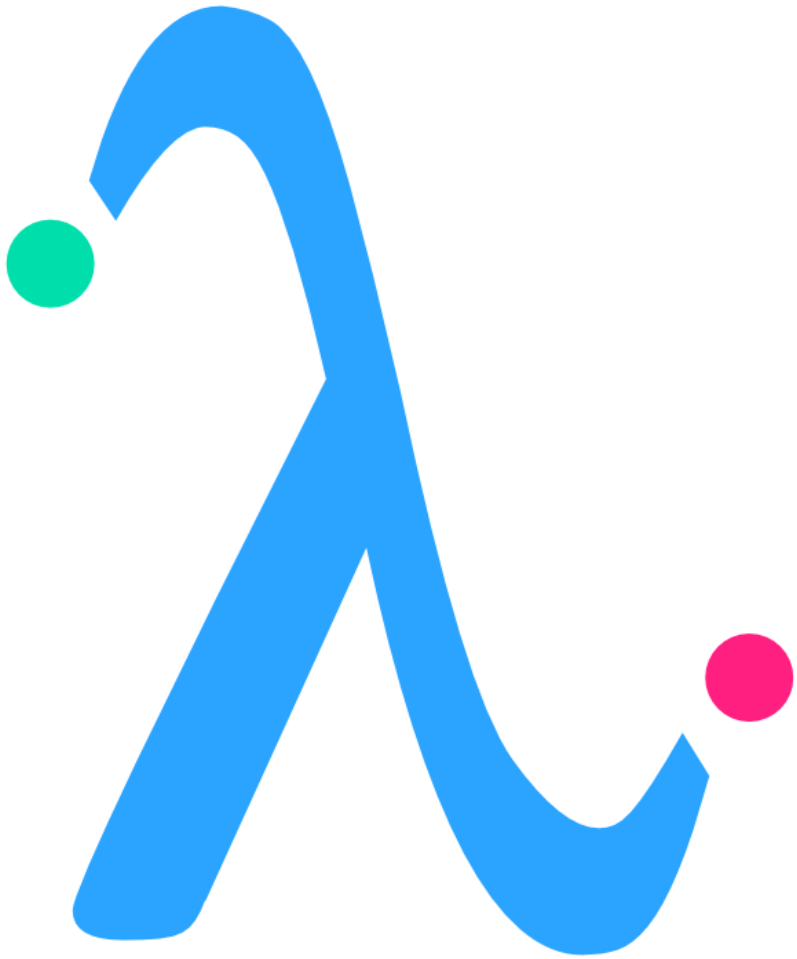
Reference Manual

# My Lisp 2.14



A Lisp interpreter

**Preface** ---------- **8**
*Overview* ... *8*
*References* ... *9*
*Limitations* ... *10*
**Presentation** ---------- **11**
*Lisp expressions* ... *11*
*Floats, integers, and rationals* ... *14*
*The interpreter* ... *16*
*Dynamic binding* ... *25*
*Startup* ... *28*
*Common questions* ... *30*
**McCarthy** ---------- **31**
*quote* ... *32*
*atom* ... *33*
*eq* ... *34*
*car* ... *35*
*cdr* ... *36*
*cons* ... *37*
*cond* ... *38*
*lambda* ... *39*
*label* ... *40*
**Core** ---------- **41**
*define* ... *42*
*eval* ... *45*
*if* ... *46*
*list* ... *47*
*load* ... *48*
*progn* ... *49*
*set!* ... *50*
*while* ... *51*
*error* ... *52*
*println* ... *53*
*print* ... *54*
*read* ... *55*
*read-string* ... *56*
*eqv?* ... *57*
*integer?* ... *58*
*list?* ... *59*
*null?* ... *60*
*number?* ... *61*
*procedure?* ... *62*
*rational?* ... *63*

*string?* ..... *64*
*string->code* ..... *65*
*string<-code* ..... *66*
*string-index* ..... *67*
*string-length* ..... *68*
*string->list* ..... *69*
*string->lower* ..... *70*
*string->number* ..... *71*
*string->string* ..... *72*
*string->symbol* ..... *73*
*string->upper* ..... *74*
*->string* ..... *75*
*->float* ..... *76*
*->integer* ..... *77*
*->rational* ..... *78*
*->denominator* ..... *79*
*->numerator* ..... *80*
*nan?* ..... *81*
*=* ..... *82*
*<* ..... *83*
*<=* ..... *84*
*>=* ..... *85*
*>* ..... *86*
*+* ..... *87*
*** ..... *88*
*-* ..... *89*
*/* ..... *90*
*abs* ..... *91*
*ceiling* ..... *92*
*floor* ..... *93*
*round* ..... *94*
*truncate* ..... *95*
*modulo* ..... *96*
*quotient* ..... *97*
*remainder* ..... *98*
*sqrt* ..... *99*
*expt* ..... *100*
*exp* ..... *101*
*log* ..... *102*
*pi* ..... *103*
*degrees->radians* ..... *104*
*radians->degrees* ..... *105*
*cos, sin, tan* ..... *106*
*acos, asin, atan* ..... *107*
*real-mode* ..... *108*

*complex-mode* ..... 109
*->complex* ..... 110
*roots* ..... 111
*solve* ..... 112
*integ* ..... 113
*date* ..... 114
*seconds-from-epoch* ..... 115
**Tools.lisp ..... 116**
*apply* ..... 117
*mapcar* ..... 118
*maplist* ..... 119
*evcar* ..... 120
*let* ..... 121
*let** ..... 122
*append* ..... 123
*last* ..... 124
*length* ..... 125
*list->string* ..... 126
*make-list* ..... 127
*nth* ..... 128
*nthcdr* ..... 129
*reverse* ..... 130
*subst* ..... 131
*subst** ..... 132
*unless* ..... 133
*when* ..... 134
*repeat* ..... 135
*repeat-eval* ..... 136
*and* ..... 137
*not* ..... 138
*or* ..... 139
*member?* ..... 140
*comment* ..... 141
**Math.lisp ..... 142**
*0?* ..... 143
*1?* ..... 144
*zero?* ..... 145
*even?* ..... 146
*odd?* ..... 147
*1+* ..... 148
*1-* ..... 149
*fact* ..... 150
*fib* ..... 151
*egcd* ..... 152

*gcd* ..... *153*
*gcd_abs* ..... *154*
*lcm* ..... *155*
*prime?* ..... *156*
**Rationals.lisp** ----- **157**
*rational->decimal-string* ..... *158*
*rational>decimal-list* ..... *159*
**Modulus.lisp** ----- **160**
*modulus::add* ..... *161*
*modulus::sub* ..... *162*
*modulus::mul* ..... *163*
*modulus::div* ..... *164*
*modulus::expt* ..... *165*
*modulus::inverse* ..... *166*
**Primes.lisp** ----- **167**
*prime-numbers::known-primes* ..... *168*
*prime-numbers::prime-to-known?* ..... *169*
*prime-numbers::prime?* ..... *170*
*prime-numbers::nth* ..... *171*
*prime-numbers::primorial* ..... *172*
*prime-numbers::product* ..... *173*
*prime-numbers::factors* ..... *174*
*prime-numbers::factors** ..... *175*
*prime-numbers::divisors* ..... *176*
*prime-numbers::practical?* ..... *177*
*prime-numbers::find-sum-from-divisors* ..... *178*
*prime-numbers::find-sum-from-divisors** ..... *179*
*prime-numbers::find-sum-from-list* ..... *180*
*prime-numbers::find-sum-from-list** ..... *181*
**Continued fractions.lisp** ----- **182**
*rational->continued-fraction* ..... *183*
*rational<-continued-fraction* ..... *184*
*quadratic->continued-fraction* ..... *185*
**Egyptian fractions.lisp** ----- **186**
*rational->egyptian-fraction* ..... *187*
*rational->egyptian-fraction::fibonacci* ..... *188*
*rational->egyptian-fraction::golomb* ..... *189*
*rational->egyptian-fraction::splitting* ..... *190*
*rational->egyptian-fraction::binary* ..... *191*
*rational->egyptian-fraction::primorial* ..... *192*
*rational->egyptian-fraction::erdos* ..... *193*
*rational->egyptian-fraction::practical* ..... *194*
*rational<-egyptian-fraction* ..... *195*
**Le_Lisp.lisp** ----- **196**

*closure* ..... *197*
*gensym* ..... *202*
**Lambda Calculus.lisp ----------203**
*Lambda expressions* ..... *204*
*Combinators* ..... *207*
*Church encoding* ..... *209*
*Utilities* ..... *210*
**Turtle graphics----------211**
*turtle::arc* ..... *214*
*turtle::background-color* ..... *215*
*turtle::backward* ..... *216*
*turtle::forward* ..... *217*
*turtle::heading* ..... *218*
*turtle::home* ..... *219*
*turtle::left* ..... *220*
*turtle::move-to* ..... *221*
*turtle::name* ..... *222*
*turtle::pen-color* ..... *223*
*turtle::pen-down* ..... *224*
*turtle::pen-up* ..... *225*
*turtle::pen-width* ..... *226*
*turtle::push* ..... *227*
*turtle::pop* ..... *228*
*turtle::reset* ..... *229*
*turtle::right* ..... *230*
*turtle::turn* ..... *231*
**Tracing & Debugging ----------232**
*sys::bindings* ..... *235*
*sys::bindings-names* ..... *236*
*sys::bindings-assoc* ..... *237*
*sys::clear-bindings* ..... *238*
*sys::debug* ..... *239*
*sys::error?* ..... *240*
*sys::print-values* ..... *241*
*sys::trace* ..... *242*
*sys::trace-mode* ..... *243*
*sys::untrace* ..... *244*
**Options----------245**
*options::integer-mode* ..... *246*
*options::integer-suffix* ..... *248*
*options::number-decimals* ..... *250*
*options::number-format* ..... *252*
*options::quote-as-quote* ..... *254*
*options::keyboard-mode* ..... *256*

**Last words... ----------257**

# Preface

## Overview

My Lisp is a complete and universal Lisp environment running directly on the iPhone, iPad, and Mac. This interpreter is true to the original John McCarthy Lisp implementation[1] with the fundamental 7 operators *quote*, *atom*, *eq*, *car*, *cdr*, *cons*, *cond*, along with *lambda* and *label*. My Lisp also contains core and mathematical operators borrowed from other Lisp dialects (Le Lisp, Lisp 1.5, MacLisp, Common Lisp, and Scheme to name a few) to make it easy to learn, program, and most importantly, enjoy Lisp. It also features built-in functions for advanced mathematics, including complex numbers and numerical analysis (roots and zeros finder, integral approximation), along with the classical LOGO turtle. The complete description of the fundamental, core, and built-in functions is available using a set of library functions completely written in My Lisp.

My Lisp offers an interpreter and an editor, all working on the iPhone, iPad, and Mac, and most importantly, without requiring any server connection, that is, the interpreter is executing locally on the iPhone, iPad, or Mac My Lisp is installed on.

Library and example files contain the source code of classical functions and problems solved by My Lisp and may be used as reference to learn Lisp and develop other programs. They include classical puzzles (hanoi and n-queens), basic mathematical functions (gcd, lcm, factorial, fibonacci, prime?), and the historical *apply*, *mapcar* and *maplist* functions. The Lambda Calculus example file contains various functions related to Lambda Calculus and Combinators, with alpha-conversion, beta-reduction, de Bruijn notations, etc. As a special note, the example file Symbolic Derivation contains a complete yet extensible symbolic derivation module allowing to compute the formal derivation of virtually any symbolic function expressed as a Lisp expression.

A user manual and a reference manual are available from within the application but also on My Lisp web site (https://lisp.lsrodier.net) and in Apple Books. The complete source code of the library and example files is part of My Lisp.

Last but not least, this overview couldn't end without a sample definition of the notorious REPL function:

```
(define (REPL eval_me) (REPL (println (eval (read)))))
```

---

[1] See "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I".

# References

My Lisp for iPhone, iPad, and Mac:

https://apple.co/33RPzGZ

AppStore

My Lisp user manual:

https://apple.co/3lEufLi

Apple Books

My Lisp home page:

https://lisp.lsrodier.net

Home Page

My Lisp reference manual:

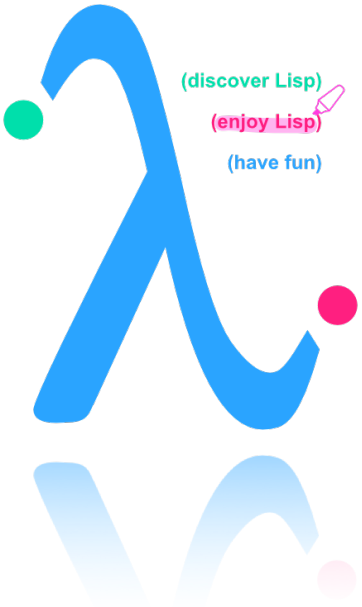
https://apple.co/3ejJSW2

Apple Books

# Limitations

This manual is neither an introduction to Lisp nor a complete guide on how-to program in Lisp. It does however contain the full description of the built-in and library functions of My Lisp that are also available within the help file of My Lisp, and has many examples that may be useful when learning Lisp.

A reference manual is not that useful without practice. Thus, you are encouraged to look at the example files of My Lisp because they provide actual usage of the engine and core functions. In particular the SICP file with many examples from the SICP book ; it is also a good reference to cope with the differences between My Lisp and Scheme, especially when it comes to lexical versus dynamic bindings.



# Presentation

## Lisp expressions

### Syntax

An expression is either an atom or a list of zero or more expressions, separated by whitespace(s) and enclosed by parentheses.

An atom is either a symbol name, a number, or a string where a string is a sequence of characters enclosed within the double-quote character ". A symbol name is a sequence of characters without any specific limitation (aside whitespaces and parentheses), that is non “conventional” characters may be used to compound names like <, =, $, ?, or 😃.

Notes:

- The . character may be interpreted as a special character when parsing dotted pairs or the arguments list of a *define* or *lambda* definition.
- The ? character may be interpreted as a special character when reading the names of the formal arguments of a *define* or *lambda* definition. See the *define* and *lambda* functions for further details.

Here are a few examples of atoms that are also symbols:

| | | |
|---|---|---|
| **Hello** | **Hello-World** | **Bonjour-le-Monde** |
| **IsNull?** | **Lisp1.5::null** | **Incr!** |
| **<=** | **==** | **++** |
| **€** | **$** | **£** |

Here are a few examples of atoms that are also numbers:

| | | |
|---|---|---|
| **42.0** | **+42.0** | **-42.0** |
| **3.14** | **1.66e-27** | **6.022e23** |

Here are a few examples of atoms that are also integer numbers:
42 +4212345678901234 -42

Here are a few examples of atoms that are also rational numbers:
42/67 +1/123456789012345 -42/67

Here are a few examples of atoms that are also strings (in their input format as opposed to their printed format):
“A string” “Another \”string\”” “Back\\slash”

Here are a few examples of lists:
() (This is a list) (A B (a nested list) E)

## Pairs

Historically, the dotted pairs were the base elements after which the lists were built. My Lisp acknowledges it and does support the dotted notation and the creation of dotted pairs using the following 3 rules:

(X . Y) = (cons ‘X ‘Y)

(car ‘(X . Y)) = X

(cdr ‘(X . Y)) = Y

When a list expression is parsed,

- If a dot is encountered before the last symbol, then the last two elements of the lists are combined into a dotted pair. That is, *(car ‘(A B . C))* is *A*, *(cdr ‘(A B . C))* is *(B . C)*, *(cadr ‘(A B . C))* is B, and *(cddr ‘(A B . C))* is *C*.
- If a dot is encountered before an open parenthesis then it is ignored. That is *(A . (B . (C . ())))* is the same as *(A B C)* which is the preferred printed form.
- If a dot is encountered in another situation, it is treated as a symbol.

The previous rules applied only when the dot character is separated from a symbol character by a blank character; for instance *(A.B)* is a list of one atom *A.B*.

✓ A dotted pair is built when parsing an expression with the previous rules or when cons’ing an expression to a symbol. Building a list with a dot like *(cons ‘A (cons ‘. ‘(B)))*

results in a list *(A . B)* made of 3 symbols and the dot character is intended as a regular symbol.

## Comments

Comments are introduced by a semi-colon ; and extend up to the end of the line. They are supported in both the editor and the console.

## Special interests

My Lisp is mainly case-insensitive and almost any Unicode character may be used as part of a symbol name except for parenthesis and blank characters. The mathematical operators (=, !=, <, <=, >=, and >) are the only ones taking into account the case of the characters when comparing strings:

```
? (eqv? "A" "a")
t

? (eqv? 'A 'a)
t

? (= "A" "a")
nil

? (= 'A 'a)
nil
```

# Floats, integers, and rationals

My Lisp supports floating point numbers based on IEEE- 754 double-precision numbers (64-bit base-2 format), along with so-called big integers and rationals which precision is limited only by the available memory.

The default behavior is to parse integer and rational strings as big integers, which in turn are automatically converted to floating point values when required by the evaluated functions. This implicit handling can be turned off or modified using the options::integer-mode function.

By default, 1.0 is parsed as a floating point value whilst 1 is an integer, and 123/237 a rational. You can also create a rational using an integer division as in (/ 123 237). Rationals are always internally represented in their reduced form, thus 123/237 is actually printed as 41/79. You can also print rationals using the function *rational->decimal-string* that prints the decimal expansion with options to identify the period within that expansion.

**? 123/237**

41/79

**? (rational->decimal-string 123/237)**

0.{5189873417721}

**? (rational->decimal-string 123/237 5)**

0.52898…

**? (rational->decimal-string 123/237 -20)**

0.51898734172215189873…

**? (rational->decimal-string 264669312513930434 5/842468587426513207 37)**

3.14159265358979323846264338327950288841...

See the rationals, egyptian fractions and continuous fractions files for examples of computing with integers and rationals.

When a function that requires floating point numbers is invoked with integers or rationals, the values are automatically converted (with a potential lose of precision implied by floating points). You can however force the conversion using the ->float function.

When mixing floating point and integer or rational numbers, the integer and rational numbers are automatically converted to floating point numbers. For instance (+ 1 3) evaluates to the integer 4 whilst (+ 1.0 3) evaluates to the floating point value 4.0. The same rule occurs when mixing rational and floating point numbers:

**? (+ 1 1/3)**
4/3

**? (+ 1.0 1/3)**
1.3333

**? (/ 1 3)**
1/3

**? (/ 1.0 3)**
0.3333

**? (+ 0.0 1/3)**
0.3333

The function *->rational* is converting an expression into a rational number; this comes handy when the exact representation of a floating point number is required; in such case, use the string representation of the floating point number as in *(->rational "12.3")* instead of *(->rational 12.3)* as in the later form the internal floating point representation 12.3 will have introduced an approximation.

# The interpreter

## Evaluation rules

My Lisp interpreter follows common rules for interpreting Lisp expressions, that is:

- An atom (number, string,...) evaluates to itself.
- A symbol evaluates to its associated expression with respect to the binding context.
- A list (F a b c...) is a function call and evaluates as follows:
  - If F is a built-in function then it is evaluated with a, b, c... as parameters. The values of the parameters are bound to the binding context but evaluated only if required by the evaluation of F.
  - If F is a library function, user-defined function, or lambda expression, then all parameters not marked as lazy are evaluated, all lazy parameters are bound to the binding context, and then F is evaluated. When F is a tail-recursive call then the binding context is the current one, a new empty one is created otherwise.
  - The evaluation is left-recursive, that is if F is a list then it is first evaluated. For instance, when evaluating *((if (> 0 1) + -) 3 2)*, the expression (if (> 0 1) + -) is first evaluated, resulting to the original expression now equivalent to *(- 3 2)* and evaluated to *1*.

A lazy parameter, introduced by the ? character as the leading character of a formal parameter name is not evaluated until after the function body explicitly requires it. Lazy parameters allow creating expressions that mimic the behavior of the built-in functions like *if* or *cond* by deferring the evaluation until explicitly required. See the *define* function for further details and examples on using lazy parameters.

Note that library functions, user-defined functions, and lambda expressions are nothing more than regular lists starting with the lambda symbol or a symbol evaluating to such lists. This means that My Lisp does not handle functions or lambda expressions as special objects nor uses hidden objects during the evaluation:

```
? ((append (lambda (x y)) '((* x y))) 6 7)
42
```

# define

The *define* function is an extension of *label* allowing to create symbols with a "permanent" association that is available after the expressions defining the symbols have been evaluated. It is used to define or change the expression associated to with a symbol within the current binding context, thus leaving unchanged other associations in other binding contexts.

## Expressions binding

The simple syntax to associate an expression to a symbol has the form *(define name expr)* which associates to the symbol *name* the result of the evaluation of *expr*.

```
? (define h2g2 42)
h2g2

? h2g2
42

? (define adder (lambda (x y) (+ x y)))
adder

? (adder 4 5)
9
```

## Functions binding

In order to simplify the declaration of lambda expressions, functions may also be defined using any these 2 extended syntaxes:

```
(define (name arg1 arg2 ...)
        body)

(define name (arg1 arg2 ...)
        body)
```

which are both equivalent to:

```
(define name (lambda (arg1 arg2 ...) body))
```

Under these forms *body* may be one or more expressions. Upon evaluation of the function, the arguments *arg1*, *arg2*, ... are bound to their values in the current binding context and then *body* is evaluated.

When *body* is made of 2 or more expressions, it is automatically evaluated as part of a *progn* expression.

### Variable number of parameters

If an argument is the dot character then all arguments of the function call following the dot are merged into a list associated to the argument following the dot as in:

```
? (define (subto a . neg) (- a (apply + neg)))
subto

? (subto 10 1 2 3)
4
```

Note that the dot character may be the first argument if all parameters must be merged into a list upon invocation.

### Default values

A function argument may be expressed as a list of 2 elements *(name expr)* where *name* is a symbol corresponding to the name of the argument and *expr* the default value to pass for the argument if missing when the function is called; the *expr* is evaluated when the function is actually *defined* as opposed to when invoked:

```
? (define (add x (y 5)) (+ x y))
add

? (add 2)
7

? (add 2 3)
5
```

Upon a function call, if an argument is omitted, it is implicitly assumed with the default value *nil*.

If the *body* part of the function requires many expressions executed in sequence, you can use the implicit *progn* form:

```
(define (maclisp::append L1 . L2)
        (define (maclisp::append-helper A L)
                (if (null? L) A
                        (maclisp::append-helper (append A (car L)) (cdr L))))
        (maclisp::append-helper L1 L2)
)

? (maclisp::append '(a b c) '(d e f) nil '(g))
(a b c d e f g)
```

### Lazy parameters

When the name of the argument of a function or lambda starts with the ? character then it is considered a lazy parameter: upon evaluation of the function, the expression of the function call is not evaluated and passed as it; if the value is necessary, then it is up to the function body to evaluate the parameter using *eval*. Lazy parameters allows creating functions like *if* or *cond* that evaluate their arguments by necessity, thus working around macros..

Examples:

```
? (define (_if test ?yes ?no) (if test ?yes ?no))
_if

? (_if 't (+ 3 4) (* 3 4))"
(+ 3 4)

? (_if '() (+ 3 4) (* 3 4))
(* 3 4)

? (define (_if test ?yes ?no) (if test (eval ?yes) (eval ?no)))
_if
```

```
? (_if 't (+ 3 4) (* 3 4))
7

? (_if '() (+ 3 4) (* 3 4))
12
```

The handling of lazy parameters is also available when processing variable numbers of parameters:

```
 ? (define (echo . ?L) (println ?L))
echo

? (echo 1 a (+ 3 4))"
(1 a (+ 3 4))
```

## set! and set!!

The *set!* function changes the value bound to a symbol. As opposed to *define*, *set!* can change bindings in other binding contexts. Its general form is:

*(set! name expr context)*

The evaluation of the expression *expr* is assigned to the symbol *name* that is not evaluated except in the following cases:

- If *name* is a list then it is first evaluated to determine the actual name.
- When *set!* is invoked from within a function and *name* is the name of a formal parameter in the calls stack then it is substituted with the actual parameter (which must resolve as a symbol).

When the *expr* value is missing or *nil* then the symbol is actually removed from its binding context.

The optional *context* parameter is the identifier of the binding context to look-up the symbol from:

- When not given or invalid, then all binding contexts are searched for, starting from the current one. When running through the binding contexts, *set!* stops at the first one with

a matching symbol. If no symbol is found then a new one is created in the current binding context.

- When the binding context is root, core, data, user or an integer greater than 3 (the minimum binding context the user can change and identified by the logical name user) but lower than the current binding context then the search is limited to that context. The example file *gensym* takes advantage of the context to "push" into the data binding context its parameters such as making it useful and workable across any binding contexts, whatever binding context the function is declared from.

The *set!!* function is a helper function to invoke the *set!* function if the name of the symbol to bind to is actually the value of the symbol *name* when it is not a function call argument or list. This function comes handy to avoid expressions like *(eval (list quote name))* and mainly used when the *gensym* function or any similar mechanism is used to obtain a symbol name. See the *closure* function described in the "lexical versus dynamic bindings" paragraph or the *make-accumulator* function of the SICP example file.

Examples:

```
? (define (f a b) (set! a b))
f

? (define x 1)
x

? (f 'x 42)
x

? x
42

? (set! y)
y

? (f 'y 6)
y

? y
```

```
nil
```

In this example, *y* remains unbound after the call to *(f 'y 6)* because the *set!* function assigned a value to the *y* symbol in the binding context of the function *f*, and thus destroyed when the function returned. This did not occur for *x* because *set!* found the symbol in a previous calling binding context and set it. In order to keep the binding for *y* regardless of whether it is defined or not, *f* must force the assignment into a specific context:

```
? (define (f a b) (set! a b 'data))
f

? (set! y)
y

? (f 'y 6)
y

? y
6
```

See the *sys::bindings*, *sys::bindings-names* and *sys::bindings-assoc* functions for listing the binding contexts.

See the Startup paragraph for details regarding the initial creation and loading of the binding contexts.

## car, cdr, and all variations

The car and cdr functions are among the most used functions when writing Lisp programs. In order to simplify the combinations of nested car and cdr, they may be grouped together in a single word cxr where x is any number of combination of the letters a and d as in: car, caar, cadr, caaaadadr, etc. The "letters" are applied from right to left, thus (cadr ...) is equivalent to (car (cdr ...)). All these functions are considered built-in functions.

## print, println, and error

When executed from within the console, all the evaluations of the expressions are automatically printed using the println function; however when loading a file or executing the code of the editor, only explicitly printed messages with the functions print, println, or error are printed. In order to override this behavior, you need to call the print function with the special string ";**SYS.OPT: PRINT=YES". To turn off and get back to the default behavior just invoke print with the string ";**SYS.OPT: PRINT=NO". These 2 special evaluations are not printed.

## Built-in functions

Built-in functions have precedence over any other expressions when evaluating, thus their behavior cannot be changed nor overridden by a *define* or *set!* expression; for instance the instruction *(set! cons <expression>)* associates an expression to the symbol *cons* within a binding context but the built-in *cons* remains the function that is invoked by the interpreter upon evaluation of *(cons a b)*.

Following this example, things can get tricky when *eval* or *apply* are used to evaluate *cons* because *cons* as a symbol now evaluates to the expression assigned to by the *set!* instruction:

```
? cons
cons                                    ; cons symbol is a built-in

? (set! cons (lambda (a b) (print "hello " a " + " b)))
cons

? cons                                  ; cons symbol association
(lambda (a b) (print "hello " a " + " b))

? (cons 'alpha '(beta))                 ; built-in cons is used to evaluate
(alpha beta)

? (apply 'cons '(alpha beta))           ; built-in cons is used to evaluate
(alpha . beta)

? (apply cons '(alpha beta))            ; associated value of cons is used
hello alpha + beta
```

## Tail-recursive functions

My Lisp is properly tail-recursive, thus iterations may be implemented using tail-recursive calls without risking stack overflow issues. The REPL function given in the overview is an example of a function taking advantage of tail-recursion to avoid using the progn for looping back to the read-eval-print sequence:

```
(define (REPL eval_me)
        (REPL (println (eval (read)))))
```

# Dynamic binding

## What is dynamic binding?

My Lisp uses dynamic binding because it is very powerful but yet simple to put in place and understand. The interpreter environment is unique and may be viewed as a stack where the bindings are pushed and popped. From an implementation point of view, this means that the interpreter does not need hidden structures to manage things like continuations: the free symbols of lambda expressions have values depending on when the lambda expressions are evaluated as opposed to when created.

The following example illustrates the handling of free variables with the *addx* function that is not adding 4 to its argument in the general case because the variable x is free inside the returned lambda:

```
(define addx
        (let ((x 4))
                (lambda (p) (+ x p))))

? addx
(lambda (p) (+ x p))

? (define x 2)
x

? (addx 3)
5

? (define x 4)
x

? (addx 3)
7
```

Here is another example from Richard Stallman Emacs presentation text:

```
(define foo1 (x) (foo2))
(define foo2 ()  (+ x 5))
```

```
? (foo1 2)
7
```

The function *foo2* is executing in a binding environment where the variable x has been bound to the value of the parameter used when invoked the function *foo1*.

## Current bindings

The interpreter binding contexts can be viewed as a stack where the bindings between the symbols and the values are stacked upon function calls or when the functions *define* and *set!* are invoked:

- The *define* function adds or changes the value associated to a symbol at the current level of the binding contexts.
- The *set!* function is able to change the value associated to a symbol at any level of the binding contexts.

It is possible to access the current bindings using the library functions *sys::bindings*, *sys::binding-names*, and *sys::bindings-assoc*.

## Lexical versus dynamic binding

Whenever necessary, it is somehow easy to add lexical binding (aka closures) to any expression using the *gensym* and *closure* functions directly borrowed from Le Lisp and defined in the sample file Le_Lisp.lisp:

```
(define add4
        (let ((x 4))
                (closure '(x)
                        (lambda (p) (+ x p)))))

? (add4 3)
7
```

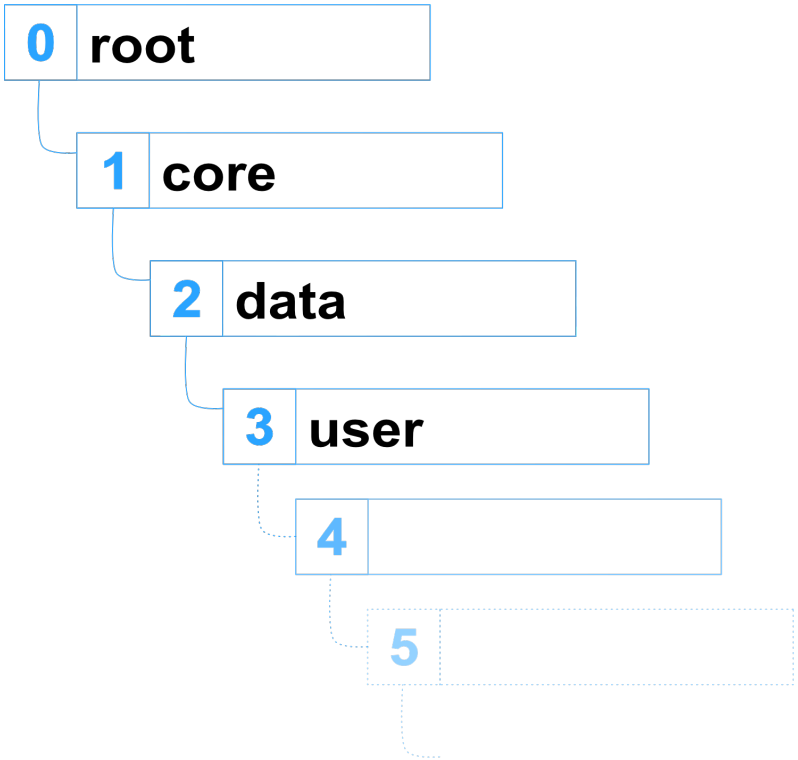
```
? (define x 42)
x

? (add4 3)
7
```

See the description of the *closure* function in Le_Lisp section for further details and examples. In particular, the *clet* function example for a lexically scoped version of *let* that is compatible with Scheme.

# Startup

Initially the interpreter is loaded with all the built-in functions and symbols, and then further enhanced with the library files *Math*, *Tools*, *Gensys*, *Closure*, *Dictionary*, *Stack*, *Rationals,* and *Turtle*. Then the console interpreter is loaded with all the user symbols defined during previous sessions in the data and user binding contexts.

The binding contexts can be represented by the following diagram:




The root binding context is the top-level context and contains common symbols like *nil* and *t*. The core binding context is where all library files are initially loaded. Both the root and core binding contexts are considered system contexts and you should not try to amend them ; the system is not preventing writing them but any change won't be saved and restored across sessions. The data binding context is a special context intended to save the values of special variables like counters; for instance *gensym* uses it to save the current state of its parameters such as they can be restored properly across sessions. The data binding context is intended for the *set!* function when values should be persisted across sessions. The user binding context is the "initial" binding context when entering the interpreter; it is also the binding context where the *load* function is executed, thus all defined symbols during the loading of a file are ending up there.

At any time during the execution of the interpreter, the current binding context is at least the user one; when looking for the value associated to with a symbol, the interpreter starts from the current binding context up to the root one. As far the *set!* function is concerned, it is possible to address any specific environment to change the value

associated to a symbol; don't forget however that only the data and user binding contexts are saved and restored across sessions.

See the *sys::bindings*, *sys::bindings-names* and *sys::bindings-assoc* functions for enumerating the binding contexts. See *set!* for changing the bindings.

Note that in older versions of My Lisp, the user context was named base. You can still use this name for compatibility reasons, but are not encouraged to.

# Common questions

## Can I create macros or syntactic definitions ?

My Lisp does not support macros as intended by Scheme or other Lisp dialects. It does however provides the universal function *define* that offers many extensions over the standard *define* ones (*define*, *defun*, *de*, ...) with non-evaluated parameters ?x, variable number of arguments, and default values.

## Where are begin, defun, etc?

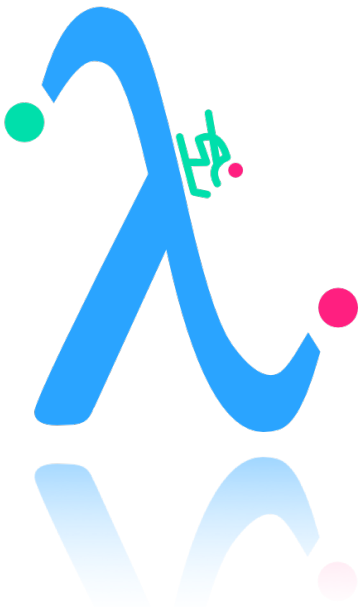
My Lisp implements the core functions using a naming convention that may not match the dialect of Lisp you are used to; for instance Scheme *begin* is *progn*. My Lisp also tries to expose a minimal set of functions, leaving out some; for most of the cases this is not a problem as you can easily implement and add them into a library file as My Lisp itself does with the file Tools.lisp and the classical *apply* and *mapcar* functions. You can also use the *define* function to create synonyms as in *(define defun define)*: from now on you can use *defun* to declare functions; this does not change the semantic behavior of *define* but let you write expressions with your favorite keywords.

## Mathematics ?

For some historical reasons My Lisp ships with some numerical analysis methods and complete complex numbers support. Thus you will be able to solve common numerical problems using Lisp recipes. You can also experience formal mathematics with the symbolic derivation module that computes the formal derivation of virtually any symbolic function expressed as a Lisp expression (or in “standard” mode using the conversion functions available in the infix sample file); feel free to expand the source code with your own rules and recipes. Numbers (integers and floating points) are based on IEEE- 754 double-precision numbers (64-bit base-2 format).

My Lisp also support so-called big integers and rationals which precisions are limited only by the available memory. The default behavior is to parse integer and rational strings are big integers, which in turn are automatically converted to floating point values when required by the evaluated functions. This implicit handling can be turned off or modified using the options::integer-mode function.

# McCarthy



# quote

## SYNOPSIS

'<expr>

(quote <expr>)

## DESCRIPTION

QUOTE returns its <expr> argument as-is and not evaluated.

## RETURN VALUE

SEXPR

## EXAMPLES

? 'a

a

? (quote a)

a

? '(a b c)

(a b c)

# atom

## SYNOPSIS

(atom <expr>)

## DESCRIPTION

ATOM returns T if <expr> evaluates to an atom or the empty list, NIL otherwise.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (atom 'a)
t

? (atom '(a b c))
nil

? (atom (atom 'a))
t

? (atom '(atom 'a))
nil
```

# eq

## SYNOPSIS

(eq <expr1> <expr2>)

## DESCRIPTION

EQ returns T if the values of <expr1> and <expr2> are the same atom or both the empty list, NIL otherwise.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (eq 'a 'a)
t

? (eq 'a 'b)
nil

? (eq '() '())
t
```

# car

## SYNOPSIS

**(car <expr>)**

## DESCRIPTION

CAR returns the first element of the value of <expr>. If the value of <expr> is not a list then NIL is returned. Note that CAR and CDR may be combined together as in CADR, CAAADR, CDAR, etc.

## RETURN VALUE

EXPR or NIL

## EXAMPLES

```
? (car '(a b c))
a

? (cadr '(a b c))
b
```

# cdr

## SYNOPSIS

**(cdr <expr>)**

## DESCRIPTION

CDR returns the tail of the value of <expr>, that is all elements but the first one. If the value of <expr> is not a list then NIL is returned. Note that CAR and CDR may be combined together as in CADR, CAAADR, CDAR, etc.

## RETURN VALUE

EXPR or NIL

## EXAMPLES

```
? (cdr '(a b c))
(b c)

? (caddr '(a b c d))
c

? (cddadr '(a (x y z) b c d))
(z)
```

# cons

## SYNOPSIS

(cons <expr1> <expr2>)

## DESCRIPTION

CONS returns a list containing the value of <expr1> followed by the elements of the value of <expr2>. If the value of <expr2> is not a list then NIL is returned.

## RETURN VALUE

EXPR or NIL

## EXAMPLES

```
? (cons 'a '(b c))
(a b c)

? (cons 'a (cons 'b (cons 'c '())))
(a b c)

? (car (cons 'a '(b c)))
a

? (cdr (cons 'a '(b c)))
(b c)
```

# cond

## SYNOPSIS

**(cond (<test> <expr> <expr>...) (<test> <expr> <expr>...) ...)**

## DESCRIPTION

COND evaluates the <test> expressions from left to right until one returns T. When one is found, COND evaluates all the corresponding <expr> expressions from left to right and returns the value of the last one. If no test is found or no expression is associated with the test, then NIL is returned.

## RETURN VALUE

EXPR or NIL

## EXAMPLES

```
? (cond ((eq 'a 'b) 'first) ((atom 'a) 'second))
second

? (cond ((eq 'a 'b) 'first) ('t 'else))
else
```

# lambda

## SYNOPSIS

**(lambda (<p> <p>...) <expr>)**

**(lambda <p> <expr>)**

## DESCRIPTION

LAMBDA defines a function with the formal parameters <p> and the <expr> as body. Upon evaluation of the lambda, a new environment is created with the invocation parameters bound to the formal parameters.

If a space-delimited period character precedes the last formal parameter, then upon invocation the last variable is bound to a list with all the tail parameters; the second declaration form is actually a shortcut for (lambda ( . <p>) <expr>).

## RETURN VALUE

LAMBDA EXPRESSION

## EXAMPLES

```
? ((lambda (f) (f '(b c))) '(lambda (x) (cons 'a x)))
(a b c)

? ((lambda x x) 3 4 5 6)
(3 4 5 6)

? ((lambda ( . x) x) 3 4 5 6)
(3 4 5 6)

? ((lambda (x y . z) z) 3 4 5 6)
(5 6)
```

# label

## SYNOPSIS

**(label name (lambda...))**

## DESCRIPTION

LABEL defines a named lambda expression and was originally used to simplify recursive calls. See the DEFINE function that is more suitable for naming expressions.

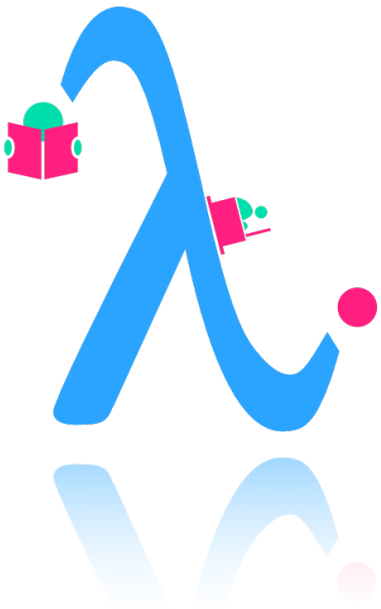
## RETURN VALUE

LAMBDA EXPRESSION

## EXAMPLES

```
? ((label recar (lambda (x) (cond ((atom x) x) ('t (recar (car x)))))) '((a b c) c d e))
a
```

# Core



# define

## SYNOPSIS

**(define <name>(<p> <p>...) <expr>...)**

**(define (<name> <p> <p>...) <expr>...)**

**(define <name> <expr>)**

## DESCRIPTION

DEFINE associates the named symbol <name> to the lambda expression (LAMBDA (<p> <p>...) <expr>). In its third form, the <expr> expression must be a single expression that is first evaluated and then associated to <name>. As opposed to LABEL, DEFINE modifies the binding of <name> within the current environment whilst LABEL creates a new binding.

If a space-delimited period character precedes the last formal parameter, then upon invocation the last variable is bound to a list with all the tail parameters. There is no requirement for a minimum number of parameters, thus (define (f . z) <sexpr>) introduces a function where z will be associated to (1 2 3 4) when evaluating (f 1 2 3 4).

Upon invocation of the LAMBDA expression arguments are always passed by value, which means that the actual argument expressions are evaluated before the procedure gains control, whether the lambda expression needs the result of the evaluation or not.

However, if the name of the parameter starts with a ? character, the evaluation rules does not apply to this parameter and it will be up to the lambda to evaluate the parameter if needed. This mechanism allows redefining all functions, including the built-in ones like if, cond, etc.; see the example of the _if function below or WHEN and UNLESS functions in Tools.lisp for further details.

A parameter <p> may be given in the form of a list (<pn> <pv>) where <pn> is the actual name of the parameter and <pv> the default for the parameter when missing during the function call; the <pv> expression is evaluated when the function is defined. If a parameter is not explicitly associated to with a default value then NIL is assumed.

## RETURN VALUE

EXPR

## EXAMPLES

```
? (define fact(n) (if (<= n 1) n (* n (fact (- n 1)))))
fact

? (fact 4)
24

? (define x (car '(a b c)))
x

? x
a

? (define zz car)
zz

? (zz '(a b c))
a

? (define (zz a b . c) (cdr c))
zz

? (zz 'a 'b 'c 'd 'e)
(d e)

? (define (_if test ?yes ?no) (if test ?yes ?no))
_if

? (_if 't (+ 3 4) (* 3 4))
(+ 3 4)

? (_if '() (+ 3 4) (* 3 4))
(* 3 4)

? (define (_if test ?yes ?no) (if test (eval ?yes) (eval ?no))
```

_if

```
? (_if 't (+ 3 4) (* 3 4))
7

? (_if '() (+ 3 4) (* 3 4))
12

? (define (addxy (x 2) (y 40)) (+ x y))
addxy

? (addxy 1 2)
3

? (addxy 1)
41

? (addxy)
42
```

# eval

## SYNOPSIS

(eval <expr>)

## DESCRIPTION

EVAL returns the evaluation of the <expr> expression.

## RETURN VALUE

EXPR

## EXAMPLES

```
? (eval '(+ 4 5))
9

? (eval '(if (< 1 2) 'first 'second))
first
```

# if

## SYNOPSIS

**(if <test> <then-expr> <else-expr> <else-expr>...)**

## DESCRIPTION

IF evaluates the <test> expression ; if it is T then the <then-expr> is evaluated and returned; otherwise all the <else-expr> expressions are evaluated from left to right and the value of the last one is returned. If no <else-expr> is found then NIL is returned.

## RETURN VALUE

EXPR or NIL

## EXAMPLES

**? (if (eq 'a 'b) 'first 'second)**

**second**

# list

## SYNOPSIS

**(list <expr> <expr> <expr>...)**

## DESCRIPTION

LIST returns the list made of all the <expr> expressions evaluated from left to right. LIST without <expr> returns NIL.

## RETURN VALUE

EXPR or NIL

## EXAMPLES

**? (list (cons 'a '(b c)) 'second 'third 4 5 6)**

**((a b c) second third 4 5 6)**

# load

## SYNOPSIS

**(load filename ...)**

## DESCRIPTION

LOAD reads and evaluates the expressions contained in the My Lisp source code file "filename". If more than one file is given then the files are loaded from left to right, sequentially.

A filename is a string or atom pointing to a file of the underlying files system. If a user file and a system file have the same name, the user file is loaded; in order to force a system file, the filename must be prefixed by the # character. User files are searched in My Lisp folder of the local device, and then in the My Lisp folder of iCloud. The / character is used to denote the path separator. There is no need to append to lisp extension to the filename as it is automatically appended.

Note that files are loaded and interpreted within the root context of the interpreter, thus symbols and definitions remain available even if loaded from nested functions; a side-effect is that the interpretation of the files depends only of the symbols and functions defined in the root context.

Upon loading, all outputs are disabled except for the PRINT and PRINTLN ones.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (load "#HelpFile")
T

? (load "my samples/test1")
T
```

# progn

## SYNOPSIS

**(progn <expr> <expr>...)**

## DESCRIPTION

PROGN evaluates all the <expr> expressions from left to right and returns the value of the last one. PROGN without <expr> returns NIL Note that when using DEFINE in the form (define (<name> <args>) <body>) with a <BODY> of at least 2 expressions, a PROGN is explicitly created and used to evaluate the BODY expressions.

This function is also known as BEGIN in Scheme.

## RETURN VALUE

EXPR or NIL

## EXAMPLES

**? (progn (car '(a b c)) (cdr '(a b c)))**

**(b c)**

# set!

## SYNOPSIS

**(set! <name> <expr>)**

## DESCRIPTION

SET! associates the named symbol <name> to the evaluation of <expr>. If <name> is a list then it is first evaluated to determine the actual symbol name. If <expr> is omitted then the expression associated to with the named symbol <name> is removed from the bindings. When there is no symbol with the given name then SET! is equivalent to DEFINE.

Typically this function is used to change the value associated to with an existing symbol outside the immediate binding context and thus has an important side-effect.

Note that built-in functions have precedence over other functions when evaluating, thus SET! does not change the evaluation result of a built-in symbol.

## RETURN VALUE

EXPR

## EXAMPLES

```
? (define x 12)
x

? (define (addx n) (set! x (+ x n)))
addx

? (addx 3)
x

? x
15
```

# while

## SYNOPSIS

**(while <test> <expr> <expr>...)**

## DESCRIPTION

WHILE evaluates the <test> expression ; if it is T then the <expr> are evaluated from left to right, and then it repeats the same evaluations from the <test> expression. Otherwise, the value of the last evaluation is returned.

Obviously WHILE expects some change(s) to the evaluation environment using DEFINE to avoid an infinite loop.

## RETURN VALUE

EXPR

## EXAMPLES

```
? (define a 10)
a

? (while (> a 0) (define a (- a 1)) (print a))
9876543210
```

# error

## SYNOPSIS

(error <expr> <expr>...)

## DESCRIPTION

ERROR evaluates all the <expr> expressions from left to right and prints them onto the current error console. A newline is appended after the last printed expression. Then the current input processing is stopped and the interpreter back to the top level.

## RETURN VALUE

NONE

## EXAMPLES

? (error "Not useful within REPL!")

Not useful within REPL!

# println

## SYNOPSIS

(println <expr> <expr>...)

## DESCRIPTION

PRINTLN evaluates all the <expr> expressions from left to right and prints them onto the current output console. A newline is appended after the last printed expression.

This function is also known as DISPLAY/NEWLINE in Scheme.

## RETURN VALUE

NONE

## EXAMPLES

```
? (println "hello world")
Hello world
```

# print

## SYNOPSIS

**(print <expr> <expr>...)**

## DESCRIPTION

PRINT evaluates all the <expr> expressions from left to right and prints them onto the current output console.

This function is also known as DISPLAY in Scheme.

## RETURN VALUE

NONE

## EXAMPLES

**? (println "hello world")**

**Hello world**

# read

## SYNOPSIS

(read)

## DESCRIPTION

READ reads the next SEXPR from the standard input, typically the current interpreter console. If the user enters more than one SEXPR, then READ returns the first one and the next call will return the next ones.

## RETURN VALUE

SEXPR

## EXAMPLES

```
? (read)
(* 2 3)    ; assuming the user entered (* 2 3)

? (eval (read))
6         ; assuming the user entered (* 2 3)
```

# read-string

## SYNOPSIS

(read-string)

## DESCRIPTION

READ-STRING reads a string from the standard input, typically the current interpreter console. As opposed to READ, the input is not parsed into a SEXPR. The string is made of all characters up to (but not including) the ENTER key the user pressed to validate the input.

## RETURN VALUE

STRING

## EXAMPLES

```
? (string? (read-string))
t
```

# eqv?

## SYNOPSIS

(eqv? <expr1> <expr2>)

## DESCRIPTION

EQV? returns T if the values of <expr1> and <expr2> are the same or equivalent, NIL otherwise. Typically if 2 expressions have the same type and the same string representations then they are equivalent.

Note that EQV? does not perform lambda-calculus alpha and beta reductions, thus considers (lambda (x) x) and (lambda (y) y) two different expressions.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (eqv? 'a 'a)
t

? (eqv? 1 2)
nil

? (eqv? '(1 2 3) "(1 2 3)")
nil

? (eqv? '(1 2 3) '(1 2 3))
t

? (eq '(1 2 3) '(1 2 3))
nil
```

# integer?

## SYNOPSIS

(integer? <expr>)

## DESCRIPTION

INTEGER? returns T if the value of <expr> is an integer, NIL otherwise. Note that a floating point number is not an integer in the sense of INTEGER?.

An integer is also a number, and a rational with 1 as denominator is an integer.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (integer?  42)
t

? (integer? 42.0)
nil

? (integer? (/ 4 2))
t
```

# list?

## SYNOPSIS

(list? <expr>)

## DESCRIPTION

LIST? returns T if the value of <expr> is a list, NIL otherwise.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (list? '(a b c))
t

? (list? 'a)
nil

? (list? '())
t
```

# null?

## SYNOPSIS

**(null? <expr>)**

## DESCRIPTION

NULL? returns T if the value of <expr> is the empty list, NIL otherwise. It is defined by (define (null? x) (eq x '())).

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (null? '(a b c))
nil

? (null? 'a)
nil

? (null? '())
t

? (null? nil)
t

? (null? (cdr '(a)))
t
```

# number?

## SYNOPSIS

(number? <expr>)

## DESCRIPTION

NUMBER? returns T if the value of <expr> is a number, NIL otherwise. The NaN expression returned by most mathematical function returns T against NUMBER? even though it indicates an invalid one.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (number? '(a b c))
nil

? (number? 3)
t

? (number? (+ 3 4))
t
```

# procedure?

## SYNOPSIS

**(procedure? <expr>)**

## DESCRIPTION

PROCEDURE? returns T if the value of <expr> is a procedure, NIL otherwise. A procedure is either a built-in function or a user function defined as a list starting with the LABEL or LAMBDA symbol.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (procedure? car)
t

? (procedure? 'car)
nil

? (procedure? (lambda (x) (* x x)))
t

? (procedure? '(lambda (x) (* x x)))
nil

? (define f (lambda (x) (* x x)))
f

? (procedure? f)
t
```

# rational?

## SYNOPSIS

(rational? <expr>)

## DESCRIPTION

RATIONAL? returns T if the value of <expr> is a rational, NIL otherwise. Note that an integer is a rational but a floating point number is not a rational in the sense of RATIONAL?.

A rational is also a number, and a floating point value is a number for NUMBER? but not a rational for RATIONAL?.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (rational?  42/67)
t

? (rational? 42.0)
nil

? (rational? 42)
t

? (define (float? u) (and (number? u) (not (rational? u))))
float?
```

# string?

## SYNOPSIS

(string? <expr>)

## DESCRIPTION

STRING? returns T if the value of <expr> is a string, NIL otherwise.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (string? '(a b c))
nil

? (string? 3)
nil

? (string? "hello world")
t
```

# string->code

## SYNOPSIS

(string->code <string>)

(string->code <string> <index>)

## DESCRIPTION

STRING->CODE returns the UNICODE integer value of the character at the 0-based <index> position in the string parameter <string>; in its first form, the position is 0. If the parameter is not a string or the index an invalid position then NAN is returned.

## RETURN VALUE

INTEGER NUMBER or NAN

## EXAMPLES

```
? (string->code "ABC")
65

? (string->code "ABC" 1)
66

? (string->code "ABC" 3)
nan

? (string->code "12😃4" 2)
128515
```

# string<-code

## SYNOPSIS

**(string<-code <value>)**

## DESCRIPTION

STRING<-CODE returns a one character string from its UNICODE value. If the code value is not an integer then NIL is returned.

## RETURN VALUE

STRING or NIL

## EXAMPLES

**? (string<-code 65)**

**"A"**

**? (string<-code 128515)**

"😃"

# string-index

## SYNOPSIS

(string-index <source> <pattern>)

## DESCRIPTION

STRING-INDEX returns the 0-based index of the string <pattern> within the string <source>. If the <pattern> string is not found or the arguments are invalids, then NAN is returned.

## RETURN VALUE

INTEGER NUMBER or NAN

## EXAMPLES

? (string-index "hello world" "world")

6

? (string-index "hello world" "he")

0

? (string-index "hello world" "monde")

nan

? string-index "12😃456" "4")

4

# string-length

## SYNOPSIS

(string-length <string>)

## DESCRIPTION

STRING-LENGTH returns the number of characters in the string argument. If <string> is not a string, then 0 is returned.

## RETURN VALUE

NUMBER

## EXAMPLES

? (string-length "hello world")

11

? (string-length "12😃456")

6

# string->list

## SYNOPSIS

(string->list <string>)

(string->list <string> <sep>)

## DESCRIPTION

STRING->LIST splits its <string> argument into a list of strings. When the field separator <sep> is omitted, the whitespace is assumed.

## RETURN VALUE

LIST

## EXAMPLES

? (string->list "hello world, bonjour le monde")

("hello" "world," "bonjour" "le" "monde")

? (string->list "hello world, bonjour le monde" ",")

("hello world" " bonjour le monde")

# string->lower

## SYNOPSIS

(string->lower <string>)

## DESCRIPTION

STRING->LOWER returns the string argument with all its characters converted to lower-case.

## RETURN VALUE

STRING

## EXAMPLES

```
? (string->lower "ABc")
" abc"
```

# string->number

## SYNOPSIS

(string->number <string>)

## DESCRIPTION

STRING->NUMBER parses its <string> argument into a number. It uses the exact same rules as My Lisp parser, thus the number format is culture independent. The special number NAN is returned is the argument is not a string or cannot be parsed.

Note that "12.3" is parsed as a floating point number; if the exact rational number is required, you must use the ->RATIONAL function.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (+ (string->number "1.2345") 2)
3.2345
```

# string->string

## SYNOPSIS

(string->string <string> <startIndex>)

(string->string <string> <startIndex> <length>)

## DESCRIPTION

STRING->STRING extracts a substring from a string starting at the given start index with the first character at offset 0. When <length> is missing, the tail of the string is returned; otherwise as many characters as indicated are returned.

## RETURN VALUE

STRING

## EXAMPLES

? (string->string "bonjour le monde" 8)

"le monde"

? (string->string "bonjour le monde" 8 2)

"le"

? (string->string "12😃456" 1 3)

"2😃4"

# string->symbol

## SYNOPSIS

(string->symbol <string>)

## DESCRIPTION

STRING->SYMBOL converts the string argument into a symbol.

## RETURN VALUE

ATOM

## EXAMPLES

```
? (define x '(1 2 3))
x

? (string->symbol "x")
x

? (eval (string->symbol "x"))
(1 2 3)

? (string->symbol (string<-code 128515))
😃
```

# string->upper

## SYNOPSIS

(string->upper <string>)

## DESCRIPTION

STRING->UPPER returns the string argument with all its characters converted to upper-case.

## RETURN VALUE

STRING

## EXAMPLES

```
? (string->upper "abC")
"ABC"
```

# ->string

## SYNOPSIS

(->string <expr> <expr>...)

## DESCRIPTION

->STRING concatenates the string representation of its arguments and returns the resulting string.

## RETURN VALUE

STRING

## EXAMPLES

```
? (->string 1 2 'hello (+ 3 4))
"12hello7"
```

# ->float

## SYNOPSIS

(->float <expr>)

## DESCRIPTION

->FLOAT forces the conversion of an integer or rational into its closest floating point number representation.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (/ 1 3)
1/3

? (/ 1 (->float 3))
0.3333
```

# ->integer

## SYNOPSIS

(->integer <expr>)

## DESCRIPTION

->INTEGER forces the conversion of a number into an integer. The result is the same as the function TRUNCATE except that the type of the result is always an integer.

## RETURN VALUE

INTEGER NUMBER

## EXAMPLES

```
? (->integer 12.3)
12

? (integer? (truncate 12.3))
nil

? (integer? (->integer 12.3))
t
```

# ->rational

## SYNOPSIS

(->rational <expr>)

## DESCRIPTION

->RATIONAL converts an expression into a rational. When <expr> is a string, its usual decimal form representation is assumed; that is “12.3” is a valid representation whilst 123/10 is not valid.

This function is especially useful when the exact representation of a decimal number is required without losing precision du to floating point approximations (see example below).

## RETURN VALUE

RATIONAL NUMBER or NAN

## EXAMPLES

```
? (->rational “12.3”)
123/10

? (->rational 12.3)
3462142213541069/281474976710656

? (rational->decimal-string (- (->rational 12.3) (->rational "12.3")))
0.00000000000000071054273576010018587 1124267578125
```

# ->denominator

## SYNOPSIS

(->denominator <expr>)

## DESCRIPTION

->DENOMINATOR returns the denominator of the rational number <expr>.

## RETURN VALUE

INTEGER NUMBER

## EXAMPLES

```
? (->denominator 42/67)
67

? (->denominator  2/6)
3

? (->denominator  42)
1
```

# ->numerator

## SYNOPSIS

(->numerator <expr>)

## DESCRIPTION

->NUMERATOR returns the numerator of the rational number <expr>.

## RETURN VALUE

INTEGER NUMBER

## EXAMPLES

```
? (->numerator 42/67)
42

? (->numerator  10/15)
2

? (->numerator  42)
42
```

# nan?

## SYNOPSIS

```
(nan? <expr>)
```

## DESCRIPTION

NAN? returns T if the value of expression <expr> is the 'not a number' or an undefined one (NAN or infinity), NIL otherwise.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (NAN? (/ 1 0))
t

? (NAN? 1)
nil
```

# =

## SYNOPSIS

(= <expr1> <expr2>...)

## DESCRIPTION

= returns T if the values of expressions <expr1>, <expr2>, ..., are all equal; NIL otherwise.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (= 1 1)
t

? (= 1 2)
nil

? (= 1 (- 2 1) (/ 4 4))
t
```

# <

## SYNOPSIS

(< <expr1> <expr2> ...)

## DESCRIPTION

< returns T if the values of the expressions <expr1>, <expr2>, ..., are monotonically increasing; NIL otherwise.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (< 1 3 5 7)
t

? (< 1 3 2 5 7)
nil

? (< 1 3 3 5 7)
nil

? (< 3 1)
nil

? (define x 5)
x

? (< x 7)
t
```

# <=

## SYNOPSIS

(<= <expr1> <expr2> ...)

## DESCRIPTION

<= returns T if the values of the expressions <expr1>, <expr2>, ..., are monotonically nondecreasing; NIL otherwise.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (<= 1 3 5 7)
t

? (<= 1 3 2 5 7)
nil

? (<= 1 3 3 5 7)
t

? (<= 3 1)
nil

? (define x 5)
x

? (<= x 7)
t
```

# >=

## SYNOPSIS

(>= <expr1> <expr2> ...)

## DESCRIPTION

>= returns T if the values of the expressions <expr1>, <expr2>, ..., are monotonically nonincreasing; NIL otherwise.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (>= 7 5 3 1)
t

? (>= 7 5 2 3 1)
nil

? (>= 7 5 3 3 1)
t

? (>= 1 3)
nil

? (define x 5)
x

? (>= 7 x)
t
```

# >

## SYNOPSIS

(> <expr1> <expr2> ...)

## DESCRIPTION

> returns T if the values of the expressions <expr1>, <expr2>, ..., are monotonically decreasing; NIL otherwise.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (> 7 5 3 1)
t

? (> 7 5 2 3 1)
nil

? (> 7 5 3 3 1)
nil

? (> 1 3)
nil

? (define x 5)
x

? (> 7 x)
t
```

# +

## SYNOPSIS

(+ <expr1> <expr2> ...)

## DESCRIPTION

+ returns the sum the values of the expressions <expr1>, <expr2>, etc.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (+ 1 2 3 4 5)
15
```

# *

## SYNOPSIS

(* <expr1> <expr2> ...)

## DESCRIPTION

* returns the product the values of the expressions <expr1>, <expr2>, etc.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (* 1 2 3 4 5)
120
```

# -

## SYNOPSIS

(- <expr0> <expr1> <expr2> ...)

## DESCRIPTION

- returns the value of the expression <expr0> minus the sum of the values of the expressions <expr1>, <expr2>, etc.

When no <expr1> is given, - returns the additive inverse of the value of <expr0>.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (- 10 2 3 4)
1

? (- 10)
-10
```

# /

## SYNOPSIS

(/ <expr0> <expr1> <expr2> ...)

## DESCRIPTION

/ returns the value of the expression <expr0> divided by the product of the values of the expressions <expr1>, <expr2>, etc.

When no <expr1> is given, / returns the multiplicative inverse of the value of <expr0>.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (/ 24 3 4)
2

? (/ 3)
0.333333

? (= 1 (* 3 (/ 3)))
t
```

# abs

## SYNOPSIS

(abs <expr>)

## DESCRIPTION

ABS returns the absolute value of the value of <expr>.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (abs (- 1 3))
2
```

# ceiling

## SYNOPSIS

(ceiling <expr>)

## DESCRIPTION

CEILING returns the smallest integer that is larger than the value of <expr>. The result has the same sign as <expr>. The result is an integer when both parameters are rationals, a float otherwise.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (ceiling 3.5)
4

? (ceiling -4.3)
-4
```

# floor

## SYNOPSIS

(floor <expr>)

## DESCRIPTION

FLOOR returns the largest integer smaller than the value of <expr>. The result has the same sign as <expr>. The result is an integer when both parameters are rationals, a float otherwise.

## RETURN VALUE

NUMBER

## EXAMPLES

? (floor 3.5)

3

? (floor -4.3)

-5

# round

## SYNOPSIS

(round <expr>)

## DESCRIPTION

ROUND returns the closest integer to the value of <expr>, rounding to even when the value of <expr> is halfway between two integers. The result is an integer when both parameters are rationals, a float otherwise.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (round 3.5)
4

? (round -4.3)
-4

? (round -4.7)
-5
```

# truncate

## SYNOPSIS

(truncate <expr>)

## DESCRIPTION

TRUNCATE returns the integer with the same sign as <expr> and such as its absolute value is the largest integer smaller than the absolute value of <expr>. This is the same function as floor on positive integers and ceiling on negative numbers. The result is an integer when both parameters are rationals, a float otherwise.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (truncate 3.5)
3

? (truncate -4.3)
-4
```

# modulo

## SYNOPSIS

(modulo <expr1> <expr2>)

## DESCRIPTION

MODULO returns the remainder of the integer division of the value of <expr1> by the value of <expr2>. The modulo is always positive and its value lower than the absolute value of <expr2>.

## RETURN VALUE

INTEGER NUMBER

## EXAMPLES

```
? (modulo 13 4)
1

? (modulo 13 -4)
1

? (modulo -13 4)
3

? (modulo -13 -4)
3
```

# quotient

## SYNOPSIS

(quotient <expr1> <expr2>)

## DESCRIPTION

QUOTIENT returns the quotient of the integer division of the value of <expr1> by the value of <expr2>. The quotient is negative if and only if exactly one of <expr1> and <expr2> is negative. The quotient q and remainder r of a and b are defined by:

$$a = bq + r, 0 \leq r < |b|$$

## RETURN VALUE

INTEGER NUMBER

## EXAMPLES

```
? (quotient 13 4)
3

? (quotient 13 -4)
-3

? (quotient -13 4)
-3

? (quotient -13 -4)
3
```

# remainder

## SYNOPSIS

(remainder <expr1> <expr2>)

## DESCRIPTION

REMAINDER returns the remainder of the integer division of the value of <expr1> by the value of <expr2>. The remainder has the same sign as the value of <sexpr1> and its absolute value is lower than the absolute value of <sexpr2>. See the quotient function for further details. The MODULO is the REMAINDER "adjusted" by a multiple of <expr2> such as being positive.

## RETURN VALUE

INTEGER NUMBER

## EXAMPLES

```
? (remainder 13 4)
1

? (remainder 13 -4)
1

? (remainder -13 4)
-1

? (remainder -13 -4)
-1
```

# sqrt

## SYNOPSIS

(sqrt <expr>)

## DESCRIPTION

SQRT returns the square root of the value of <expr>.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (sqrt 4)
2

? (sqrt 2)
1.4142135623731
```

# expt

## SYNOPSIS

**(expt <expr x> <expr n>)**

## DESCRIPTION

EXPT returns the value of <expr x> raised to the power of the value of <expr n>.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (expt 3 4)
81
```

# exp

## SYNOPSIS

```
(exp <expr>)
```

## DESCRIPTION

EXP returns the exponential of the value of <expr>.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (exp 1)
2.71828182845905

? (exp (log 2))
2
```

# log

## SYNOPSIS

(log <expr>)

## DESCRIPTION

LOG returns the natural logarithm of the value of <expr>.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (log 1)
0

? (log (exp 3))
3

? (floor (/ (log 100) (log 10)))
2

? (floor (/ (log 999) (log 10)))
2
```

# pi

## SYNOPSIS

```
pi
```

## DESCRIPTION

PI returns the value of the universal constant π.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? pi
3.14159265358979
```

# degrees->radians

## SYNOPSIS

(degrees->radians <expr>)

## DESCRIPTION

DEGREES->RADIANS returns the value of the <expr> converted from degrees to radians.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (/ (degrees->radians 360) pi)
2
```

# radians->degrees

## SYNOPSIS

```
(radians->degrees <expr>)
```

## DESCRIPTION

RADIANS->DEGREES returns the value of the <expr> converted from radians to degrees.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (radians->degrees pi)
180
```

# cos, sin, tan

## SYNOPSIS

(sin <expr>)

(cos <expr>)

(tan <expr>)

## DESCRIPTION

SIN, COS, and TAN return the sine, cosine, and tangent of the value of <expr> expressed in radians.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (sin (/ pi 6))
0.5

? (cos (/ pi 3))
0.5

? (tan (/ pi 4))
1
```

# acos, asin, atan

## SYNOPSIS

(asin <expr>)

(acos <expr>)

(atan <expr>)

(atan <expr y> <expr x>)

## DESCRIPTION

ASIN, ACOS, and ATAN return the arcsine, arccosine, and arctangent of the value of <expr>. The returned angles are expressed in radians.

The 2 parameters version of ATAN returns the angle in radians between the X axis and the point of (x,y) adjusted such as the absolute value of the angle is lower or equal to π/2.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (radians->degrees (asin 0.5))
30

? (radians->degrees (acos 0.5))
60

? (radians->degrees (atan 1))
45

? (radians->degrees (atan 2 1))
63.4349
```

# real-mode

## SYNOPSIS

```
(real-mode)
```

## DESCRIPTION

REAL-MODE turns the math calculations to real mode, the default. All computations are made using real numbers only.

The real and complex modes are states of the interpreter, not of the application; thus real-mode is automatically restored when the interpreter is reset or the application started.

## RETURN VALUE

NONE

## EXAMPLES

```
? (real-mode)
? (sqrt -1)
nan
```

# complex-mode

## SYNOPSIS

**(complex-mode)**

## DESCRIPTION

COMPLEX-MODE turns the math calculations to complex mode. All computations are made using complex numbers; most functions are behaving as expected.

The real and complex modes are states of the interpreter, not of the application; thus real-mode is automatically restored when the interpreter is reset or the application started.

## RETURN VALUE

NONE

## EXAMPLES

```
? (complex-mode)
? (sqrt -1)
i
? (log -1)
3.141593i
```

# ->complex

## SYNOPSIS

(->complex <sexpr1> <sexpr2>)

(->complex <sexpr>)

## DESCRIPTION

(->COMPLEX <sexpr1> <sexpr2>) returns a complex number with the real part made of the value of <sexpr1> and the imaginary part made from the value of <sexpr2>.

(->COMPLEX <sexpr>) returns a complex number with the real part made from the first element of the list value of <sexpr> and the imaginary part made from the second element of the list value of <sexpr>.

->COMPLEX switches the interpreter in complex mode.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (->complex 1 2)
1 + 2i
? (->complex '(1 2))
1 + 2i
```

# roots

## SYNOPSIS

(roots <sexpr5> <sexpr4> <sexpr3> <sexpr2> <sexpr1> <sexpr0>)

(roots <sexpr4> <sexpr3> <sexpr2> <sexpr1> <sexpr0>)

(roots <sexpr3> <sexpr2> <sexpr1> <sexpr0>)

(roots <sexpr2> <sexpr1> <sexpr0>)

(roots <sexpr1> <sexpr0>)

## DESCRIPTION

ROOTS searches for an approximates of the polynomial defined its coefficients. It solves the polynomial equations using radicals, thus accepts polynomials up to the 4th grade.

## RETURN VALUE

NUMBER

## EXAMPLES

```
; 3x^3 + 2x^2 + x + 5 = 0
? (real-mode)
? (roots 3 2 1 5)
(-1.342780)

? (complex-mode)
? (roots 3 2 1 5)
(-1.342780 0.338057+1.061566i 0.338057-1.061566i)
```

# solve

## SYNOPSIS

(solve <f> <min> <max>)

## DESCRIPTION

SOLVE searches a solution to the equation f(x) = 0 with an initial estimate between the values of <min> and <max>. The search is made using numerical analysis and is not guaranteed to be within the search range.

## RETURN VALUE

NUMBER

## EXAMPLES

```
; solving sqrt(x) - x = 0.2
? (solve (lambda (x) (- (sqrt x) x 0.2)) 0 5)
-0.2

; solving 5000(1 - e^(-x/20)) - 200x = 0
? (define F(x) (- (* 5000 (- 1 (exp (/ (- x) 20)))) (* 200 x)))
F
? (solve F 5 6)
9.284255
```

# integ

## SYNOPSIS

(integ <f> <min> <max>)

## DESCRIPTION

INTEG estimates the integral of the function f(x) within the interval defined by the values of <min> and <max>.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (integ (lambda (x) (/ (sin x) x)) 0 2)
1.605413

? (integ (lambda (x) (/ 1 x)) 1 2)
0.693147
```

# date

## SYNOPSIS

(date)

## DESCRIPTION

DATE returns the current date/time as a string with the format "YYYY-MM-DD HH:MM:SS".

## RETURN VALUE

STRING

## EXAMPLES

? (date)

2018-03-21 17:42:06

# seconds-from-epoch

## SYNOPSIS

```
(seconds-from-epoch)
```

## DESCRIPTION

SECONDS-FROM-EPOCH returns the number of seconds since 01/01/1970 as an integer. This is quite useful when random values are necessary.

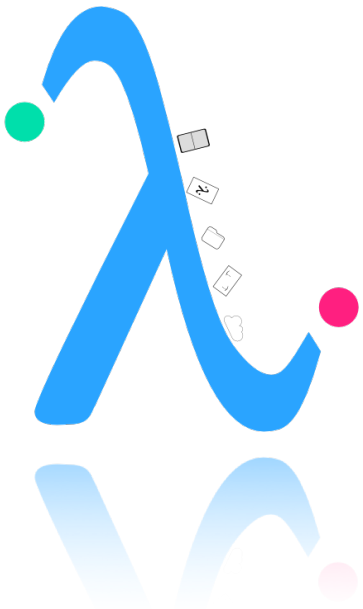
## RETURN VALUE

NUMBER

## EXAMPLES

```
? (seconds-from-epoch)
1521560312
```

# Tools.lisp



# apply

## SYNOPSIS

(apply f <expr>)

## DESCRIPTION

APPLY evaluates the expression (f <expr>).

## RETURN VALUE

EXPR

## EXAMPLES

```
? (apply '* '(3 4 5))
60

? (define compose (lambda (f g) (lambda (args) (f (apply g args)))))
compose

? ((compose sqrt *) '(12 75))
30
```

# mapcar

## SYNOPSIS

(mapcar f <list>)

## DESCRIPTION

MAPCAR applies the function f to all the elements of the <list> expression, and returns the list of the results.

## RETURN VALUE

EXPR

## EXAMPLES

```
? (mapcar '(lambda (x) (+ x x)) '(1 2 3))
(2 4 6)
```

# maplist

## SYNOPSIS

(maplist f <list>)

## DESCRIPTION

MAPLIST applies the function f to all the CDR of the <list> expression, and returns the list of the results.

## RETURN VALUE

EXPR

## EXAMPLES

? (maplist 'cdr '(1 2 3))

((2 3) (3) nil)

# evcar

## SYNOPSIS

(evcar <list>)

## DESCRIPTION

EVCAR applies the EVAL function to all the CAR of the <list> expression, and returns the result of the last evaluation.

It is a faster shortcut to:

```
(car (reverse (mapcar eval <list>)))
```

## RETURN VALUE

EXPR

## EXAMPLES

```
? (evcar '((print 1) (print 2) (print 3) (+ 3 4)))
1237
```

# let

## SYNOPSIS

(let <bindings> <action> <action>...)

## DESCRIPTION

LET creates a binding context from <bindings>, then it evaluates all the <action> expressions from left to right, and returns the value of the last one.

The <bindings> expression is a list of the form ((<name> <value>) (<name> <value> ...)) with each pair (<name> <value>) added to the binding context of the following <action> expressions; the <value> expressions are evaluated from left to right within the binding context of the caller.

It is a shortcut to the following expression when the (xi vi) expressions are not related in any way:

(progn (define x1 v1) (define x2 v2).... <action> <action>...)

## RETURN VALUE

EXPR

## EXAMPLES

```
? (let ((x 3) (y 4)) (* x y))
12

? (let ((x 2) (y 3)) (let ((x 7) (z (+ x y))) (* z x)))
35
```

# let*

## SYNOPSIS

**(let* <bindings> <action> <action>...)**

## DESCRIPTION

LET* creates a binding context from <bindings>, then it evaluates all the <action> expressions from left to right, and returns the value of the last one.

The <bindings> expression is a list of the form ((<name> <value>) (<name> <value> ...)) with each pair (<name> <value>) added to the binding context of the following <action> expressions; the <value> expressions are evaluated from left to right within the current binding context, which is the key difference between LET and LET*.

It is a shortcut to:

**(progn (define x1 v1) (define x2 v2).... <action> <action>...)**

## RETURN VALUE

EXPR

## EXAMPLES

```
? (let* ((x 3) (y 4)) (* x y))
12

? (let ((x 2) (y 3)) (let* ((x 7) (z (+ x y))) (* z x)))
70
```

# append

## SYNOPSIS

(append <list1> <list2>)

## DESCRIPTION

APPENDS returns a list made of all elements of <list1> followed by all elements of <list2>.

## RETURN VALUE

EXPR

## EXAMPLES

```
? (append '(1 2 3) '(4 5 6 7))
(1 2 3 4 5 6 7)
```

# last

## SYNOPSIS

(last <list>)

## DESCRIPTION

LAST returns the last CDR of the list <*list*>.

## RETURN VALUE

LIST

## EXAMPLES

```
? (last '(1 2 3 4 5 6))
(6)
```

# length

## SYNOPSIS

(length <list>)

## DESCRIPTION

LENGTH returns the number of elements in <*list*>. It returns 0 if NIL or not a list.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (length '(1 2 (1 2 3) 4))
4
```

# list->string

## SYNOPSIS

(list->string <list> <sep>)

(list->string <list>)

## DESCRIPTION

LIST->STRING converts the list <*list*> into a string using a separator string <*sep*> between the elements. When the separator is omitted, a space is assumed.

Note that LIST->STRING is not recursive in the sense that nested list are concatenated using their print form.

## RETURN VALUE

STRING

## EXAMPLES

```
? (list->string '(1 2 3 a b (4 5)))
1 2 3 a b (4 5)

? (list->string '(1 2 3 a b (4 5)) 'M)
1M2M3MaMbM(4 5))
```

# make-list

## SYNOPSIS

**(make-list <n> <expr>)**

## DESCRIPTION

MAKE-LIST creates a list <*n*> long filled with <*expr*>.

## RETURN VALUE

LIST

## EXAMPLES

**? (make-list 4 2)**

**(2 2 2 2 )**

**? (make-list 3 '(1 2 3))**

**((1 2 3) (1 2 3) (1 2 3))**

# nth

## SYNOPSIS

(nth <n> <list>)

## DESCRIPTION

NTH returns the <*n*>-th element of the list <*list*> where n=0 denotes the first element. If NTH gets past the last element of the list then NIL is returned.

## RETURN VALUE

EXPR

## EXAMPLES

? (nth 2 '(1 2 3 4 5 6)

3

? (nth 2 ‘((1 2) (3 4) (5 6) (7 8)))

(5 6)

# nthcdr

## SYNOPSIS

(nthcdr <n> <list>)

## DESCRIPTION

NTHCDR returns the <*n*>-th CDR element of the list <*list*> where n=0 denotes the whole list. If NTHCDR gets past the last element of the list then NIL is returned.

## RETURN VALUE

EXPR

## EXAMPLES

```
? (nthcdr 2 '(1 2 3 4 5 6)
(3 4 5 6)

? (nthcdr 2 '((1 2) (3 4) (5 6) (7 8)))
((5 6) (7 8))
```

# reverse

## SYNOPSIS

(reverse <list>)

## DESCRIPTION

REVERSE returns a list with all the elements of <*list*> in the opposite order.

## RETURN VALUE

LIST

## EXAMPLES

? (reverse '(1 2 3))

(3 2 1)

# subst

## SYNOPSIS

**(subst <new> <old> <expr>)**

## DESCRIPTION

SUBST substitutes all occurrences of <*old*> with <*new*> in the expression <*expr*>. This function is fundamental to the CLOSURE function.

## RETURN VALUE

EXPR

## EXAMPLES

**? (subst 1 'x '(x y z)**

**(1 y z)**

**? (subst 42 'a '(lambda (x y)  (+ x y a))**

**(lambda (x y) (+ x y 42))**

# subst*

## SYNOPSIS

**(subst* <patterns> <expr>)**

## DESCRIPTION

SUBST* executes SUBST with a set of patterns <*patterns*> of the form (<new1> <old1> <new2> <old2>…), thus allowing to execute substitutions in one single call.

## RETURN VALUE

EXPR

## EXAMPLES

**? (subst* (1 x 2 y 3 z) '(x y z))**

**(1 2 3)**

# unless

## SYNOPSIS

**(unless <test> <action> <action>...)**

## DESCRIPTION

UNLESS evaluates the *<test>* expression ; if it is not NIL it returns NIL; otherwise all the *<action>* expressions are evaluated from left to right and the value of the last one is returned.

It is a shortcut to:

**(if test '() <action> <action>...)**

## RETURN VALUE

EXPR or NIL

## EXAMPLES

**? (unless (eq 'a 'b) 'this 'is 'a 'set 'of 'atoms)**

**atoms**

# when

## SYNOPSIS

**(when <test> <action> <action>...)**

## DESCRIPTION

WHEN evaluates the *<test>* expression ; if it is NIL then it returns NIL; otherwise all the *<action>* expressions are evaluated from left to right and the value of the last one is returned.

It is a shortcut to:

**(if (not test) '() <action> <action>...)**

## RETURN VALUE

EXPR or NIL

## EXAMPLES

**? (when (eq 'a 'b) 'this 'is 'a 'set 'of 'atoms)**

**nil**

# repeat

## SYNOPSIS

**(repeat <counter> <action> <action>...)**

## DESCRIPTION

REPEAT repeatedly evaluates all the *<action>* expressions from left to right until the counter variable *<counter>* goes to 0. The counter is decremented at each iteration and the repeat loop is not executed if the counter is equal or lower than 0 upon the initial call.

This repeat loop is effective when some of the actions have some side effects on the environment such as producing some variation on each loop.

It is a shortcut to:

**(repeat-eval <counter> (list 'progn <action> <action>...))**

## RETURN VALUE

NIL

## EXAMPLES

```
? (define x 1)
? (repeat 5 (println "x=" x) (set! x (+ x 1)))
x = 1
x = 2
x = 3
x = 4
x = 5
```

# repeat-eval

## SYNOPSIS

**(repeat-eval <counter> <list>)**

## DESCRIPTION

REPEAT-EVAL repeatedly applies the EVAL function to the <*list*> until the counter variable <*counter*> goes to 0. The counter is decremented at each iteration and the repeat loop is not executed if the counter is equal or lower than 0 upon the initial call.

This repeat loop is effective when some of the actions have some side effects on the environment such as producing some variation on each loop."

## RETURN VALUE

NIL

## EXAMPLES

**? (define x 1)**

**? (repeat-eval 5 '(progn (println "x=" x) (set! x (+ x 2))))**

**x = 1**

**x = 3**

**x = 5**

**x = 7**

**x = 9**

# and

## SYNOPSIS

**(and <expr>...)**

## DESCRIPTION

AND returns NIL if the value of one <*expr*> is NIL, T otherwise. The expressions are evaluated from left to right and the function returns as soon as one evaluates to NIL : thus AND evaluates only the <*expr*> that are required to determine its return value.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (and (eq 'a 'a) (eq 'b 'b))
t

? (and (eq 'a 'a) (eq 'b 'a))
nil

? (and (progn (println "a") nil) (progn (println "b") 't))
a
nil

? (and (progn (println "a") 'a) (progn (println "b") 't))
a
b
t

?(and)
t
```

# not

## SYNOPSIS

**(not <expr>)**

## DESCRIPTION

NOT returns T if the value of the expression <*expr*> is NIL, NIL otherwise.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (not (eq 'a 'a))
nil

? (not (eq 'a 'b))
t

? (define (not x) (if x '() 't))
not
```

# or

## SYNOPSIS

(or <expr>...)

## DESCRIPTION

OR returns T if the value of one <*expr*> is T, NIL otherwise. The expressions are evaluated from left to right and the function returns as soon as one evaluates to T : thus OR evaluates only the <*expr*> that are required to determine its return value.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (or (eq 'a 'a) (eq 'b 'b))
t

? (or (eq 'a 'b) (eq 'b 'a))
nil

? (or (progn (println "a") nil) (progn (println "b") 't))
a
b
t

? (or (progn (println "a") 'a) (progn (println "b") 't))
a
t

? (or)
nil
```

# member?

## SYNOPSIS

(member? <expr> <list>)

## DESCRIPTION

MEMBER? returns the part of the list <*list*> starting with the expression <*expr*>, NIL if not part is matching. The tests are made against the CAR of the list, not sublists.

## RETURN VALUE

EXPR

## EXAMPLES

```
? (member? 3 '(1 2 3 4 5))
(3 4 5)

? (member? '(4 2) '(1 2 (4 2) 6 7))
((4 2) 6 7)

? (member? '(1 2) '(1 2 3 4 5))
nil
```

# comment

## SYNOPSIS

(comment <expr>...)

## DESCRIPTION

COMMENT returns its unevaluated argument expressions <*expr*>. This function is usually used to comment out chunks of valid expressions.

## RETURN VALUE

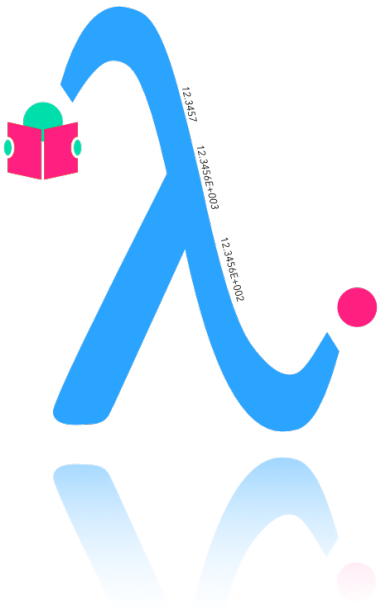
EXPR

## EXAMPLES

```
? (comment (* 6 7) (+ 4 2))
(* 6 7) (+ 4 2)
```

# Math.lisp

The Math.lisp library file contains various general-purpose mathematical functions not otherwise part of the core functions. This file is **automatically** loaded when My Lisp interpreter is started or reset.



# 0?

## SYNOPSIS

(0? <n>)

## DESCRIPTION

0? returns T if *<n>* is 0, NIL otherwise. This function is equivalent to *(= <n> 0)* and is usually used for the sake of code clarity.

Note that starting with version 1.89, this function is considered obsolete and kept only for compatibility reasons.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (0? 3)
nil

? (0? (- 3 3))
t
```

# 1?

## SYNOPSIS

(1? <n>)

## DESCRIPTION

1? returns T if *<n>* is 1, NIL otherwise. This function is equivalent to *(= <n> 1)* and is usually used for the sake of code clarity.

Note that starting with version 1.89, this function is considered obsolete and kept only for compatibility reasons.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (1? 3)
nil

? (1? (- 3 2))
t
```

# zero?

## SYNOPSIS

(zero? <n>)

## DESCRIPTION

ZERO? returns T if *<n>* is 0, NIL otherwise. This function is equivalent to *(= <n> 0)* and is usually used for the sake of code clarity.

Note that starting with version 1.89, this function is considered obsolete and kept only for compatibility reasons.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (zero? 3)
nil

? (zero? (- 3 3))
t
```

# even?

## SYNOPSIS

**(even? <n>)**

## DESCRIPTION

EVEN? returns T if <*n*> is an even number, NIL otherwise.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (even? 3)
nil

? (even? 4)
t
```

# odd?

## SYNOPSIS

(odd? <n>)

## DESCRIPTION

ODD? returns T if <*n*> is an odd number, NIL otherwise.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (odd? 3)
t

? (odd? 4)
nil
```

# 1+

## SYNOPSIS

(1+ <n>)

## DESCRIPTION

1+ returns the value of <*n*> incremented by 1. This function is equivalent to *(+ <n> 1)* and is usually used for the sake of code clarity.

Note that starting with version 1.89, this function is considered obsolete and kept only for compatibility reasons.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (1+ 3)
4
```

# 1-

## SYNOPSIS

(1- <n>)

## DESCRIPTION

1- returns the value of <*n*> decremented by 1. This function is equivalent to *(- <n> 1)* and is usually used for the sake of code clarity.

Note that starting with version 1.89, this function is considered obsolete and kept only for compatibility reasons.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (1- 3)
2
```

# fact

## SYNOPSIS

(fact <n>)

(! <n>)

## DESCRIPTION

FACT computes the factorial of the integer <*n*>.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (fact 0)
1

? (fact 5)
120

? (! 6)
720
```

# fib

## SYNOPSIS

(fib <n>)

## DESCRIPTION

FIB computes the <*n*>-th Fibonacci number.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (fib 0)
0

? (fib 1)
1

? (fib 2)
1

? (fib 8)
21
```

# egcd

## SYNOPSIS

**(egcd <m> <n>)**

## DESCRIPTION

EGCD computes the extended greatest common divisor of <*m*> and <*n*>. The result is a list (s t d) such as d = gcd(m,n) and s.m + t.n = d.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (egcd 12 5)
(-2 5 1)

? (egcd 12 -5)
(-2 -5 1)

? (egcd 12 54)
(-4 1 6)
```

# gcd

## SYNOPSIS

(gcd <m> <n>)

## DESCRIPTION

GCD computes the greatest common divisor of <*m*> and <*n*>. The result is always positive.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (gcd 12 5)
1

? (gcd 18 48)
6

? (gcd 18 -48)
6
```

# gcd_abs

## SYNOPSIS

(gcd_abs <m> <n>)

## DESCRIPTION

GCD_ABS computes the greatest common divisor of <*m*> and <*n*> that are assumed positive integers. The function does not check the validity of its parameters, thus one must ensure they are valid.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (gcd_abs 12 5)
1

? (gcd_abs 18 48)
6
```

# lcm

## SYNOPSIS

(lcm <m> <n>)

## DESCRIPTION

LCM computes the least common multiple of the expressions <*m*> and <*n*>, two integers. The computed LCD is always positive.

## RETURN VALUE

NUMBER

## EXAMPLES

? (lcm 12 5)

60

? (lcm 18 48)

144

# prime?

## SYNOPSIS

(prime? <n>)

## DESCRIPTION

PRIME? returns T if the integer <*n*> is a prime number, NIL otherwise.

**Note that the evaluation of *prime?* is delegated to the *prime-numbers::prime?* function of the primes.lisp file which is automatically loaded if required.**

## RETURN VALUE

T or NIL

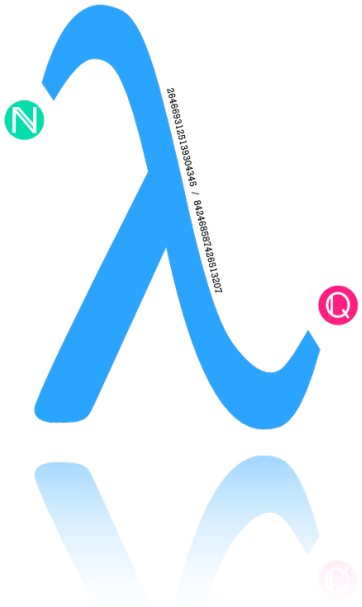
## EXAMPLES

```
? (prime? 19)
t

? (prime? 827)
t

? (prime?  201)
nil
```

# Rationals.lisp

The Rationals.lisp library file contains various functions for manipulating and representing rational numbers. This file is **automatically** loaded when My Lisp interpreter is started or reset.

For alternate representation of rational numbers, see the example files Egyptian fractions and Continued fractions.



# rational->decimal-string

## SYNOPSIS

(rational->decimal-string <fract>)

(rational->decimal-string <fract> <max-digits>)

## DESCRIPTION

RATIONAL->DECIMAL-STRING returns the decimal expansion of the rational number <*fract*>. The optional <*max-digits*> number (default to 50) indicates how many digits to use in the expansion and whether to search for the period: when positive, the period of the rational is searched into the first <*max-digits*> digits of the expansion and printed using the {period digits}… format; when negative, all the digits of the decimal expansion up to the absolute value of <*max-digits*> are printed.

Note that the function returns a symbol and not a string; this is a choice to ease the reading of the output on the console.

## RETURN VALUE

SYMBOL

## EXAMPLES

? (rational->decimal-string 1/4)

0.25

? (rational->decimal-string 1/3)

0.{3}...

? (rational->decimal-string 303/106)

2.8{5849056603773}...

? (rational->decimal-string 303/106 -20)

2.85849056603773584905...

# rational>decimal-list

## SYNOPSIS

(rational->decimal-list <fract>)

(rational->decimal-list <fract> <max-digits>)

## DESCRIPTION

RATIONAL->DECIMAL-LIST behaves as per RATIONAL->DECIMAL-STRING except that it returns the result as a list of the symbols making up the decimal representation. Actually, the RATIONAL->DECIMAL-STRING function applies the LIST->STRING function to the result of this function.

Warning that the default maximum number of digits to search the period is limited to 10 whilst it is 50 for the string representation.

## RETURN VALUE

LIST

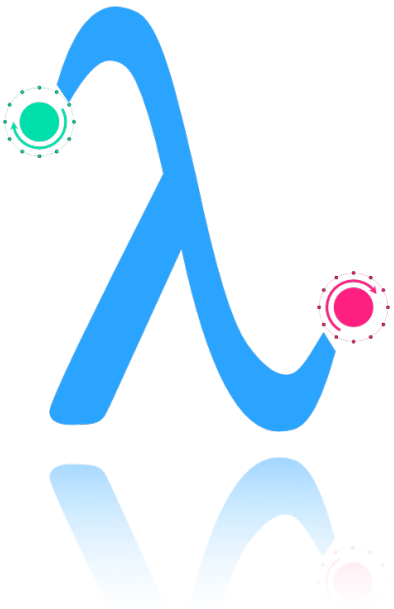
## EXAMPLES

? (Rational->decimal-list 303/106 20)

(2 . 8 { 5 8 4 9 0 5 6 6 0 3 7 7 3 }...)

# Modulus.lisp

The Modulus.lisp library file contains various functions related to modular arithmetic operations in modulus set $\mathbb{Z}/m\mathbb{Z}$. This file is **automatically** loaded when My Lisp interpreter is started or reset.



# modulus::add

## SYNOPSIS

**(modulus::add <m> <a> <b>)**

## DESCRIPTION

MODULUS::ADD adds <*a*> and <*b*> modulo <*m*>, where <*m*>, <*a*>, and <*b*> are integers.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (modulus::add 5 4 3)
2

? (modulus::add 7 -5 23)
4
```

# modulus::sub

## SYNOPSIS

**(modulus::sub <m> <a> <b>)**

## DESCRIPTION

MODULUS::SUB substracts <*b*> from <*a*> modulo <*m*>, where <*m*>, <*a*>, and <*b*> are integers.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (modulus::sub 5 4 7)
2

? (modulus::sub 7 42 3)
4

? (modulus::sub 3 1 2)
2
```

# modulus::mul

## SYNOPSIS

**(modulus::mul <m> <a> <b>)**

## DESCRIPTION

MODULUS::MUL multiplies <*a*> and <*b*> modulo <*m*>, where <*m*>, <*a*>, and <*b*> are integers.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (modulus::mul 5 4 3)
2

? (modulus::mul 5 2 -3)
4
```

# modulus::div

## SYNOPSIS

**(modulus::div <m> <a> <b>)**

## DESCRIPTION

MODULUS::DIV divides <*b*> by <*a*> modulo <*m*>, where <*m*>, <*a*>, and <*b*> are integers. The division is valid if and only if <*b*> has an inverse modulo <*m*>.

## RETURN VALUE

NUMBER

## EXAMPLES

**? (modulus::div 5 4 3)**

**3**

**? (modulus::div 6 4 3)**

**nan**

# modulus::expt

## SYNOPSIS

**(modulus::expt <m> <a> <b>)**

## DESCRIPTION

MODULUS::EXPT raises <*a*> to the power of <*b*> modulo <*m*>, where <*m*>, <*a*>, and <*b*> are integers. When <*b*> is negative, the function is valid if and only if <*a*> has an inverse modulo <*m*>.

## RETURN VALUE

NUMBER

## EXAMPLES

**? (modulus::expt 5 2 3)**

**3**

**? (modulus::expt 5 2 -3)**

**2**

**? (modulus::expt 6 2 -3)**

**nan**

# modulus::inverse

## SYNOPSIS

**(modulus::inverse <m> <a>)**

## DESCRIPTION

MODULUS::INVERSE computes the inverse of <*a*> modulo <*m*>, where <*m*> and <*a*> are integers. An inverse exists if and only if <*m*> is coprime with <*a*>.

## RETURN VALUE

NUMBER

## EXAMPLES

**? (modulus::inverse 7 3)**

**5**

**? (modulus::inverse 28 15)**

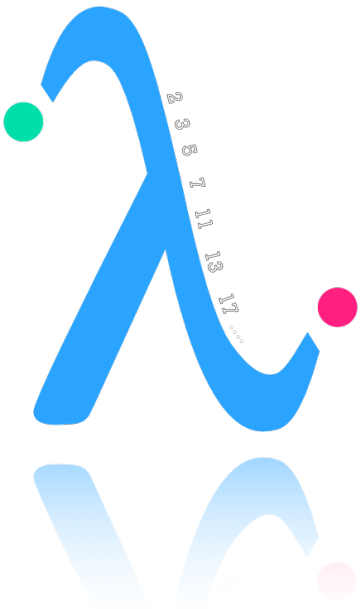
**15**

**? (modulus::inverse 28 7)**

**nan**

# Primes.lisp

The Primes.lisp library file contains various functions related to prime numbers and number theory. This file is **automatically** loaded when the *prime?* function is evaluated.



# prime-numbers::known-primes

## SYNOPSIS

**prime-numbers::known-primes**

## DESCRIPTION

PRIME-NUMBERS::KNOWN-PRIMES contains the list of all the known prime numbers. This list grows over time as new prime numbers are discovered. This list is stored in the data bindings context and thus maintained across sessions ; it is however reset with all primes lower than 9999 when the file primes.lisp is loaded or reloaded.

## RETURN VALUE

N/A

## EXAMPLES

**? (load "#primes")**

**primes**

**? prime-numbers::known-primes**

**(2 3 5 7 …. 9973)**

# prime-numbers::prime-to-known?

## SYNOPSIS

**(prime-numbers::prime-to-known? <n>)**

## DESCRIPTION

PRIME-NUMBERS::PRIME-TO-KNOWN? determines whether the number <*n*> is relatively prime with all known prime numbers of *prime-numbers::known-primes*. The response of the function will depend on how many primes are currently known; however when nil is returned, the result will always be nil unless the list of the known prime is reset.

Note that a number that is not relatively prime is not prime.

## RETURN VALUE

NUMBER

## EXAMPLES

**? (prime-numbers::prime-to-known? 997)**

**t**

**? (prime-numbers::prime-to-known? 5389)**

**nil**

# prime-numbers::prime?

## SYNOPSIS

(prime-numbers::prime? <n>)

## DESCRIPTION

PRIME-NUMBERS::PRIME? determines whether the number <*n*> is a prime number. The set of the known primes will grow if it is required to determine the result.

## RETURN VALUE

T or NIL

## EXAMPLES

```
? (prime-numbers::prime?  997)
t

? (prime-numbers::prime? 5389)
nil

? (prime-numbers::prime? 10001)
nil

? (prime-numbers::prime? 12041)
t
```

# prime-numbers::nth

## SYNOPSIS

(prime-numbers::nth <n>)

## DESCRIPTION

PRIME-NUMBERS::NTH returns the <*n*>-th prime number, counting from 1. The list of known primes is extended if required such as it contains at least <n> elements after the function returns.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (prime-numbers::nth 1)
2

? (prime-numbers::nth 2)
3

? (prime-numbers::nth 100)
541

? (prime-numbers::nth 1442)
12041
```

# prime-numbers::primorial

## SYNOPSIS

(prime-numbers::primorial <n>)

## DESCRIPTION

PRIME-NUMBERS::PRIMORIAL returns the product of the prime numbers lower than or equal to the number <*n*>. This function is known as the primorial function and sometime denoted #. See https://en.wikipedia.org/wiki/Primorial for details.

## RETURN VALUE

NUMBER

## EXAMPLES

? (prime-numbers::primorial 5)

30

? (prime-numbers::primorial 100)

2305567963945518424753102147331756070

# prime-numbers::product

## SYNOPSIS

(prime-numbers::product <n>)

## DESCRIPTION

PRIME-NUMBERS::PRODUCT returns the product of the first <*n*>-th prime numbers. This product is a practical number.

## RETURN VALUE

NUMBER

## EXAMPLES

? (prime-numbers::product 5)

2310

? (prime-numbers::product 20)

557940830126698960967415390

# prime-numbers::factors

## SYNOPSIS

(prime-numbers::factors <n>)

## DESCRIPTION

PRIME-NUMBERS::FACTORS returns a list with all the prime numbers that are necessary to factorize the number <*n*>. A factor is repeated as many time as necessary when its power is a factor. The list is always sorted from the lowest factor to the highest.

## RETURN VALUE

LIST

## EXAMPLES

? (prime-number::factors 12)

(2 2 3)

? (prime-number::factors 41)

(41)

? (prime-numbers::factors 34866)

(2 3 3 13 149)

? (prime-number::factors 42)

(2 3 7)

? (prime-number::factors 30031)

(59 509)

# prime-numbers::factors*

## SYNOPSIS

(prime-numbers::factors* <n>)

## DESCRIPTION

PRIME-NUMBERS::FACTORS* returns a list with all the prime numbers that are necessary to factorize the number <*n*>. The factors are represented by lists (p e) where p is the factor and e its associated exponentiation. The list is always sorted from the lowest factor to the highest.

## RETURN VALUE

LIST

## EXAMPLES

? (prime-number::factors* 12)

((2 2) (3 1))

? (prime-number::factors* 41)

((41 1))

? (prime-numbers::factors* 34866)

((2 1) (3 2) (13 1) (149 1))

? (prime-number::factors* 42)

((2 1) (3 1) (7 1))

? (prime-number::factors* 30031)

((59 1) (509 1))

# prime-numbers::divisors

## SYNOPSIS

(prime-numbers::divisors <n>)

## DESCRIPTION

PRIME-NUMBERS::DIVISORS returns the list of all the divisors of the number <*n*>. This list is always sorted from the lowest (1) to the highest (<n>).

## RETURN VALUE

LIST

## EXAMPLES

? (prime-numbers::divisors 12)

(1 2 3 4 6 12)

? (prime-numbers::divisors 41)

(1 41)

? (prime-numbers::divisors 42)

(1 2 3 6 7 14 21 42)

# prime-numbers::practical?

## SYNOPSIS

(prime-numbers::practical? <n>)

## DESCRIPTION

PRIME-NUMBERS::PRACTICAL? determines whether the number <*n*> is a pratical number. See https://fr.wikipedia.org/wiki/Nombre_pratique for details on practical numbers.

## RETURN VALUE

T OR NIL

## EXAMPLES

? (prime-numbers::practical? 12)

t

? (prime-numbers::practical? 42)

nil

? (prime-numbers::practical? 44)

nil

# prime-numbers::find-sum-from-divisors

## SYNOPSIS

**(prime-numbers::find-sum-from-divisors <n> <Nk>)**

## DESCRIPTION

PRIME-NUMBERS::FIND-SUM-FROM-DIVISORS determines a set of distinct divisors of the <*Nk*> such as their sum is <*n*>. The function does not assume that <*Nk*> is a pratical number, and when no sum can be found, nil is returned. The divisors are always sorted from the lowest to the highest.

## RETURN VALUE

LIST

## EXAMPLES

**? (prime-numbers::find-sum-from-divisors 7 12)**

**(1 6)**

**? (prime-numbers::find-sum-from-divisors 41 42)**

**(6 14 21)**

**? (prime-numbers::find-sum-from-divisors 41 44)**

**nil**

# prime-numbers::find-sum-from-divisors*

## SYNOPSIS

(prime-numbers::find-sum-from-divisors* <n> <Nk>)

## DESCRIPTION

PRIME-NUMBERS::FIND-SUM-FROM-DIVISORS* determines all sets of distinct divisors of the <*Nk*> such as their sum is <*n*>. The function does not assume that <*Nk*> is a pratical number, and when no sum can be found, nil is returned. The divisors are always sorted from the lowest to the highest.

## RETURN VALUE

LIST

## EXAMPLES

? (prime-numbers::find-sum-from-divisors* 7 12)

((1 6) (1 2 4) (3 4))

? (prime-numbers::find-sum-from-divisors* 41 42)

((1 2 3 14 21) (6 14 21))

? (prime-numbers::find-sum-from-divisors* 41 44)

nil

# prime-numbers::find-sum-from-list

## SYNOPSIS

(prime-numbers::find-sum-from-list <n> <L>)

## DESCRIPTION

PRIME-NUMBERS::FIND-SUM-FROM-LIST determines a set of distinct elements of the list <*L*> such as their sum is <*n*>. When no sum can be found, nil is returned. The divisors are always sorted from the lowest to the highest.

## RETURN VALUE

LIST

## EXAMPLES

? (prime-numbers::find-sum-from-list 7 '(1 2 3 4 5 6))

(1 6)

# prime-numbers::find-sum-from-list*

## SYNOPSIS

(prime-numbers::find-sum-from-list* <n> <L>)

## DESCRIPTION

PRIME-NUMBERS::FIND-SUM-FROM-LIST* determines all sets of distinct elements of the list <*L*> such as their sum is <*n*>. When no sum can be found, nil is returned. The divisors are always sorted from the lowest to the highest.

## RETURN VALUE

LIST

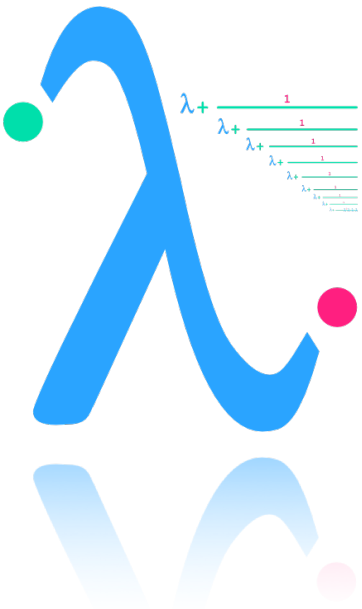
## EXAMPLES

? (prime-numbers::find-sum-from-list* 7 '(1 2 3 4 5 6))

((1 6) (2 5) (1 2 4) (3 4))

# Continued fractions.lisp

The Continued fractions file contains various functions for the representation of rational numbers as continued fractions.



# rational->continued-fraction

## SYNOPSIS

(rational->continued-fraction <fract>)

## DESCRIPTION

RATIONAL->CONTINUED-FRACTION returns the list of the numbers making up the continued fraction of the rational <*fract*>. The first number of the list is the integer part of the fraction. When the fraction is not 1, then the last number of the list is not 1.

## RETURN VALUE

LIST

## EXAMPLES

? (rational->continued-fraction 12/5)

(2 2 2)

# rational<-continued-fraction

## SYNOPSIS

(rational<-continued-fraction <L>)

## DESCRIPTION

RATIONAL->CONTINUED-FRACTION creates a rational from a list of integers making up the continued fraction development of the rational.

This is the inverse of the *rational->continued-fraction* function.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (rational<-continued-fraction '(2 2 2))
12/5

? (rational<-continued-fraction '(2 3 1 4))
43/19

? (rational<-continued-fraction '(1 2 2 2))
17/12

? (rational<-continued-fraction '(1 2 2 1 1))
17/12
```

# quadratic->continued-fraction

## SYNOPSIS

(quadratic->continued-fraction <n> <a> <b> <p>)

(quadratic->continued-fraction <n>)

## DESCRIPTION

QUADRATIC->CONTINUED-FRACTION returns a list with the first <*p*> numbers making up the continued fraction of a quadratic number $a.\sqrt{n} + b$ where <*n*> is an integer and <*a*> and <*b*> rationals.

If <*n*> is not a perfect square then the list is limited to the first <*p*> elements; otherwise the full continued fraction development of the rational is returned.

The second form of the function is equivalent to *(quadratic->continued-fraction n 1 0 20)*.

## RETURN VALUE

LIST

## EXAMPLES

? (quadratic->continued-fraction 2)

(1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2)

? (quadratic->continued-fraction 3)

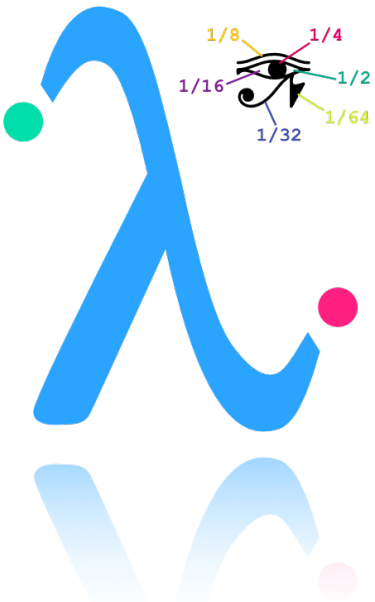
(1 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1)

? (quadratic->continued-fraction 5 1/2 1/2 10)

(1 1 1 1 1 1 1 1 1 1)

# Egyptian fractions.lisp

The Egyptian fractions.lisp library file contains functions and algorithms around the egyptian fraction representation of rational numbers.



# rational->egyptian-fraction

## SYNOPSIS

(rational->egyptian-fraction <fract> <algorithm>)

(rational->egyptian-fraction <fract>)

## DESCRIPTION

RATIONAL->EGYPTIAN-FRACTION returns a list of integers making up the egyptian fraction representation of the fraction <*fract*>. The first term of the list is the integer part of the fraction and the rest the inverses of decomposition; thus (1 2) is the number 1+½, that is 1.5.

The optional <*algorithm*> is either the name of the decomposition method or the decomposition method itself; when the method is given, its invocation is *(algorithm f)* where f is a rational number strictly lower than 1.

## RETURN VALUE

LIST

## EXAMPLES

? (rational->egyptian-fraction 3/2)

(1 2)

? (rational->egyptian-fraction 4/17)

(0 5 29 1233 3039345)

? (rational->egyptian-fraction 4/17 'erdös)

(0 6 17 102)

# rational->egyptian-fraction::fibonacci

## SYNOPSIS

(rational->egyptian-fraction::fibonacci <fract>)

## DESCRIPTION

RATIONAL->EGYPTIAN-FRACTION::FIBONACCI uses the Fibonacci (aka, Sylvester or greedy) algorithm to find the Egyptian fraction terms of a fraction lower than 1.

This function is called when the algorithm for the RATIONAL->EGYPTIAN-FRACTION is not defined, Fibonacci, Sylvester, or Greedy.

## RETURN VALUE

LIST

## EXAMPLES

? (rational->egyptian-fraction::fibonacci 3/7 )

(3 11 231)

? (rational->egyptian-fraction 3/7 'fibonacci)

(0 3 11 231)

# rational->egyptian-fraction::golomb

## SYNOPSIS

(rational->egyptian-fraction::golomb <fract>)

## DESCRIPTION

RATIONAL->EGYPTIAN-FRACTION::GOLOMB uses the Golomb algorithm to find the Egyptian fraction terms of a fraction lower than 1.

This function is called when the algorithm for the RATIONAL->EGYPTIAN-FRACTION is Golomb.

## RETURN VALUE

LIST

## EXAMPLES

? (rational->egyptian-fraction::golomb 3/7 )

(3 15 35)

? (rational->egyptian-fraction 3/7 'golomb)

(0 3 15 35)

# rational->egyptian-fraction::splitting

## SYNOPSIS

(rational->egyptian-fraction::splitting <fract>)

## DESCRIPTION

RATIONAL->EGYPTIAN-FRACTION::SPLITTING uses the Splitting algorithm to find the Egyptian fraction terms of a fraction lower than 1.

This function is called when the algorithm for the RATIONAL->EGYPTIAN-FRACTION is Splitting.

Note that the algorithm may expand to a very large number of terms. The algorithm generates an error when this number reaches a threshold (100 by default) stored in the variable rational->egyptian-fractions::splitting-max-terms.

## RETURN VALUE

LIST

## EXAMPLES

? (rational->egyptian-fraction::splitting 3/7 )

(7 8 9 56 57 72 3192)

? (rational->egyptian-fraction 3/7 'splitting)

(0 7 8 9 56 57 72 3192)

# rational->egyptian-fraction::binary

## SYNOPSIS

(rational->egyptian-fraction::binary <fract>)

## DESCRIPTION

RATIONAL->EGYPTIAN-FRACTION::BINARY uses the Binary algorithm to find the Egyptian fraction terms of a fraction lower than 1. This is mainly the practical algorithm limited to practical numbers that are a power of 2.

This function is called when the algorithm for the RATIONAL->EGYPTIAN-FRACTION is Binary.

## RETURN VALUE

LIST

## EXAMPLES

? (rational->egyptian-fraction::binary 3/7 )

(4 8 28 56)

? (rational->egyptian-fraction 3/7 'binary)

(0 4 8 28 56)

? (rational->egyptian-fraction::binary 5/21 )

(8 16 32 84 168 672)

? (rational->egyptian-fraction 5/21 'binary)

(0 8 16 32 84 168 672)

# rational->egyptian-fraction::primorial

## SYNOPSIS

(rational->egyptian-fraction::primorial <fract>)

## DESCRIPTION

RATIONAL->EGYPTIAN-FRACTION::PRIMORIAL uses the practical numbers algorithm limited to primorials to find the Egyptian fraction terms of a fraction lower than 1. This would be equivalent to the Bleicher/Erdös algorithm if not for the adjustment of the rest in the modulo/quotient computation.

This function is called when the algorithm for the RATIONAL->EGYPTIAN-FRACTION is Primorial.

## RETURN VALUE

LIST

## EXAMPLES

? (rational->egyptian-fraction::primorial 3/7 )

(3 15 35)

? (rational->egyptian-fraction 3/7 'primorial)

(0 3 15 35)

# rational->egyptian-fraction::erdos

## SYNOPSIS

(rational->egyptian-fraction::erdos <fract>)

## DESCRIPTION

RATIONAL->EGYPTIAN-FRACTION::ERDOS uses the Bleicher/Erdös algorithm to find the Egyptian fraction terms of a fraction lower than 1.

This function is called when the algorithm for the RATIONAL->EGYPTIAN-FRACTION is one of Bleicher-Erdos, Bleicher-Erdos, Bleicher, Erdos, or Erdös.

## RETURN VALUE

LIST

## EXAMPLES

? (rational->egyptian-fraction::erdos 3/7 )

(3 14 42)

? (rational->egyptian-fraction 3/7 'erdos)

(0 3 14 42)

? (rational->egyptian-fraction::erdos 5/121 )

(0 30 242 605 726 1210)

? (rational->egyptian-fraction 5/121 'erdös)

(0 30 242 605 726 1210)

# rational->egyptian-fraction::practical

## SYNOPSIS

(rational->egyptian-fraction:practical <fract> <findNk>)

## DESCRIPTION

RATIONAL->EGYPTIAN-FRACTION::PRACTICAL uses the base algorithm with practical numbers to find the Egyptian fraction terms of a fraction lower than 1. The *<findNk>* parameter is a function to compute the sequence of practical numbers to use. See the binary and primorial functions for examples.

## RETURN VALUE

LIST

## EXAMPLES

```
(define (find-2n p)
        (define (find n p)
                (if (>= n p) n
                        (find (* 2 n) p))
        )

        (find 2 p)
)

? (rational->egyptian-fraction::practical 3/7 find-2n)
(4 8 28 56)
```

# rational<-egyptian-fraction

## SYNOPSIS

(rational<-egyptian-fraction <L>)

## DESCRIPTION

RATIONAL<-EGYPTIAN-FRACTION computes the rational number defined by its terms in the list <L>. The first term of the list is the integer part of the rational and the rest is made of the inverses of the egyptian fraction representation of its decimal part. This function is the inverse of the RATIONAL->EGYPTIAN-FRACTION; the other way around is not true as the egyptian fraction decomposition is never unique.

## RETURN VALUE

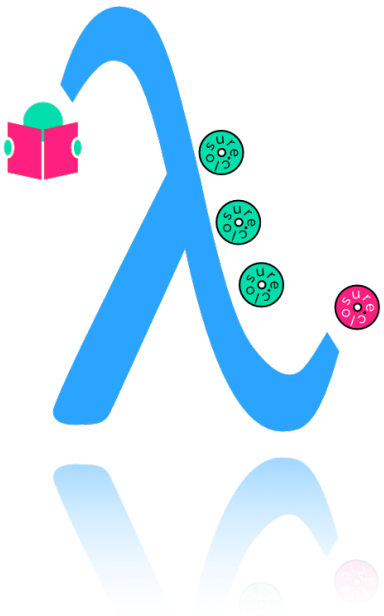
NUMBER

## EXAMPLES

```
? (rational<-egyptian-fraction '(0 2 3))
5/6

? (define x <whatever rational>)
x

? (= x (rational<-egyptian-fraction (rational->egyptian-fraction x)))
t
```

# Le_Lisp.lisp

The Le_Lisp.lisp library file contains various functions borrowed from Le Lisp implementation of Lisp. Its most important functions are *gensym* and *closure* that allow adding lexical binding (aka closure) to lambda expressions. These 2 functions are so much useful that they are actually available in standalone files, gensym.lisp and closure.lisp and loaded **automatically** when My Lisp interpreter is started or reset.



# closure

## SYNOPSIS

(closure <symbols> <expr>)

## DESCRIPTION

CLOSURE returns the expression <*expr*> with various symbols replaced with newly allocated symbols, thus allowing lexical bindings for all symbols. The <*symbols*> argument is a list of symbols that will be substituted by *gensym*'d symbols within <*expr*>.

## RETURN VALUE

SYMBOL

## EXAMPLES

```
? (define (make-adder init)
        (let ((counter init))
                (closure '(counter)
                        (lambda ((val 0))
                                (progn
                                        (if val (set! counter (+ counter val)))
                                        counter)))))
make-adder

? (definer counter (make-adder 3))
counter

? (counter 4)
7

? (counter)
7
```

### Natural number generator:

```
(define nextint
        (let ((n 0))
                (closure '(n)
                        (lambda ()
                                (progn
                                        (set! n (+ n 1))
                                        n)))))


? (nextint) (nextint) (nextint)
1
2
3
```

### Fibonacci numbers generator:

```
(define nextfib
        (let ((x 0) (y 1))
                (closure '(x y)
                        (lambda ()
                                (progn
                                        (define res y)
                                        (set! y (+ x y))
                                        (set! x res)
                                        res)))))


? (nextfib) (nextfib) (nextfib) (nextfib) (nextfib) (nextfib)
1
1
2
3
5
8
```

### Composition of functions:

```
(define (compose f g)
        (closure '(f g)
                (lambda (x) (f (g x)))))
```

We can now use the compose function to create the function $e^{1/x}$:

```
? (define ei (compose exp /))
ei

? (ei 10)
1.1052

? (ei 1)
2.7183
```

### Currying of functions:

```
(define (curry f . args)
        (closure '(f args)
                (lambda (. more-args)
                        (apply f (append args more-args)))))

? (define add4 (curry + 4))
add4

? (add4 5)
9

? (define (muldiff a b c) (* (+ a c) (- b c)))
muldiff

? (define muldiff4_5 (curry muldiff 4 5))
muldiff4_5

? (muldiff4_5 1)
20
```

### Flip-flop:

```
(define flip-flop
        (let ((x 1))
                (closure '(x)
                        (lambda()
                                (progn
                                        (set! x (- 1 x))
                                        x)))))

? (flip-flop)
0

? (flip-flop)
1
```

### Lexically scoped let:

The following *clet* function is the same as the standard *let* one except that it is lexically scoped, thus the value of its variables do not depend the environment once after the *clet* function has been evaluated. This makes this definition of *clet* compatible with the one found in Scheme.

```
(define (clet::env clet::vars clet::init clet::varinits)
        (if (null? clet::varinits) (cons clet::vars clet::init)
                (clet::env
                        (cons (caar clet::varinits) clet::vars)
                        (cons (list 'define (caar clet::varinits)
                                (list 'quote (eval (cadar clet::varinits)))) clet::init)
                        (cdr clet::varinits))
))

(define (clet ?clet::vars . ?clet::body)
(progn
        (define init (clet::env nil nil ?clet::vars))
```

```
        (define pvar (car init))
        (define init (cdr init))
        (eval (cons 'progn init))
        (eval (cons 'progn (closure pvar ?clet::body)))
))

? (define cp (clet ((x 1) (y 2)) (lambda (z) (+ x y z))))
cp

? (cp 3)
6

? (define x 5)
x

? (cp 3)
6
```

# gensym

## SYNOPSIS

(gensym <name>)

(gensym)

## DESCRIPTION

GENSYM is used to generate new symbols. Each time it is called, GENSYM returns a new symbol of the form G*xxx* where *xxx* is a sequential counter number. The prefix G is stored within the global variable *gensym::key* and the prefix symbol G within the global variable *gensym::prefix*. When the GENSYM function is called with a string parameter *name* then this string is used in place of the standard prefix.

The *gensym::key* and *gensym::prefix* symbols are stored in the data environment.

## RETURN VALUE

SYMBOL

## EXAMPLES

? (gensym)

G1101

? (gensym)

G1102

? (let ((gensym::prefix "etiq")) (gensym))

etiq1103

? (gensym “whatever”)

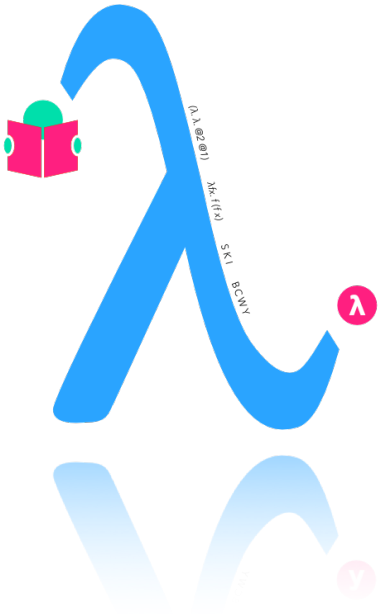
whatever1104

# Lambda Calculus.lisp

The example file Lambda Calculus adds definitions related to the lambda calculus and combinators theories. They allow the transformation of lambda calculus expressions written as My Lisp expressions with currying, alpha-conversion, beta-reduction, SK combinators rewriting, etc. The de Bruijn notation is also used to assert equivalence between expressions.

All functions directly related to the lambda calculus example file are prefixed with the *lc::* sequence. All lambda expressions directly related to the Church expressions are prefixed with the *church::* sequence.

The source code contains regression tests that is a good reference to learn the usage of most of the functions.



# Lambda expressions

My Lisp evaluations rules are still valid when running the Lambda Calculus functions; thus it is important to prevent the standard *eval* function from evaluating lambda expressions upon reading. You can use quotation but most likely you will need to use the special non evaluating function *lc::expr* when nested definitions are used:

```
? (lc::eval (lc::expr lc::B (lc::expr lc::B lc::W) (lc::expr lc::B lc::B lc::C)))
S

? (lc::redux* (lc::expr church::succ (lc::expr church::succ church::0)))
(lambda (f x) (f (f x)))
```

✓ The *lc::expr* function is nothing but the standard *list* function.

## alpha conversion

The *(lc::subst M x N)* function substitutes the term *x* for *N* in *M*, assuming that *N* and subterms of *M* have no common free variables. The *(lc::alpha M x N)* function substitutes the term *x* for *N* in *M* with explicit alpha-conversion whenever necessary:

```
? (lc::subst (lambda (y) (y z)) 'z (lambda (y) (x y y)))
 (lambda (y) (y (lambda (y) (x y y))))

? (lc::subst (lambda (y) (y z)) 'z (lambda (y) (x y y)))
 (lambda (y) (y (lambda (y) (x y y))))

? (lc::subst (lambda (y) (y z)) 'z (lambda (y) (x y y)))
 (lambda (y) (y (lambda (y) (x y y))))
```

In the previous second example the *lc::subst* is usually considered incorrect because the free-variable *x* in *(lambda (y) (x y y))* becomes a bound variable after the substitution. When the *lc::alpha* function is used instead, the alpha conversion is applied against the *x* term of the leading lambda expression *(lambda (x)...)* such as using the term *@x1*, thus returning a proper result.

## beta conversion and redux

The *(lc::beta M N)* function applies the term *N* to the lambda expression *M* and the *(lc::beta* M . N)* function applies all the terms of the implicit list *N* to *M*. Both functions are guaranteed to terminate because they only reduce once per argument N:

```
? (lc::beta lc::S 'M)
 (lambda (y z) (M z (y z)))

? (lc::beta* (lambda (x y z) (x z (y z))) 'M 'N)
 (lambda (z) (M z (N z)))
```

The *(lc::redux M)* function tries to reduce once the lambda expression *M*, that is applying the *lc::beta* function onto itself when possible. The *(lc::redux* M)* tries to invoke the reduction for a maximum given number of iterations indicated by the variable *lc::redux*::max-iterations* (initially set to 10) thus as avoiding infinite loops when handling terms without normal form. Here are a few examples:

```
? (lc::redux* '((lambda (x y z) ((x z) (y z))) (lambda (x y) x) (lambda (x y) x)))
(lambda (z) z)

? (set! lc::redux*::max-iterations 10)
lc::redux*::max-iterations

? (lc::redux* (lc::expr church::pred (lc::expr church::succ church::2)))
(lambda (f x) (f (f ((lambda (u) x) f))))

? (set! lc::redux*::max-iterations 20)
lc::redux*::max-iterations

? (lc::redux* (lc::expr church::pred (lc::expr church::succ church::2)))
(lambda (f x) (f (f x)))
```

In the previous examples, the last 2 *lc::redux** calls returned different values because the *lc::redux* function was applied a different number of times.

## de Bruijn notation

The *(lc::indexed M)* function returns the lambda expression M using the de Bruijn notation. This notation is quite important because it can be used to assert whether two lambda expressions are the same without any need to take into account alpha conversion:

```
? (lc::indexed (lambda (x y) (x y)))
(λ. λ. @2 @1)

? (lc::indexed (lambda (x y z) ((x y) (lambda (t) (t x z)))))
(λ. λ. λ. @3 @2 (λ. @1 @4 @2))
```

# Combinators

The Lambda Calculus file also adds classical S and K combinators with functions to manipulate them. Underneath combinators are lambda expressions and essentially interpreted as such.

## Conversion

The *(lc::->sk M)* function converts the lambda expression *M* into its equivalent sequence of *S* and *K* symbols. The optional second parameter is a list with the symbols to use for *S* and *K*, and optionally *I* (intended as *SKK*) when the standard *SKK* simplification is required or needed. The result sequence is hardly optimized (in the sense of minimum number of terms) and the function itself makes sense when M has no free variables.

```
? (lc::->sk (lambda (x y z) (z y x)))
(S (K (S (S (K S) (S (K (S (S K K))) (S (K K) (S K K)))))) (S (K K) (S (K K) (S K K))))

? (lc::->sk (lambda (x y z) (z y x)) '(S K I))
(S (K (S (S (K S) (S (K (S I)) (S (K K) I))))) (S (K K) (S (K K) I)))
```

The *(lc::parse expr)* function converts a sequence of combinators into a lambda expression. The combinators may be any of those known to the system, that is *S*, *K*, *I*, *B*, *C*, *W*, and *Y* (see the *lc::register-term* function):

```
? (lc::parse '(S K S))
((lambda (x y z) (x z (y z))) (lambda (x y) x) (lambda (x y z) (x z (y z))))

? (lc::eval (lc::parse '(S K S)))
I

? (lc::indexed (lc::redux* (lc::parse '(B (B  W) (B B)))))
(λ. λ. λ. @3 @2 (@1 @1))
```

## Evaluations

The *lc::known-terms* symbol is a dictionary of all the known terms. The keys are user names and the values lists associated to these names, with the first element the lambda expressions and the second element their indexed (de Bruijn) representations. The standard combinators *S*, *K*, *I*, *B*, *C*, *W*, and *Y* are part of the known symbols. You can add a lambda expression to the dictionary using the function *(lc::register-term name expr)*:

```
? church::1
(lambda (f x) (f x))

? (lc::register-term '1 church::1)
(λ. λ. @2 @1)

? (dictionary::find lc::known-terms 1)
((lambda (f x) (f x)) (λ. λ. @2 @1))
```

The *(lc::eval M)* tries to evaluate the lambda expression *M* into a known term; if the term is not associated to with a *lc::known-term* entry then its indexed form is returned; the function uses the *lc::redux** function to reduce *M*, thus the result depends on the parameter *lc::redux*::max-iterations*:

```
? (lc::eval (lc::expr lc::B (lc::expr lc::B lc::W) (lc::expr lc::B lc::B lc::C)))
S

? (lc::eval (lc::parse '(B (B W) (B B C))))
S
```

# Church encoding

The Lambda Calculus file also provides a few expressions based on Alonzo Church encoding, with *church::0*, *church::1*, *church::add*, etc. for numbers, *church::cons*, *church::head*, and *church::tail* for lists, and so on. Note that none of the church-prefixed definitions are part of the *lc::known-terms* dictionary, thus you will need to add them if you want to evaluate expressions into them.

```
? (lc::eval (lc::expr church::tail (lc::expr church::cons 'a 'b)))
b

? (lc::register-term '1 church::1)
(λ. λ. @2 @1)

? (lc::eval (lc::expr church::pred church::2))
1

? (lc::eval (lc::expr church::pred (lc::expr church::succ church::2)))
(lambda (f x) (f (f x)))

? (lc::register-term '2 church::2)
(λ. λ. @2 (@2 @1))

? (lc::eval (lc::expr church::pred (lc::expr church::succ church::2)))
2
```

Note that if the last *lc::eval* functions did not returned the expected result, you should check whether the variable *lc::redux*::max-iterations* is set to at least 11 iterations.

# Utilities

## new-arg

The *(lc::new-arg)* function returns a new symbol of the form *@xi* when *i* is a positive integer starting from 1. The *(lc::new-arg-reset)* allows restarting the *i* counter from 1. This function is used to avoid name conflict during explicit alpha conversions and assumes that no user symbol is of the form *@x1*, *@x2*, etc.

## free-vars

The *(lc::free-vars M)* function returns the list of the free variables of *M* whilst *(lc::free-var? x M)* determines whether *x* is a free variable of *M*:

```
? (lc::free-vars (lc::expr (lambda (x y z) (+ x a b y))))
(b a +)
```

## curry

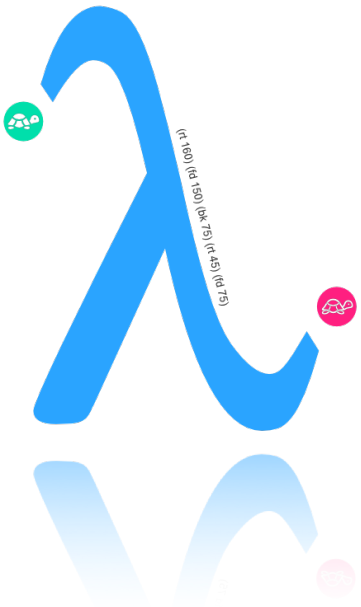
The *(lc::curry M)* function returns the lambda expression *M* as a composition of lambda expressions with no more than one argument each:

```
? (lc::curry '(lambda (x y) (x y)))
(lambda (x) (lambda (y) (x y)))

? (lc::curry '(lambda (x y) (x (lambda (t z) (y t (x z))))))
(lambda (x) (lambda (y) (x (lambda (t) (lambda (z) (y t (x z)))))))
```

# Turtle graphics

The turtle graphics engine allows drawing graphics using the LOGO language set of commands: the system renders the tracks of a turtle that is moved onto the screen using simple commands. Though the set of commands is simple and intuitive, you can obtain very complex graphics like Hilbert curves or Barnsley's fern.

My Lisp provides a lot of examples, most of which are extracted from the book “Turtle Geometry” by Harold Abelson and Andrea diSessa.



## Logo language

My Lisp implements most of the commands of the LOGO language and its variants, either as built-in functions or library ones loaded when the interpreter starts. The special library file "turtle-shortcuts" further simplifies the commands using the common abbreviations; this is especially useful when working on an iPhone.

## Colors

A color is either a string or symbol (black, white, red, …), or a list of floats ranging 0..1 and defining an RGBA color; the alpha component is optional. The known names are:

- white, black, gray
- red, orange,  yellow, green, mint, teal, cyan, blue, indigo, purple, pink, brown,
- gray1, gray2, gray3, gray4, gray5, gray6

The colors are adjusted depending on the dark or light mode of the device.

## Shortcuts

The file library file turtle-shortcuts.lisp contains common LOGO shortcuts that may simplify the writing of LOGO programs on small devices or when porting from LOGO programs:

| shortcut | shortcut alt | function |
|---|---|---|
| pc | | turtle::pen-color |
| pc# | | turtle::pen-color using the n-th system color |
| bc | | turtle::background-color |
| bc# | | turtle::background-color using the n-th system color. |
| | home | turtle::home |
| lt | left | turtle::left |
| rt | right | turtle::right |
| fd | forward | turtle::forward |
| bk | back | turtle::backward |
| pu | penup | turtle::pen-up |
| pd | pendown | turtle::pen-down |
| mv | move-to | turtle::move-to |

# turtle::arc

## SYNOPSIS

**(turtle::arc <degrees> <radius>)**

## DESCRIPTION

TURTLE::ARC draws an arc of a circle. The center of the circle is the current position of the turtle. The arc is drawn with the given radius <radius>. It starts at the current heading of the turtle and continues clockwise for the given number of degrees <degrees> if <degrees> is positive, or counter-clockwise when <degrees> is negative. The turtle does not move.

## RETURN VALUE

T or NIL

## EXAMPLES

**draw circles:**

```
(turtle::arc 360 10)
(turtle::arc 360 20)
(turtle::arc 360 30)
```

# turtle::background-color

## SYNOPSIS

(turtle::background-color <color>)

(turtle::background-color)

## DESCRIPTION

TURTLE::BACKGROUND-COLOR sets the background color of the drawing canvas. When no color is given, the default system background color is used (white in light mode, black in dark mode). See the color section at the beginning of the chapter for interpreting the <color> parameter.

## RETURN VALUE

SEXPR

## EXAMPLES

```
(turtle::name "Square")
(turtle::reset)
(turtle::background-color 'default)
(turtle::pen-color 'default )
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
```

# turtle::backward

## SYNOPSIS

(turtle::backward <distance>)

## DESCRIPTION

TURTLE::BACKWARD is a turtle.lisp shortcut for (TURTLE::FORWARD (- <distance>)).

## RETURN VALUE

NONE

## EXAMPLES

# turtle::forward

## SYNOPSIS

(turtle::forward <distance>)

## DESCRIPTION

TURTLE::FORWARD advances the turtle by the given distance. If the pen is down, then a line is drawn.

## RETURN VALUE

NONE

## EXAMPLES

```
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
```

# turtle::heading

## SYNOPSIS

(turtle::heading <angle>)

## DESCRIPTION

TURTLE::HEADING sets the turtle direction to a specific direction given by an angle expressed in degrees. The north is at position 0, west at 90, south at 180, and east at 270.

## RETURN VALUE

NONE

## EXAMPLES

```
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
```

# turtle::home

## SYNOPSIS

**(turtle::home)**

## DESCRIPTION

TURTLE::HOME is a turtle.lisp shortcut for (TURTLE::MOVE-TO 0 0).

## RETURN VALUE

NONE

## EXAMPLES

# turtle::left

## SYNOPSIS

(turtle::left <degrees>)

## DESCRIPTION

TURTLE::LEFT is a turtle.lisp shortcut for (TURTLE::TURN <degrees>).

## RETURN VALUE

NONE

## EXAMPLES

# turtle::move-to

## SYNOPSIS

**(turtle::move-to <x> <y>)**

## DESCRIPTION

TURTLE::MOVE-TO moves the turtle to a specific absolute position. No line is drawn.

## RETURN VALUE

NONE

## EXAMPLES

```
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
```

# turtle::name

## SYNOPSIS

(turtle::name <string>)

## DESCRIPTION

TURTLE::NAME names the current drawing canvas. Canvas with different names are preserved within the turtle window. If reusing an existing canvas, then the drawings are appended to it.

## RETURN VALUE

NONE

## EXAMPLES

```
(turtle::name "Square")
(turtle::reset)
(turtle::background-color 'default)
(turtle::pen-color 'default )
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
```

# turtle::pen-color

## SYNOPSIS

(turtle::pen-color <color>)

(turtle::pen-color)

## DESCRIPTION

TURTLE::PEN-COLOR sets the colors of the pen. It returns the previous color. When no color is given, the default system foreground color is used (black in light mode, white in dark mode). See the color section at the beginning of the chapter for interpreting the <color> parameter.

## RETURN VALUE

NUMBER

## EXAMPLES

```
(turtle::name "Square")
(turtle::reset)
(turtle::background-color 'default)
(turtle::pen-color 'default )
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
```

# turtle::pen-down

## SYNOPSIS

(turtle::pen-down)

## DESCRIPTION

TURTLE::PEN-DOWN moves the pen down, thus enabling drawing when the turtle is displaced onto the canvas.

## RETURN VALUE

NONE

## EXAMPLES

```
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
```

# turtle::pen-up

## SYNOPSIS

(turtle::pen-up)

## DESCRIPTION

TURTLE::PEN-UP moves the pen up, thus disabling any drawing when the turtle is displaced onto the canvas.

## RETURN VALUE

NONE

## EXAMPLES

```
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
```

# turtle::pen-width

## SYNOPSIS

(turtle::pen-width <size>)

## DESCRIPTION

TURTLE::PEN-WIDTH sets the size in pixels of the drawing pen. It returns the previous width.

## RETURN VALUE

NUMBER

## EXAMPLES

```
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
```

# turtle::push

## SYNOPSIS

(turtle::push)

## DESCRIPTION

TURTLE::PUSH saves the turtle state onto a stack and for later restoration using the TURTLE::POP function. Each turtle has its own stack and a state is made of the heading, position and pen up/down status of the turtle.

## RETURN VALUE

NONE

## EXAMPLES

# turtle::pop

## SYNOPSIS

(turtle::pop)

## DESCRIPTION

TURTLE::POP restores the turtle state that was last pushed using the TURTLE::PUSH function and removes this state from the stack of the states. If the stack is empty, then the turtle state is left unchanged. Each turtle has its own stack and a state is made of the heading, position and pen up/down status of the turtle.

## RETURN VALUE

NONE

## EXAMPLES

# turtle::reset

## SYNOPSIS

(turtle::reset)

## DESCRIPTION

TURTLE::RESET clears the current canvas and recenters the turtle, pen down. It is a good practice to call this function when starting a new drawing.

## RETURN VALUE

NONE

## EXAMPLES

# turtle::right

## SYNOPSIS

**(turtle::right <degrees>)**

## DESCRIPTION

TURTLE::RIGHT is a turtle.lisp shortcut for (TURTLE::TURN (- <degrees>)).

## RETURN VALUE

NONE

## EXAMPLES

# turtle::turn

## SYNOPSIS

(turtle::turn <angle>)

## DESCRIPTION

TURTLE::TURN turns to the left the turtle by an <angle> expresses in degrees; when the angle is negative, the turtle head is turned to the right. The TURTLE::LEFT and TURTLE::RIGHT functions are shortcuts to indicate the left or right direction.

## RETURN VALUE

NONE

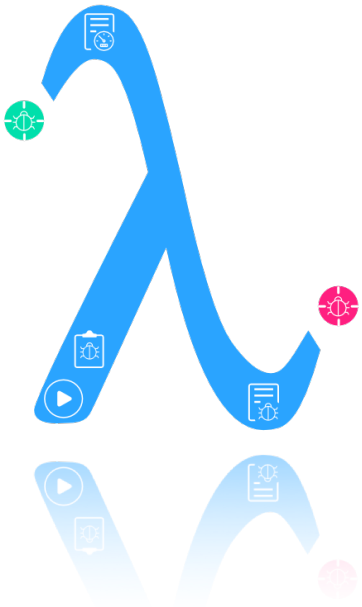
## EXAMPLES

```
(turtle::name 'square)
(turtle::reset)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
(turtle::turn 90)
(turtle::forward 100)
```

# Tracing & Debugging

Tracing is one of the most useful mechanism for debugging or understanding how Lisp programs and the interpreter work.

My Lisp ships with an integrated debugger and an advanced trace facility. The debugger allows suspend and resume evaluations along with the viewing and editing of the bindings. It is invoked as a standard My Lisp function *SYS::DEBUG*.



The function *(STS::TRACE-MODE <mode>)* allows controlling the level of tracing required, from no trace to all evaluation steps. The *SYS::DEBUG* function is not affected by the current level of tracing. The supported modes are:

**None**

All tracing functions are actually disabled and no no tracing is printed.

**Stats**

The statistics mode consists in printing general purpose performance counters of the expressions that have been evaluated. Once a full evaluation has completed, the elapsed time, the number of evaluations made and the deepest stack level that was reached are printed. These informations are merely informative but interesting to understand or feel the complexity of the operations involved. They have nearly no impact onto the interpreter and its execution behavior and speed.

**Print**

The print mode consists in printing trace messages inserted within the expressions by the programmer. When activated, all messages printed with the *println* function and starting with the *";**SYS.DBG:"* or *";**SYS::DBG:"* prefixes are printed as traces; when the mode is not activated, all these messages are ignored.

This mode allows keeping permanent programming traces within the programs as an alternative to other traces command. When the print mode is not enabled, the parameters of the *println* function are not evaluated once the *";**SYS.DBG:"* or *";**SYS::DBG:"* prefixes have been encountered, thus the performance impact is somehow minimal if the prefix is given as the first parameter; for instance *(println ";**SYS::DBG:" A B C)* will not evaluate *A*, *B*, and *C* when the trace mode is different from *'print*.

### Eval

The eval mode is the most detailed mode as it prints out almost all evaluation steps during the interpretation of an expression. These traces are merely informative (that is, they do not change the behavior of the interpreter) but quite fundamental to understand what's going on under the hood.

Warning: these traces are verbose and have a severe performance impact. For most and every complex scenarios, the trace mode will be more effective.

### Trace

The trace mode is a targeted eval mode with the eval function printed only for a set of functions previously registered with the SYS::TRACE function. For each of these functions, the parameters are printed upon invocation and the return value is also printed when the functions returned; each call is also indented and prefixed with the current stack level in order to understand when tail-recursion or recursion are in action.

The interpreter is automatically stopped and the debugger started when a traced function is entered or exited. It is possible to disable this behavior from within the debugger itself. This behavior allows entering the debugger without any need to add SYS::DEBUG function calls.

Note that when invoking the *SYS::TRACE* function, user functions must be quoted as in:

*(SYS::TRACE 'my-function)*

Starting with version 2.02, the debugger is automatically started when running in trace mode and a trace is printed, thus smoothly combining the debugger and the original trace engine.

# sys::bindings

## SYNOPSIS

**(sys::bindings)**

**(sys::bindings <number>)**

## DESCRIPTION

SYS::BINDINGS returns the environment bindings. When invoked without parameters it returns the current level ; when <number> is given then it returns the list of the bound symbols for the given level. The symbols are returned in a list that is sorted alphabetically according to the names of the symbols.

See the Startup paragraph for a description of the bindings contexts.

Note that starting with version 1.82, the original BINDINGS function is considered obsolete and should be avoided; this original function BINDINGS is now implemented within the Tools.lisp library file as a synonym to the SYS::BINDINGS function.

## RETURN VALUE

NUMBER or LIST

## EXAMPLES

**? (sys::bindings)**

**3**

**? (sys::bindings 'root)**

**(("nan" nan) ("nil" nil) ("pi" 3.1416) ("t" t) ("π" 3.1416))**

# sys::bindings-names

## SYNOPSIS

**(sys::bindings-names <number>)**

## DESCRIPTION

SYS::BINDINGS-NAMES returns the list of the bound symbols for the given level; when the level <number> is omitted then the current level is assumed. The symbols are returned in a list that is sorted alphabetically.

This function is a user function implemented within the tools.lisp library file. See the Startup paragraph for a description of the bindings contexts.

Note that starting with version 1.82, the original BINDINGS-NAMES function is considered obsolete and should be avoided.

## RETURN VALUE

LIST

## EXAMPLES

**? (sys::bindings-names 0)**

**(“nan” “nil” “pi” “t” “π”)**

# sys::bindings-assoc

## SYNOPSIS

**(sys::bindings-assoc <name> <number>)**

## DESCRIPTION

SYS::BINDINGS-ASSOC returns the value associated to the bound symbol for the given level.

This function is a user function implemented within the tools.lisp library file. See the Startup paragraph for a description of the bindings contexts.

Note that starting with version 1.82, the original BINDINGS-ASSOC function is considered obsolete and should be avoided.

## RETURN VALUE

SEXPR

## EXAMPLES

**? (sys::bindings-assoc "nan" 0)**

**nan**

**? (sys::bindings-assoc "pi" 0)**

**3.1416**

**? (sys::bindings)**

**3**

**? (sys::bindings-assoc "pi")**

**nil**

# sys::clear-bindings

## SYNOPSIS

**(sys::clear-bindings )**

## DESCRIPTION

SYS::CLEAR-BINDINGS clears all the bindings at the current level. This is not the same as a reset of the interpreter because you can not clear or reset to their original values the bindings at a level below the current one.

## RETURN VALUE

NIL

## EXAMPLES

**? (length (sys::bindings-names))**

**405**

**? (sys::clear-bindings)**

**nil**

**? (length (sys::bindings-names))**

**0**

# sys::debug

## SYNOPSIS

**(sys::debug)**

**(sys::debug (when <expr>) (name <expr>) )**

## DESCRIPTION

SYS::DEBUG starts the debugger and stops the execution of the interpreter until a user action is given. When the optional *(when <expr>)* part is indicated, the debugger is started only if *<expr>* evaluates to T. When the optional *(name <expr>)* part is indicated, the text associated to with the evaluation of *<expr>* is printed in the debugger toolbar.

Note that you can take advantage of PROGN and the (when…) or (name…) parts to print trace messages or create special bindings to give further context when the debugger is presented.

## RETURN VALUE

NONE

## EXAMPLES

```
(define (test a b)
        (sys::debug
                (when (not (number? a)))
                (name (->string "wrong value for a=" a)))
        (+ a b))
```

The debugger will start when the test function is invoked against a value of a that is not numeric. At this point, the user will be able to stop the current evaluation, edit some bindings, and resume the execution.

# sys::error?

## SYNOPSIS

**(sys::error? <expr>)**

## DESCRIPTION

SYS::ERROR? determines whether the evaluation of <expr> generated an interpreter error. It allows recovering fatal errors in programs running an interpreter within an interpreter.

## RETURN VALUE

T or NIL

## EXAMPLES

```
(define (test)
        (define check (some-undefined-function))
        (unless  (error? check)
                (something-else)))
```

# sys::print-values

## SYNOPSIS

**(sys::print-values <name>…)**

## DESCRIPTION

SYS::PRINT-VALUES prints the values associated to variables. It comes handy when debugging some code to understand the state of variables.

This function is a user function implemented within the tools.lisp library file.

## RETURN VALUE

NONE

## EXAMPLES

```
? (define x 5)
x

? (define y 8)
y

? (sys::print-values x y)
x=5, y=8
```

# sys::trace

## SYNOPSIS

**(sys::trace <name>…)**

## DESCRIPTION

SYS::TRACE starts tracing the functions described by the <name>… parameters. You can trace as many functions as required and the set of traced functions is kept across sessions. When invoked without parameters it returns the list of the traced functions.

Note that the functions are effectively traced if the current trace mode is TRACE, which is the default when the interpreter is started or following the execution of the instruction *(SYS::TRACE-MODE 'trace)*.

## RETURN VALUE

T or LIST

## EXAMPLES

```
? (sys::trace 'fib 'fib::helper)
t

? (fib 6)
   [001] fib --> n=6
   [001] fib::helper --> n=6 a=1 b=1
   [001] fib::helper --> n=5 a=1 b=2
   [001] fib::helper --> n=4 a=2 b=3
   [001] fib::helper --> n=3 a=3 b=5
   [001] fib::helper --> n=2 a=5 b=8
   [001] fib::helper  x 5 <-- 8
   [001] fib <-- 8
8

? (sys::trace)
(fib fib::helper)
```

# sys::trace-mode

## SYNOPSIS

**(sys::trace-mode)**

**(sys::trace-mode <mode>)**

## DESCRIPTION

SYS::TRACE-MODE gets or sets the current tracing mode. When supplied, the <mode> parameter may be a string or a symbol matching one of the following modes: none, stats, print, eval, trace.

See the initial page of the chapter and the SYS::TRACE function for details regarding tracing and debugging.

## RETURN VALUE

SYMBOL or NIL

## EXAMPLES

**? (sys::trace-mode 'stats)**

```
Total evaluations       : 2
Maximum stack level     : 2
Evaluation time         : 0.000s
```

**trace**

**? (sys::trace-mode)**

```
Total evaluations       : 1
Maximum stack level     : 1
Evaluation time         : 0.000s
```

**stats**

**? (sys::trace-mode 'trace)**

**stats**

# sys::untrace

## SYNOPSIS

(sys::untrace <name>…)

## DESCRIPTION

SYS::UNTRACE removes the tracing added to functions with calls to the SYS::TRACE function. When invoked without parameters, all functions are removed from the traces.

See the initial page of the chapter and the SYS::TRACE function for details regarding tracing and debugging.

## RETURN VALUE

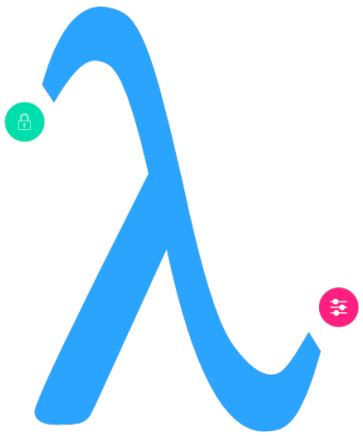
T or NIL

## EXAMPLES

```
? (sys::untrace 'fib::helper)
t

? (fib 6)
   [001] fib --> n=6
   [001] fib <-- 8
8
```

# Options

Some presentation options of the interpreter are accessible through predefined functions. Lisp settings are used as opposed to UI ones for the sake of simplicity, though the My Lisp application may expose them directly on a settings screen in the future.



# options::integer-mode

## SYNOPSIS

(options::integer-mode <mode>)

(options::integer-mode)

## DESCRIPTION

OPTIONS::INTEGER-MODE indicates whether and how big integers and rationals are supported. When no parameter is given, the current mode is returned. When a parameter is given, it changes the current integer mode accordingly and returns the previous value. The supported modes are:

- auto, the default, where the parser infers the integer and rational types from the input. This is the only mode where rationals can be input.
- none, where integer and rational numbers must be explicitly created from the functions like ->INTEGER.
- suffix, where integer numbers are recognized when the integer value is suffixed by the character indicated in the option OPTIONS::INTEGER-SUFFIX. In such a case, the suffix is used only for input, not output.

Note: you should reset the interpreter when changing the integer mode to ensure that all definitions are read as per the value of the option.

## RETURN VALUE

STRING

## EXAMPLES

? (options::integer-mode 'auto)

"auto"

? (integer? 1)

t

```
? (/ 1 3)
1/3

? (options::integer-mode 'none)
"auto"

? (integer? 1)
nil

? (/ 1 3)
0.3333

? (options::integer-mode 'suffix)
"none"

? (integer? 1)
nil

? (integer? 1#)
t

? (/ 1 3)
0.3333

? (/ 1# 3#)
1/3

? (options::integer-mode 'auto)
"suffix"

? (/ 1 3)
1/3
```

# options::integer-suffix

## SYNOPSIS

(options::integer-suffix <suffix>)

(options::integer-suffix)

## DESCRIPTION

OPTIONS::INTEGER-SUFFIX indicates the character to use as suffix to recognize integers when the OPTIONS::INTEGER-MODE is suffix or auto. When no parameter is given, the current suffix is returned. When a parameter is given, it changes the current integer suffix accordingly and returns the previous value. The default value is #.

Note: you should reset the interpreter when changing the integer mode to ensure that all definitions are read as per the value of the option.

## RETURN VALUE

STRING

## EXAMPLES

? (options::integer-mode 'suffix)

"auto"

? (integer? 1)

nil

? (/ 1 3)

1/3

? (options::integer-suffix)

"#"

? (integer? 1#)

```
t

? (/ 1# 3#)
1/3

? (options::integer-suffix 'N)
"#"

? (integer? 1N)
t

? (integer? 1#)
nil

? (/ 1N 3N)
1/3

? (options::integer-suffix '#)
"N"

? (options::integer-mode 'auto)
"suffix"
```

# options::number-decimals

## SYNOPSIS

(options::number-decimals <number>)

(options::number-decimals)

## DESCRIPTION

OPTIONS::NUMBER-DECIMALS indicates the number of decimals to use when formatting floating point values. The default is 4. When a parameter is given it changes the current number of decimals accordingly and returns the previous value.

Note that the number of decimals has no consequence on the internal precision of the values and operations.

The value of this option is common to all interpreter instances and saved/restored across sessions.

Obsolete synonym: starting with version 1.85, the original NUMBER->STRING-DECIMALS function is obsolete and must be avoided; this original function is now implemented within the Tools.lisp library file as a synonym to the OPTIONS::NUMBER-DECIMALS function.

## RETURN VALUE

NUMBER

## EXAMPLES

```
? (number->string-decimals)
4

? (/ 1.0 3)
0.3333

? (number->string-decimals 6)
4
```

```
? (number->string-decimals)
6

? (/ 1.0 3)
0.333333
```

# options::number-format

## SYNOPSIS

(options::number-format <format>)

(option::number-format)

## DESCRIPTION

OPTIONS::NUMBER-FORMAT indicates the output format to use when formatting floating point values. The supported formats are FIX (fixed point), ENG (engineering), and SCI (scientific). The default is FIX. When a parameter is given it changes the current format accordingly and returns the previous value.

Note that the output format has no consequence on the internal precision of the values and operations.

The value of this option is common to all interpreter instances and saved/restored across sessions.

Obsolete synonym: starting with version 1.85, the original NUMBER->STRING-FORMAT function is obsolete and must be avoided; this original function is now implemented within the Tools.lisp library file as a synonym to the OPTIONS::NUMBER-FORMAT function.

## RETURN VALUE

STRING

## EXAMPLES

? (number->string-format)

FIX

? (* 1.23456789 10000)

12345.6789

? (number->string-format 'SCI)

FIX

```
? (number->string-format)
SCI

? (* 1.23456789 10000)
1.2346E+004

? (number->string-format 'ENG)
SCI

? (number->string-format)
ENG

? (* 1.23456789 10000)
12.346E+003
```

# options::quote-as-quote

## SYNOPSIS

(options::quote-as-quote)

(options::quote-as-quote <sexpr>)

## DESCRIPTION

OPTIONS::QUOTE-AS-QUOTE indicates whether the QUOTE function should be printed in plain or using the standard abbreviated apostrophe character. When <expr> is not NIL, the QUOTE function is printed in plain. Note that when the apostrophe is used, the QUOTE function is still printed in plain when the expression containing the QUOTE function is not a proper one. When the <sexpr> argument is missing then the current value is returned.

The value of this option is common to all interpreter instances and saved/restored across sessions.

Prior to version 1.85, OPTIONS::QUOTE-AS-QUOTE was always T.

## RETURN VALUE

T or NIL

## EXAMPLES

? (define (kwote S) (list 'quote S))

kwote

? (options::quote-as-quote 't)

nil

? (options::quote-as-quote)

t

? kwote

(lambda (S) (list (quote quote) S))

```
? (options::quote-as-quote '())
t

? (options::quote-as-quote)
nil

? kwote
(lambda (S) (list 'quote S))
```

# options::keyboard-mode

## SYNOPSIS

**(options::keyboard-mode)**

**(options::keyboard-mode <mode>)**

## DESCRIPTION

OPTIONS::KEYBOARD-MODE indicates the current keyboard scheme. When no parameter is given, the current mode is returned. When a parameter is given, it changes the current mode accordingly and returns the previous value. The supported modes are:

- default (0), the default, where keyboard shortcuts are used whenever possible on the software keyboard.
- no-shortcut (1), where no shortcut is displayed on the software keyboard and the keyboard toolbar always presented.

Note 1: changing the keyboard mode requires a screen refresh; you need to change the orientation of the iPad or change the view to the editor or help one.

Note 2: the keyboard mode has effect only on the iPad.

## RETURN VALUE

STRING

## EXAMPLES

**? (options::keyboard-mode)**

**default**

**? (options::keyboard-mode 'no-shortcut)**

**default**

**? (options::keyboard-mode 'default)**

**no-shortcut**

# Last words...

My Lisp is proudly developed by Laurent Rodier, a freelance software developer.  You can get more information at:

https://www.lsrodier.net

https://lisp.lsrodier.net

May the Lisp be with you…