))
("a" 98)
> (assoc-if-not #'is_list/1 '(("a" . "b") (3 . 4) (r . x) (s . y) (r . z)))
(3 . 4)
```

rassoc

```
(rassoc value list)
```

`rassoc` is the reverse form of `assoc`; it searches for a pair whose `cdr` satisfies the `test`, rather than the `car`.

Examples:

```
> (rassoc 'a '(("a" . "b") (r . 4) (3 . x) (s . y) (r . z)))
()
> (rassoc '4 '(("a" . "b") (r . 4) (3 . x) (s . y) (r . z)))
(r . 4)
> (rassoc 'z '(("a" . "b") (r . 4) (3 . x) (s . y) (r . z)))
(r . z)
```

rassoc-if

```
(rassoc-if test list)
```

`rassoc` is the reverse form of `assoc`; it searches for a pair whose `cdr` satisfies the `test`, rather than the `car`.

Examples:

```
> (rassoc-if #'is_atom/1 '(("a" . "b") (r . 4) (3 . x) (s . y) (r . z)))
(3 . x)
> (rassoc-if
    (lambda (x)
      (== x 'y))
    '(("a" . "b") (r . 4) (3 . x) (s . y) (r . z)))
(s . y)
```

rassoc-if-not

```
(rassoc-if-not test list)
```

`rassoc` is the reverse form of `assoc`; it searches for a pair whose `cdr` satisfies the `test`, rather than the `car`.

Example:

```
> (rassoc-if-not
```

```
    (lambda (x)
      (or (is_list x) (is_integer x)))
    '(("a" . "b") (r . 4) (3 . x) (s . y) (r . z)))
(3 . x)
```

Substitution of expressions

These are the same as found in Common Lisp.

`subst`

```
(subst new old tree)
```

`(subst new old tree)` makes a copy of `tree`, substituting `new` for every subtree or leaf of `tree` (whether the subtree or leaf is a `car` or a `cdr` of its parent) such that `old` and the subtree or leaf are equal to `new`. It returns the modified copy of `tree`. The original `tree` is unchanged.

Examples:

```
> (subst 'tempest 'hurricane
         '(shakespeare wrote (the hurricane)))
(shakespeare wrote (the tempest))
> (subst 'tempest 'hurricane
         '(thoughts:
           (shakespeare wrote (the hurricane))
           (hurricane (on (a (teacozy))))))
(thoughts: (shakespeare wrote (the tempest)) (tempest (on (a (teacozy)))))
> (subst 'foo '() '(shakespeare wrote (twelfth night)))
(shakespeare wrote (twelfth night . foo) . foo)
```

`subst-if`

```
(subst-if new test tree)
```

`(subst new old tree)` makes a copy of `tree`, substituting `new` for every subtree or leaf of `tree` (whether the subtree or leaf is a `car` or a `cdr` of its parent) such that `old` and the subtree or leaf satisfy the test. It returns the modified copy of `tree`. The original `tree` is unchanged.

Example:

```
> (subst-if
    '(a . cons)
    (lambda (x)
      (== x '(old . pair)))
    '((old . spice) ((old . shoes) old . pair) (old . pair)))
((old . spice) ((old . shoes) a . cons) (a . cons))
```

`subst-if-not`

```
(subst-if-not new test tree)
```

`(subst new old tree)` makes a copy of `tree`, substituting `new` for every subtree or leaf of `tree` (whether the subtree or leaf is a `car` or a `cdr` of its parent) such that `old` and the subtree or leaf satisfy the test. It returns the modified copy of `tree`. The original `tree` is unchanged.

Examples:

```
> (subst-if-not
    '(a . cons)
    (lambda (x)
      (== x '(old . pair)))
    '((old . spice) ((old . shoes) old . pair) (old . pair)))
(a . cons)
> (subst-if-not
    '(a . cons)
    (lambda (x)
      (/= x '(old . pair)))
    '((old . spice) ((old . shoes) old . pair) (old . pair)))
((old . spice) ((old . shoes) a . cons) (a . cons))
```

`subls`

```
(sublis list tree)
```

`sublis` makes substitutions for objects in a `tree` (a structure of `cons`es). The first argument to `sublis` is an association `list`. The second argument is the `tree` in which substitutions are to be made, as with `subst`. `sublis` looks at all subtrees and leaves of the `tree`; if a subtree or leaf appears as a key in the association `list` (that is, the key and the subtree or leaf satisfy the test), it is replaced by the object with which it is associated. This operation is non-destructive. In effect, `sublis` can perform several `subst` operations simultaneously.

Examples:

```
> (sublis '((x . 100) (z . zprime))
          '(plus x (minus g z x p) 4 . x))
(plus 100 (minus g zprime 100 p) 4 . 100)
> (sublis '(((+ x y) . (- x y)) ((- x y) . (+ x y)))
          '(* (/ (+ x y) (+ x p)) (- x y)))
(* (/ (- x y) (+ x p)) (+ x y))
```

Expansion macros

```
(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: When no explicit environment is given to 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.

Evaluation

```
(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))
```