lfe_clj(3)

Tim Dysinger, Duncan McGreggor, Eric Bailey

2015-2016

NAME

clj - LFE Clojure interface library.

SYNOPSIS

This module provides Clojure-inspired functions and macros for use in LFE.

EXPORTS

N.B. Instead of making fully-qualified calls to the macros exported from *clj*, you may `(include-lib "lfe/include/clj.lfe")` and then call them directly, e.g.

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

(-> 2 (+ 2) (=:= 4)) ; 'true
```

Function Macros

(defn name [arg …] {{doc-string}} …)

(defn {{doc-string}} ([argpat …] …))

Define and automatically export a function.

(defn- name [arg …] {{doc-string}} …)

(defn- {{doc-string}} ([argpat …] …))

Equivalent to **defun**.

(fn (arg …) …)

Equivalent to `lambda`.

Threading Macros

Note: The original versions were copied from Tim Dysinger's lfesl repo here:

`https://github.com/lfex/lfesl/blob/master/include/thread.lfe`

They have since been modified to be safely exportable.

(-> …)

Thread first.

Example usage, demonstrating ordering:

```
> (set o '(#(a 1) #(b 2) #(c 3)))
(#(a 1) #(b 2) #(c 3))
> (clj:-> o
>         (++ '(#(d 4)))
>         (++ '(#(e 5)))
>         (++ '(#(f 6))))
(#(a 1) #(b 2) #(c 3) #(d 4) #(e 5) #(f 6))
```

Note that the use of **->** in this example results in each successive value being *appended* to the input list.

Another example showing how this works:

```
> (lists:sublist
>   (lists:reverse
>     (lists:sort
>       (lists:merge
>         (string:tokens
>           (string:to_upper "a b c d e")
>           " ")
>         '("X" "F" "L"))))
>   2 3)
("L" "F" "E")
```

Can be rewritten as this:

```
> (clj:-> "a b c d e"
>         (string:to_upper)
>         (string:tokens " ")
>         (lists:merge '("X" "F" "L"))
>         (lists:sort)
>         (lists:reverse)
>         (lists:sublist 2 3))
("L" "F" "E")
```

(->> …)

Thread last.

Example usage, demonstrating ordering:

```
> (set o '(#(a 1) #(b 2) #(c 3)))
(#(a 1) #(b 2) #(c 3))
> (clj:->> o
>          (++ '(#(d 4)))
>          (++ '(#(e 5)))
>          (++ '(#(f 6))))
(#(f 6) #(e 5) #(d 4) #(a 1) #(b 2) #(c 3))
```

Note that the use of **->>** in this example results in each successive value being *prepended* to the input list.

Another example showing how this:

```
> (lists:foldl #'+/2 0
>   (clj:take 10
>     (lists:filter
>       (clj:comp #'clj:even?/1 #'clj:round/1)
>       (lists:map
>         (lambda (x)
>           (math:pow x 2))
>         (clj:seq 42)))))
1540.0
```

Can be rewritten as this:

```
> (clj:->> (clj:seq 42)
>          (lists:map (lambda (x) (math:pow x 2)))
>          (lists:filter (clj:comp #'clj:even?/1 #'clj:round/1))
>          (clj:take 10)
>          (lists:foldl #'+/2 0))
1540.0
```

(as-> expr name . sexps)

Bind `name` to `expr`, evaluate the first `sexp` in the lexical context of that binding, then bind `name` to that result, repeating for each successive `sexp` in `sexps`, returning the result of the last `sexp`.

(cond-> expr . clauses)

Given an `expr`ession and a set of `test`/`sexp` pairs, thread `x` (via **->**) through each `sexp` for which the corresponding `test` expression is truthy, i.e. neither `'false` nor `'undefined`. Note that, unlike **cond** branching, **cond->** threading does not short circuit after the first truthy test expression.

(cond->> expr . clauses)

Given an `expr`ession and a set of `test`/`sexp` pairs, thread `x` (via **->>**) through each `sexp` for which the corresponding `test` expression is truthy, i.e. neither `'false` nor `'undefined`. Note that, unlike **cond** branching, **cond->>** threading does not short circuit after the first truthy `test` expression.

(some-> x . sexps)

When `x` is not `'undefined`, thread it into the first `sexp` (via **->**), and when that result is not `'undefined`, through the next, etc.

(some->> x . sexps)

When `x` is not `'undefined`, thread it into the first `sexp` (via **->>**), and when that result is not `'undefined`, through the next, etc.

Conditional Macros

(if-let ((patt test)) then {{else}})

If `test` evaluates to anything other than `'false` or `'undefined`, evaluate `then` with `patt` bound to the value of `test`, otherwise `else`, if supplied, else `'undefined`.

(iff-let ((patt test)) . body)

When `test` evaluates to anything other than `'false` or `'undefined`, evaluate `body` with `patt` bound to the value of `test`, otherwise return `'undefined`.

(condp pred expr . clauses)

Given a binary predicate, an expression and a set of clauses of the form:

```
test-expr result-expr

test-expr >> result-fn
```

where `result-fn` is a unary function, if `(pred test-expr expr)` returns anything other than `'undefined` or `'false`, the clause is a match.

If a binary clause matches, return `result-expr`. If a ternary clause matches, call `result-fn` with the result of the predicate and return the result.

If no clause matches and a single default expression is given after the clauses, return it. If no default expression is given and no clause matches, throw a `no-matching-clause` error.

(if-not test then)

(if-not test then else)

If `test` evaluates to `'false` or `'undefined`, evaluate and return `then`, otherwise `else`, if supplied, else `'undefined`.

(iff test . body)

Like Clojure's `when`. If `test` evaluates to anything other than `'false` or `'undefined`, evaluate `body` in an implicit `progn`.

(when-not test . body)

If `test` evaluates to `'false` or `'undefined`, evaluate `body` in an implicit `progn`. Otherwise return `'undefined`.

(not= x)

(not= x y)

(not= x y . more)

Same as `(not (== ...))`.

Predicate Macros

Allowed in guards, unless otherwise stated.

(tuple? x)

Return `'true` if `x` is a tuple.

(atom? x)

Return `'true` if `x` is an atom.

(binary? x)

Return `'true` if `x` is a binary.

(bitstring? x)

Return `'true` if `x` is a bitstring.

(boolean? x)

(bool? x)

Return `'true` if `x` is a boolean.

(float? x)

Return `'true` if `x` is a float.

(function? f)

(func? f)

Return `'true` if `x` is a function.

(function? f n)

(func? f n)

Return `'true` if `f` is an `n`-ary function.

(integer? x)

(int? x)

Return `'true` if `x` is an integer.

(number? x)

Return `'true` if `x` is a number.

(record? x record-tag)

(record? x record-tag size)

Return `'true` if `x` is a tuple and its first element is `record-tag`. If `size` is given, check that `x` is a `record-tag` record of size `size`.

N.B. **record?/2** may yield unexpected results, due to difference between the Erlang and LFE

compilers. As such, whenever possible, prefer **record?/3**."

(reference? x)

Return `'true` if `x` is a reference.

(map? x)

Return `'true` if `x` is a map. Return `'false` on versions of Erlang without maps.

(undefined? x)

(undef? x)

Return `'true` if `x` is the atom `'undefined`.

(nil? x)

Return `'true` if `x` is the atom `'nil` or the empty list.

(true? x)

Return `'true` if `x` is the atom `'true`.

(false? x)

Return `'true` if `x` is the atom `'false`.

(falsy? x)

Return `'true` if `x` is one of the atoms `'false` and `'undefined`.

(odd? x)

Return `'true` if `x` is odd.

(even? x)

Return `'true` if `x` is even.

(zero? x)

Return `'true` if `x` is zero.

(pos? x)

Return `'true` if `x` is greater than zero.

(neg? x)

Return `'true` if `x` is less than zero.

(identical? x)

Return `'true` if `x` is exactly equal to `y`.

Other Macros

(str x1, x2 … xn)

Given arbitrary number of arguments, return a string consisting of each of their string representations.

N.B. Because Erlang characters are represented as integers, this will not work for chars, e.g. `#\a`, which will be presented in the return value as its integer value, i.e. `"97"`.

```
> (clj:str #\a "bc")
"97bc"
> (clj:str "a" "bc")
"abc"
```

(lazy-seq)

(lazy-seq seq)

Return a (possibly infinite) lazy sequence from a given lazy sequence `seq` or a finite lazy sequence from given list `seq`. A lazy sequence is treated as finite if at any iteration it produces the empty list, instead of a cons cell with data as the head and a nullary function for the next iteration as the tail.

(conj coll . xs)

conj[oin] a value onto an existing collection. Prepend to a list, append to a tuple, and merge maps.

Clojure-inspired *if* Macro

(if test then)

(if test then else)

If `test` evaluates to anything other than `'false` or `'undefined`, return `then`, otherwise `else`, if given, else `'undefined`.

Function Composition

(comp f g)

Right to left function composition.

(comp fs x)

Compose a list of functions `fs`, right to left, and apply the resulting function to `x`.

(comp f g x)

Equivalent to `(funcall (comp f g) x)`.

(comp fs)

Compose a list of functions `fs` from right to left.

(comp)

Equivalent to `#'identity/1`.

Usage

The following examples assume `#'1+/1` is defined:

```
> (defun 1+ (x) (+ x 1))
1+
```

```
> (funcall (clj:comp #'math:sin/1 #'math:asin/1) 0.5)
0.49999999999999994
> (funcall (clj:comp (list #'1+/1 #'math:sin/1 #'math:asin/1) 0.5))
1.5
```

Or used in another function call:

```
> (lists:filter (clj:comp #'not/1 #'zero?/1)
    '(0 1 0 2 0 3 0 4))
(1 2 3 4)
```

The usage above is best when **comp** will be called by higher-order functions like **lists:foldl/3** or **lists:filter/2**, etc. However, one may also call **comp** in the following manner, best suited for direct usage:

```
> (clj:comp #'math:sin/1 #'math:asin/1 0.5)
0.49999999999999994
> (clj:comp (list #'1+/1 #'math:sin/1 #'math:asin/1) 0.5)
1.5
```

Partial Application

(partial f args)

(partial f arg-1)

Partially apply `f` to a given argument `arg-1` or list of `args`.

Usage

```
> (set f (clj:partial #'+/2 1))
#Fun<clj.3.121115395>
> (funcall f 2)
3
> (set f (clj:partial #'+/3 1))
#Fun<clj.3.121115395>
> (funcall f '(2 3))
6
> (set f (clj:partial #'+/3 '(2 3)))
#Fun<clj.3.121115395>
> (funcall f 4)
9
> (set f (clj:partial #'+/4 '(2 3)))
#Fun<clj.3.121115395>
> (funcall f '(4 5))
14
```

Note that to partially apply a function that expects a list, you must wrap said list into a (singleton) list.

```
> (set double (clj:partial #'*/2 2))
#Fun<clj.5.16146786>
> (set f (clj:partial #'lists:map/2 double))
#Fun<clj.5.16146786>
> (funcall f '((1 2 3)))
(2 4 6)
```

Predicate Functions

N.B. These functions may *not* be used in guards.

(string? data)

Return `'true` if `data` is a flat list of printable characters.

(unicode? data)

Return `'true` if `data` is a flat list of printable Unicode characters.

(list? data)

Return `'true` if `data` is a list and not a string.

(set? data)

Return `'true` if `data` is appears to be a (possibly ordered) set.

(dict? data)

Return `'true` if `data` is a dictionary.

(proplist? lst)

Return `'true` if `lst` is a list where **proplist-kv?/1** returns `'true` for all elements in `lst`.

(proplist-kv? data)

Return `'true` if a `data` is a key/value tuple or an atom.

(queue? x)

Return `'true` if `x` is a queue.

(empty? x)

Return `'true` if `x` is the empty list, tuple, map, dictionary, queue, or general balanced tree.

(every? pred lst)

(all? pred lst)

Return `'true` if `(pred x)` returns `'true` for every `x` in `lst`.

(any? pred lst)

Return `'true` if `(pred x)` returns `'true` for any `x` in `lst`.

(not-any? pred lst)

Return `'false` if `(pred x)` returns `'true` for any `x` in `lst`.

(element? elem data)

Return `'true` if `elem` is an element of `data`, where `data` is a list, set or ordset.

Sequence Functions

(seq end)

Equivalent to `(seq 1 end)`.

(seq start end)

Equivalent to `(seq start end 1)`.

(seq start end step)

Return a sequence of integers, starting with `start`, containing the successive results of adding `step` to the previous element, until `end` has been reached or password. In the latter case, `end` is not an element of the sequence.

(next func)

Equivalent to `(next func 1 1)`.

(next func start)

Equivalent to `(next func start 1)`.

(next func start step)

Return a nullary function that returns a cons cell with `start` as the head and a nullary function, `(next func (funcall func start step) step)` as the tail. The result can be treated as a (possibly infinite) lazy list, which only computes subsequent values as needed.

(lazy-seq seq)

Return a lazy sequence (possibly infinite) from given lazy sequence `seq` or finite lazy sequence from given list `seq`. Lazy sequence is treated as finite if at any iteration it produces empty list instead of data as its head and nullary function for next iteration as its tail.

(cycle lst)

Return a lazy infinite sequence with all elements from a given list `lst` or another lazy sequence cycled.

See **next/3** for details on the structure.

(range)

Equivalent to `(range 1 1)`.

(range start)

Equivalent to `(range start 1)`.

(range start step)

Return a lazy list of integers, starting with `start` and increasing by `step`. Equivalent to `(next #'+/2 start step)`. See also: **next/3**.

(drop n lst)

(drop 'all lst)

Return a list of all but the first `n` elements in `lst`. If `n` is the atom `all`, return the empty list.

(take n lst)

(take 'all lst)

Given a (possibly lazy) list `lst`, return a list of the first `n` elements of `lst`, or all elements if there are fewer than `n`. If `n` is the atom `all` and `lst` is a "normal" list, return `lst`.

(split-at n lst)

Return a tuple of `` `#(,(take n lst) ,(drop n lst)) ``.

(partition n lst)

Equivalent to `(partition n n lst)`.

(partition n step lst)

Equivalent to `(partition n step () lst)`.

(partition n step pad lst)

Return a list of lists of `n` items each, at offsets `step` apart. Use the elements of `pad` as necessary to complete the last partition up to `n` elements. In case there are not enough padding elements, return a partition with less than `n` items.

(partition-all n lst)

Equivalent to `(partition-all n n lst)`.

(partition-all n step lst)

Return a list of lists like **partition/3**, possibly including partitions with fewer than `n` elements at the end.

(interleave list-1 list-2)

Return a list of the first element of each list, then the second, etc.

(get-in data keys)

Equivalent to `(get-in data keys 'undefined)`.

(get-in data keys not-found)

Return the value in a nested associative structure, where `keys` is a list of keys or list indices. Return the atom `not-found` if the key is not present or index is out of bounds, or the `not-found` value.

(reduce func (cons head tail))

Equivalent to `(reduce func head tail)`.

(reduce func acc lst)

Equivalent to `(lists:foldl func acc lst)`.

(repeat x)

Return a lazy infinite sequence of `xs`.

See **next/3** for details on the structure.

(repeat n f)

Given a nullary function `f`, return a list of `n` applications of `f`.

(repeat n x)

Given a term `x`, return a list of `n` copies of `x`.

Other Functions

(identity x)

Identity function.

(constantly x)

Return a unary function that returns `x`. N.B. This is like Haskell's `const` rather than Clojure's `constantly`.

(inc x)

Increment `x` by 1.

(dec x)

Decrement `x` by 1.