C-lisp - Lisp REPL written in C

Release 0.6.0

Alexandre Barbieri

Jan 24, 2021

CONTENTS

CHAPTER

ONE

ABOUT

Do not expect great implementations or even a language that can be used in daily basis.

This project is about: **"How can I implement a REPL?"**, so please consider this as a toy project.

1.1 Why lisp?

Lisp is much more easier than you think. Take a look some concepts behind this language:

- Polish Notation
- Weak types
- Immutable objects
- Functional Programming

CHAPTER

TWO

REFERENCE

Basic reference manual of functions supported.

2.1 Boolean

(bool? v)
Check if value is boolean

```
> (bool? #t)
#t
> (bool? 1)
#f
```

(eq v1 v2)
Compare equality between two values

```
> (eq 1 2)
#f
> (eq 1 1)
#t
> (eq 1 #t)
#f
```

(ne v1 v2)
Compare inequality between two values

```
> (ne 1 2)
#t
> (ne 1 1)
#f
> (ne 1 #t)
#t
```

(not v)
Inverse of boolean value

```
> (not #t)
#f
> (not #f)
#t
> (not 1)
Error: Wrong argument type. Got: Number, Expected: Boolean
```

(and v1 v2)
Logical and operation

```
> (and #t #t)
#t
> (and #f #t)
#f
> (and #t 1)
Error: Wrong argument type. Got: Number, Expected: Boolean
```

(or v1 v2)
Logical or operation

```
> (or #t #t)
#t
> (or #f #f)
#f
> (or #t 1)
Error: Wrong argument type. Got: Number, Expected: Boolean
```

(xor v1 v2)
Logical xor operation

```
> (xor #t #t)
#f
> (xor #t #f)
#t
> (xor #t 1)
Error: Wrong argument type. Got: Number, Expected: Boolean
```

(nand v1 v2)
Logical nand operation

```
> (nand #t #t)
#f
> (nand #f #f)
#t
> (nand #t 1)
Error: Wrong argument type. Got: Number, Expected: Boolean
```

(nor v1 v2)
Logical nor operation

```
> (nor #t #t)
#f
> (nor #f #f)
#t
> (nor #t 1)
Error: Wrong argument type. Got: Number, Expected: Boolean
```

2.2 Error

(error? v)
Check if the value is an error

```
> (error? (error "Some error"))
#t
> (error? #t)
#f
```

(error message)
Create error object

```
> (error "Some error")
Error: Some error
> (error 1)
Error: Wrong argument type. Got: Number, Expected: String
```

2.3 Input/Output

(display t)
Display info about type

```
> (display 1)
Number -> 1
> (display [1 2 3 4])
List -> [1 2 3 4]
```

(print t)
Print type value

```
> (print 1)
1
> (print [1 2 3 4])
[1 2 3 4]
> (print +)
<builtin>
```

(load file)
Load clisp script

```
> (load "stl/prelude.clisp")
> (pi)
3.1415
```

2.4 List

(list? v)
Check if value is a list

```
> (list? [1 2 3 4])
#t
> (list? 1)
#f
```

(list e1 ...)
Create a list of elements

```
> (list 1 2 3 4 5)
[1 2 3 4 5]
```

(head xs)
Get the first element of a list

```
> (head [1 2 3 4 5])
1
> (head 1)
Error: Wrong argument type. Got: Number, Expected: List
```

(tail xs)
Discard the first element of a list

```
> (tail [1 2 3 4])
[2 3 4]
> (tail 1)
Error: Wrong argument type. Got: Number, Expected: List
```

(append xs el)
Append an element in the list

```
> (append [1 2 3 4] 5)
[1 2 3 4 5]
> (append 1 1)
Error: Wrong argument type. Got: Number, Expected: List
```

(length xs)
Get the number of elements in the list

```
> (length [1 3 4])
3
> (length 1)
Error: Wrong argument type. Got: Number, Expected: List
```

(empty? xs)
Check if the list is empty

```
> (empty? [])
#t
> (empty? [1])
#f
> (empty? 1)
Error: Wrong argument type. Got: Number, Expected: List
```

(cons el xs)
Get two elements and create one list

```
> (cons 1 2)
[1 2]
> (cons 1 [2 3])
[1 2 3]
```

2.5 Number

(number? v)
Check if value is number

```
> (number? 1)
#t
> (number? #t)
#f
```

(zero? v)
Check if value is equal to zero

```
> (zero? 0)
#t
> (zero? 1)
#f
> (zero? #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(positive? v)
Check if value is positive

```
> (positive? -1)
#f
> (positive? 1)
#t
> (positive? #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(negative? v)
Check if value is negative

```
> (negative? 1)
#f
> (negative? -1)
#t
> (negative? #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(even? v)
Check if value is even

```
> (even? 1)
#f
> (even? 2)
#t
```

(continues on next page)

(continued from previous page)

```
> (even? #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(odd? v)

Check if value is odd

```
> (odd? 1)
#t
> (odd? 2)
#f
> (odd? #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(> v1 v2)

Check if v1 is greater than v2

```
> (> 1 2)
#f
> (> 2 1)
#t
> (> 1 #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(>= v1 v2)

Check if v1 is greater or equal v2

```
> (>= 1 2)
#f
> (>= 2 2)
#t
> (>= 1 #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(< v1 v2)

Check if v1 is lesser than v2

```
> (< 1 2)
#t
> (< 2 2)
#f
> (< 1 #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(<= v1 v2)

Check if v1 is lesser or equal v2

```
> (<= 1 2)
#t
> (<= 2 2)
#t
> (<= 1 #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

2.6 Math

(+ v1 v2)
Sum two numbers

```
> (+ 1 2)
3
> (+ 1 #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(- v1 v2)
Subtract two numbers

```
> (- 2 1)
1
> (- 1 #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(* v1 v2)
Multiply two numbers

```
> (* 2 1)
2
> (* 1 #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(/ v1 v2)
Divide two numbers

```
> (/ 2 1)
2
> (/ 1 #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(% v1 v2)
Rest of division

```
> (% 10 3)
1
> (% 1 #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(^ v1 v2)
Power of a number

```
> (^ 10 3)
1000
> (^ 1 #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(ceil v)
Round number to ceiling

```
> (ceil 2.1)
3
> (ceil #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(floor v)
Round number to floor

```
> (floor 2.9)
2
> (floor #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(round v)
Round number

```
> (round 2.3)
2
> (round 2.7)
3
> (round #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(sqrt v)
Square root

```
> (sqrt 4)
2
> (sqrt #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(log10 v)
Logarithm base 10

```
> (log10 1000)
3
> (log10 #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(log v)
Natural logarithm

```
> (log 1)
0
> (log #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(sin v)
Sine of a radian

```
> (sin 3.1415)
0
> (sin #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(cos v)
Cosine of a radian

```
> (cos 3.1415)
1
> (cos #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(tan v)
Tangent of a radian

```
> (tan 0.7854)
1
> (tan #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

(abs v)
Absolute value of a number

```
> (abs -1)
1
> (abs 1)
1
> (abs #t)
Error: Wrong argument type. Got: Boolean, Expected: Number
```

2.7 String

(string? v)
Check if value is a string

```
> (string? "Value")
#t
> (string? 1)
#f
```

(string->upper v)
Transform all characters in uppercase

```
> (string->upper "Value")
VALUE
> (string->upper 1)
Error: Wrong argument type. Got: Number, Expected: String
```

(string->lower v)
Transform all characters in lowercase

```
> (string->lower "Value")
value
> (string->lower 1)
Error: Wrong argument type. Got: Number, Expected: String
```

(string->split v s)
Split a phrase in a list of words separated by element

```
> (string->split "Hello World" " ")
["Hello" "World"]
> (string->split 1 " ")
Error: Wrong argument type. Got: Number, Expected: String
```

(string->concat el ...)
Concat all elements in a single string

```
> (string->concat "Hello " "World" " ...")
"Hello World ..."
> (string->concat 1 " ")
Error: Wrong argument type. Got: Number, Expected: String
```

(string->length str)
Count the number of characters in the string

```
> (string->length "Hello World")
11
> (string->length 1)
Error: Wrong argument type. Got: Number, Expected: String
```

2.8 Syntactic

(if c e1 e2)
Conditional that executes first expression if conditional is true, else it executes second expression

```
> (if (> 2 1) "foo" "bar")
foo
> (if (< 2 1) "foo" "bar")
bar
> (if 1 "foo" "bar")
Error: Wrong argument type. Got: Number, Expected: Boolean
```

(cond expr1 ...)
Cond clause test expression by expression and execute if condition is true

```
> (cond ((> 2 1) "foo") (#t 2))
foo
> (cond ((< 2 1) "foo") (#t "bar"))
bar
```

(for it command)
Iterate over a list and execute the command

```
> (for (i [1 2 3]) (print i))
1
2
3
> (for i (print i))
Error: Wrong argument type. Got: Symbol, Expected: Expression
```

(def (name p1 ..) body)
Define a function

```
> (def (add1 x) (+ x 1))
> (add1 1)
2
```

(fn args body)
Create a lambda function to pass as parameter

```
> (filter (fn (a) (> a 4)) [3 4 5 6])
[5 6]
```

(when expr1 ...)

Execute all expressions that conditional is true

```
> (when (#t (print "foo")) (#f (print "foo")) (#t (print "bar")))
foo
bar
```

(unless expr1 ...)

Execute all expressions that conditional is false

```
> (unless (#t (print "foo")) (#f (print "foo")) (#t (print "bar")))
foo
```

(type v)

Show the type of value

```
> (type 1)
Number
```

CHAPTER

THREE

STANDARD LIBRARY

3.1 Prelude

Helper functions written in clisp

3.1.1 Math

```
; Math
(def (pi) 3.1415)
(def (add1 z) (+ z 1))
(def (sub1 z) (- z 1))
(def (sign x) (cond ((positive? x) 1) ((negative? x) -1) (#t 0)))
(def (double b) (* b 2))
(def (triple b) (* b 3))
(def (fib n) (cond ((< n 0) 0) ((eq n 1) 1) (#t (+ (fib (- n 1)) (fib (- n 2))))))
(def (fact n) (if (<= n 0) 1 (* n (fact (- n 1)))))
```

3.1.2 List

```
; List
(def (reverse xs) (if (empty? xs) [] (append (reverse (tail xs)) (head xs))))
(def (nth n xs) (cond ((empty? xs) []) ((<= n 1) (head xs)) (#t (nth (- n 1) (tail
↪xs)))))
(def (last xs) (nth (length xs) xs))
(def (first xs) (nth 1 xs))
(def (remove v xs) (filter (fn (t) (ne t v)) xs))

(def (map f xs) (if (empty? xs) [] (append (list (f (head xs))) (map f (tail xs)))))
(def (fold f acc xs) (cond ((empty? xs) acc) (#t (fold f (f acc (head xs)) (tail
↪xs)))))
(def (reduce f xs) (if (empty? xs) (head xs) (fold f (head xs) (tail xs))))
(def (filter f xs) (cond ((empty? xs) []) ((f (head xs)) (append (list (head xs))
↪(filter f (tail xs)))) (#t (filter f (tail xs)))))
(def (for-each f xs) (cond ((> (length xs) 0) (when (#t (f (head xs))) (#t (for-each
↪f (tail xs)))))))

(def (flatten xs) (cond ((empty? xs) []) ((list? (head xs)) (append (flatten (head
↪xs)) (flatten (tail xs)))) (#t (append (list (head xs)) (flatten (tail xs))))))
(def (zip xs ys) (cond ((empty? xs) []) ((empty? ys) []) (#t (append (list (list
↪(head xs) (head ys))) (zip (tail xs) (tail ys))))))
```

(continues on next page)

(continued from previous page)

```
(def (range s e) (cond ((> s e) []) (#t (append (list s) (range (+ s 1) e)))))
(def (range-step s e st) (cond ((> s e) []) (#t (append (list s) (range-step (+ s st)␣
↪e st)))))
(def (repeat t n) (map (fn (a) t) (range 1 n)))
(def (all xs) (if (empty? xs) #f (fold and (head xs) (tail xs))))
(def (any xs) (if (empty? xs) #f (fold or (head xs) (tail xs))))

(def (take n xs) (cond ((empty? xs) []) ((eq n 0) []) (#t (append (list (head xs))␣
↪(take (- n 1) (tail xs))))))
(def (drop n xs) (cond ((empty? xs) []) ((eq n 0) (append (list (head xs)) (drop 0␣
↪(tail xs)))) (#t (drop (- n 1) (tail xs)))))
(def (takeWhile f xs) (cond ((empty? xs) []) ((f (head xs)) (append (list (head xs))␣
↪(takeWhile f (tail xs)))) (#t [])))
(def (dropWhile f xs) (cond ((empty? xs) []) ((f (head xs)) (dropWhile f (tail xs))) (
↪#t xs)))
(def (takeFirst f xs) (cond ((empty? xs) []) ((f (head xs)) (head xs)) (#t (takeFirst␣
↪f (tail xs)))))
(def (takeLast f xs) (takeFirst f (reverse xs)))
(def (slice s e xs) (take e (drop s xs)))

(def (sum xs) (reduce + xs))
(def (multiply xs) (reduce * xs))
(def (max xs) (reduce (fn (acc v) (if (>= acc v) acc v)) xs))
```

3.2 Unit test

Helper functions to write your own tests

```
(def (check-eq? f s m) (cond ((ne f s) (error m))))
(def (check-not-eq? f s m) (cond ((eq f s) (error m))))
(def (check-true? b m) (cond ((ne b #t) (error m))))
(def (check-false? b m) (cond ((ne b #f) (error m))))
(def (check-count? xs c m) (cond ((ne (length xs) c) (error m))))
(def (check-eqv? f s e m) (cond ((> (abs (- f s)) e) (error m))))

(def (check-bool? t1 m) (cond ((not (bool? t1)) (error m))))
(def (check-error? t1 m) (cond ((not (error? t1)) (error m))))
(def (check-list? t1 m) (cond ((not (list? t1)) (error m))))
(def (check-number? t1 m) (cond ((not (number? t1)) (error m))))
(def (check-string? t1 m) (cond ((not (string? t1)) (error m))))
(def (check-type? t1 t2 m) (cond ((ne (type t1) t2) (error m))))
```

3.2.1 Examples

```
(load "stl/unittest.clisp")

(test-suite "Boolean suite case")

; Check type
(test-case "Test Bool? without args" (check-error? (bool?) "Arg should be error"))
(test-case "Test Bool? using boolean" (check-true? (bool? #f) "Arg should be boolean
↪"))
(test-case "Test Bool? using number" (check-false? (bool? 1) "Arg should be boolean"))
```