Eulerlisp Language Documentation

[2019-08-22 Thu 11:55]

Contents

1 Cons & Lists

`(cons a b)` creates a pair (cons-cell) `(a . b)`.

- `(fst c)` can be used to extract the first element,
- `(rst c)` for the second element.
- `(set-fst! c v)` sets the first element of pair `c` to `v`
- `(set-rst! c v)` sets the second element of pair `c` to `v`

Combinations of `fst` and `rst` are available for up to four levels of nesting, e.g. `(rrfrst c)` can be used instead of `(rst (rst (fst (rst c))))`.

A **list** is made up of nested cons-cells where the `rst` of the last one is `nil`.

- `(list a b c)` creates a list `(cons a (cons b (cons c nil)))`
- `(length l)` returns the length of list `l`
- `(reverse l)`
- `(range from to)` generates a list of **ascending** numbers `(from from+1 ... to)`
- `(any? pred lst)` returns `#t` if `(pred e)` is true for any element `e` of `lst`
- `(all? pred lst)`
- `(none? pred lst)`
- TODO `(count pred lst)`
- TODO `(map f lst)`

- TODO `(flatmap f lst)`
- TODO `(select f lst)`
- TODO `(reject f lst)`
- TODO `(transpose lst)`
- TODO `(reduce f acc lst)`
- TODO `(reduce-sum f lst)` (if the list is empty, 0 is returned)
- TODO `(reduce-product f lst)` (if the list is empty, 1 is returned)
- TODO `(reduce-min f lst)` (if the list is empty, `nil` is returned)
- TODO `(reduce-max f lst)` (if the list is empty, `nil` is returned)
- TODO `(append lst1 lst2)`
- TODO `(nth n lst)`
- TODO `(last n)`
- TODO `(delete e lst)`
- TODO `(delete-nth n lst)`
- TODO `(max-by)`
- TODO `(min-by)`
- TODO `(take n lst)`
- TODO `(zip lsts)`
- TODO `(windows n lst)`
- TODO `(chunks n lst)`
- TODO `(flatten lst)`
- TODO `(each f lst)`
- TODO `(each-with-index f lst)` Parameters to `f`: current element, index
- TODO `(concat lst)`
- TODO `(enumerate lst)` Turns `lst` into a list of pairs `(index . element)`

2 Vectors

- TODO `vector-init!`
- TODO `vector-swap!`
- TODO `vector-add!`
- TODO `vector-sub!`

3 Operations on Ranges

- TODO `range-each`
- TODO `reverse-range-each`
- TODO `range-count`
- TODO `range-max`
- TODO `range-min`
- TODO `range-first`
- TODO `sum`
- TODO `product`

4 Definitions & Assignment

4.1 Variable Assignment

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (def name var)

4.2 Constant Assignment

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defconst name constant)

The difference between variables and constants is that the latter can't change during runtime and are constant-folded if possible.

4.2.1 TODO Example for constant folding

4.2.2 TODO Benchmark comparing variables & constants

4.3 Function Definitions

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defn name args body...)

`defn` is implemented as a macro that expand to language=Lisp,label= ,caption= ,captionpos=b,numbers=none (def name (fn args body...))

4.4 `let` and `let*`

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (let ((var1 val1) (var2 val2) ...) body...)

`let` evaluates `body` in an environment where `varN` is bound to `valN`. All `valN` are evaluated in the same environment as the parent `let`.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (let* ((var1 val1) (var2 val2) ...) body...) is equivalent to writing language=Lisp,label= ,caption= ,captionpos=b,numbers=none (let ((var1 val1)) (let ((var2 val2)) body...))

Each `valN` is evaluated in an environment where the previous `varN` are already bound.

5 Flow Control, Conditionals

5.1 If, Unless, When

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (if condition consequence) (if condition consequence alternative)

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (unless condition consequence) (unless condition consequence alternative) is equivalent to language=Lisp,label= ,caption= ,captionpos=b,numbers=none (if (not condition) consequence alternative)

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (when condition consequence...) is equivalent to language=Lisp,label= ,caption= ,captionpos=b,numbers=none (if condition (begin consequence...))

If no `alternative` is provided, `nil` is returned if the `condition` is false.

5.2 Cond

The branches of a conditional can have multiple forms. Below each form is used once for demonstrative purposes. Of course multiple branches can have the same form.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (cond (test1 result1...) (test2 => result2) (test3) (else result4))

If `test1` is true, the expressions `result1...` are evaluated and the value of the last one is returned.

If `test2` is true, `(result2 test2)` is returned.

If `test3` is true, the value it evaluates to is returned.

Otherwise `result4` is returned.

If no `else` branch is defined and all tests evaluate to false, `nil` is returned.

5.3 Case

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (case key (key1 result1) (key2 result2) (else result3))

If key is not an atom, it is evaluated first. Then it is compared to the keys of each branch using `equal?` and the result of the first matching branch is returned.

6 Threading Macro

The threading macro "folds" its argument by inserting the them as the **last** argument of the following expression.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (> 1 (+ 5) square println)

Is the same as writing

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (println (square (+ 5 1)))

7 Functions

7.1 Function Composition

`comp` implements function composition. `(comp h g f)` returns a function `(fn (x) (h (g (f x))))`.