Common Lisp Quick Reference

Compiled by W. Burger, WS 95/96

1 Symbols

`nil`
: Constant, whose value is itself (`NIL`).

`t`
: Constant, whose value is itself (`T`).

`(symbolp` *e*`)`
: Returns `T` if *e* evaluates to a symbol, otherwise `NIL`.

`(boundp` *e*`)`
: Returns `T` if the value of *e* (which must be a symbol) has neither a global nor a local value.

`(defvar` *sym e*`)`
: Defines *sym* to be a global variable with initial value *e*.

`(defparameter` *sym e*`)`
: Defines *sym* to be a global parameter whose value (initial *e*) will may change at runtime but is not expected to.

`(defconstant` *sym e*`)`
: Defines *sym* to be a global constant whose value (*e*) will not change while the program is running.

2 Value Assignment

`(setf` *place e*`)`
: Stores the value of *e* in the place specified by *place*.

`(setq` *sym e*`)`
: Evaluates *e* and makes it the value of the symbol *sym*; the value of *e* is returned.

3 Input/Output

`(read [`*stream*`])`
: Reads the printed representation of a single object from *stream* (which defaults to `*standard-input*`), builds a corresponding object, and returns the object.

`(read-line [`*stream eof-error-p eof-value recursive-p*`])`
: Reads a line of text terminated by a `newline` or `end-of-file` character from *stream* (which defaults to `*standard-input*`) and returns two values: the line as a character string and a Boolean value, `T` if the line was terminated by an `end-of-file` and `NIL` if it was terminated by a `newline`.

`(read-char [`*stream eof-error-p eof-value recursive-p*`])`
: Reads one character from *stream* (which defaults to `*standard-input*`), and returns the corresponding character object.

`(read-char-no-hang [`*stream eof-error-p eof-value recursive-p*`])`
: Reads and returns a character from *stream* (which defaults to `*standard-input*`) if one is immediately available, otherwise immediately returns `NIL`.

`(read-byte` *binary-stream* `[`*eof-error-p eof-value recursive-p*`])`
: Reads one byte from the binary input stream *binary-stream* returns it in the form of an integer.

`(read-from-string` *string* `[`*eof-error-p eof-value recursive-p* `:start` *start* `:end` *end*
`:preserve-whitespace` *preserve-whitespace*`])`
: Reads and returns an expression, taking input from *string*. A second value returned indicates the index of the first character in string not read.

`(prin1` *e* `[`*stream*`])`
: Prints a readable representation of *e* to *stream* (which defaults to `*standard-output*`).

`(print` *e* `[`*stream*`])`
: Prints a readable representation of *e* to *stream* (which defaults to `*standard-output*`), at the beginning of a new line.

`(format` *stream control-string* `[`e_1 ... e_k`])`
: Writes formatted output to *stream* using *control-string* and arguments $e_1 \ldots e_k$. If the value of Argstream is `T` the output goes to `*standard-output*` and `format` returns `NIL`. If the value of Argstream is `T` the output goes to a new string which is returned.

`(fresh-line [`*stream*`])`
: Outputs a `newline` to *stream* (which defaults to `*standard-output*`) unless the stream is not already at the start of a line.

`(write-char` *char* `[`*stream*`])`
: Outputs *char* onto *stream* and returns *char*.

`(write-byte` *integer binary-stream*`)`
: Writes one byte, the value of *integer*, onto *binary-stream*.

`(with-open-file (`*stream-var filename options*`)` e_1 ... e_k`)`
: Sets up an input (default) or output stream to the named file. It then evaluates $e_1, \ldots, e_k$ with *stream-var* bound to the open stream and finally closes the file.

`(load` *filename*`)`
: Loads the (Lisp) file with the name given by the string *filename* into the Lisp environment.

4 Lists

`(list [`e_1 ... e_k`])`
: Creates a list containing elements e_1, ..., e_k.

`(cons` *e lst*`)`
: Creates a list with *e* as the first element and *lst* as the rest.

`(make-list` *size*`)`
: Creates and returns a list containing *size* elements each of which is initialized to `NIL` (other options available).

`(length` *lst*`)`
: Returns the number of (top-level) elements of the list *lst*. May fail on circular lists.

`(list-length` *lst*`)`
: Returns the number of (top-level) elements of the list *lst*. Returns `NIL` on circular lists.

`(listp` *lst*`)`
: Returns `T` if *lst* is a list (even when *lst* is empty).

`(first` *lst*`)`
: Returns the first element of the list *lst* (`NIL` if *lst* is empty); equivalent to `car`.

`(second` *lst*`)`
: Returns the second element of the list *lst*.

`(rest` *lst*`)`
: Returns a list containing all exept the first element of the list *lst* (`NIL` if the length of *lst* is less than 2); equivalent to `cdr`.

`(nth` *n lst*`)`
: Returns the element at the *n*-th position of the list *lst* (the first list element has positon 0). `NIL` is returned if *n* is outside the list.

`(last` *lst* `[`*n*`])`
: Returns a list that contains the last*n* (default is 1) elements of *lst* (`NIL` if *lst* is empty).

`(butlast` *lst*`)`
: Returns a copy of the list *lst* with the last element removed.

`(append` lst_1 lst_2 ... lst_k`)`
: Returns a list that is the concatenation of lists lst_1, lst_2, ..., lst_k.

`(member` *item lst*`)`
: Returns true (actually the tail of *lst* beginning with the first matching element) if *item* occurs anywhere at the top level in *lst*.

`(remove` *item lst*`)`
: Returns a list similar to *lst* with all elements `eql` to *item* removed. `remove` is non-destructive; additional options are available.

`(copy-list` *lst*`)`
: Returns a top-level copy of *lst*.

(subseq *lst start* [*end*])
: Creates a new list using values from *lst*. If *start* evaluates to 0 and *end* is not given, then the whole list is copied.

(reverse *lst*)
: Returns a new list with the same values as those in the value of *lst* but in reverse order.

(push *item place*)
: Inserts *item* at the beginning of the list stored at *place* and stores the resulting list back at *place*.

(pop *place*)
: Returns the `first` of the list stored at *place* and stores the `rest` back in *place*.

5 Predicates

(atom *e*)
: Returns `T` if *e* evaluates to an atom and `NIL` otherwise. Thus `atom` is `T` if the value of *e* is not a cons.

(null *e*)
: Returns `T` if *e* evaluates to `NIL`; otherwise returns `NIL`.

(consp *lst*)
: Returns `T` if *lst* is a true list (i.e., a non-empty list, called a "cons").

(endp *lst*)
: Returns `T` if the value of *lst* is `NIL`, and `NIL` if it is some non-empty list.

(typep *object type*)
: Returns `T` if the value of *object* is of type *type*, `NIL` otherwise.

6 Testing for Equality

(eq e_1 e_2)
: Returns `T` if and only if e_1 and e_2 are the same object (same address). Efficient test for the equality of two symbols, but not useful for testing the equality of lists, numbers, or other Lisp objects. Things that print the same are not necessarily `eq`, numbers with the same value need not be `eq`, and two similar lists are usually not `eq`.

(eql e_1 e_2)
: Returns `T` if e_1 and e_2 are `eq`, or if they are numbers of the same type with the same value, or if they are character objects that represent the same character.

(equal e_1 e_2)
: Returns `T` when e_1 and e_2 are structurally similar. A rough rule of thumb is that objects are `equal` when their printed representation is the same. `equal` is case sensitive when comparing strings and characters.

(equalp e_1 e_2)
: Returns `T` if e_1 and e_2 are `equal`; if they are characters and satisfy `char-equal`; if they are numbers with the same numeric value (even if they are of different types); or if they have components that are all `equalp`. Special rules apply to arrays, hash tables, and structures (see the full Common Lisp specification).

7 Arithmetic

(+ n_1 n_2 ... n_k)
: Returns $n_1 + n_2 \ldots + n_k$.

(- n_1 n_2 ... n_k)
: Returns $n_1 - n_2 \ldots - n_k$.

(* n_1 n_2 ... n_k)
: Returns $n_1 * n_2 \ldots * n_k$.

(/ n_1 n_2 ... n_k)
: Returns $(\ldots(n_1/n_2)\ldots/n_k)$.

(incf *place* [*delta*]
: Adds *delta* (which defaults to 1) to the value stored at *place* and stores the new value in the same location.

`(decf` *place* `[`*delta*`])`
: Subtracts *delta* (which defaults to 1) from the value stored at *place* and stores the new value in the same location.

`(max` n_1 `[`n_2 ... n_k`])`
: Returns the maximum value of its arguments.

`(min` n_1 `[`n_2 ... n_k`])`
: Returns the minimum value of its arguments.

`(mod` *number divisor*`)`
: Returns *number* modulo *divisor*.

`(rem` *number divisor*`)`
: Returns the remainder of *number* divided by *divisor*. The result has the same sign as *divisor*.

`(signum` n`)`
: Returns 1 if the value of n is positive, 0 if the value is zero, and -1 if the value is negative. The result has the same type as n.

`(abs` n`)`
: Returns the absolute value of n.

`(ceiling` n`)`
: Returns the smallest integer that is not smaller than the value of n.

`(floor` n`)`
: Returns the largest integer that is not larger than the value of n.

`(sqrt` n`)`
: Returns the principal square root of the value of n.

`(exp` n`)`
: Returns e^n.

`(expt` *base* n`)`
: Returns $base^n$.

`(log` n `[`*base*`])`
: Returns the logarithm base *base* (which defaults to e) of n.

`(cos` n`)`
: Returns the cosine of n (given in radians).

`(sin` n`)`
: Returns the sine of n (given in radians).

`(tan` n`)`
: Returns the tangent of n (given in radians).

`(acos` n`)`
: Returns the arc cosine (in radians) of n.

`(asin` n`)`
: Returns the arc sine (in radians) of n.

`(atan` n`)`
: Returns the arc tangent of (in radians) n.

`(float` n`)`
: Returns the value of n as a floating point number.

`(truncate` n`)`
: Returns the integer having the largest absolute value that is not larger than the absolute value of n. Thus `truncate` truncates toward zero.

`(round` n`)`
: Returns the integer closest to the value of n.

`(random` n`)`
: Returns a random number between 0 (inclusive) and the value of n (exclusive). The type returned is the same as that of n (integer or floating point).

`pi`
: A constant whose value is the floating-point approximation of π.

8 Comparisons and Predicates on Numbers

`(=` n_1 ... n_k`)`
: Returns `T` if all the arguments are numerically equal; otherwise returns `NIL`.

`(/=` n_1 ... n_k`)`
: Returns `T` if the arguments are all different; otherwise returns `NIL`.

`(< `n_1 ... n_k`)`
Returns true if each argument is less than the one following it; otherwise returns nil.

`(<= `n_1 ... n_k`)`
Returns true if each argument is less or equal than the one following it; otherwise returns nil.

`(> `n_1 ... n_k`)`
Returns `T` if each argument is greater than the one following it; otherwise returns `NIL`.

`(>= `n_1 ... n_k`)`
Returns `T` if each argument is greater or equal than the one following it; otherwise returns `NIL`.

`(zerop `n`)`
Returns `T` if the number n is zero (either the integer zero, a floating-point zero, or a complex zero); otherwise returns `NIL`. `(zerop -0.0)` is always true.

`(minusp `n`)`
Returns `T` if the number n is strictly greater than zero; otherwise returns `NIL`.

`(minusp `n`)`
Returns `T` if the number n is strictly less than zero; otherwise returns `NIL`.

`(numberp `e`)`
Returns `T` if e evaluates to any Common Lisp number.

9 Logical Functions

`(and [`e_1 ... e_k`])`
Evaluates each argument e_i sequentially. If `and` reaches an argument that returns `NIL`, it returns `NIL` without evaluating any more arguments. If it reaches the *last* argument, it returns that argument's value.

`(or [`e_1 ... e_k`])`
Evaluates each argument e_i sequentially. If `or` reaches an argument that is not `NIL`, it returns the value of that argument without evaluating any more arguments. If it reaches the *last* argument, it returns that argument's value.

`(not `e`)`
Returns `T` if the value of e is `NIL`; otherwise returns `NIL`.

10 Block Constructs

`(progn [`e_1 ... e_n`])`
Evaluates e_1 through e_n and returns the value of e_n.

`(prog1 `e_1 `[`e_2 ... e_n`)`
Evaluates e_1 through e_n and returns the value of e_1.

`(let (`lvb_1 ... lvb_k`) `e_1 ... e_n`)`
Sets up local variable bindings lvb_1, ..., lvb_k, evaluates e_1 through e_n, and returns the value of e_n (implicit `progn`).

`(let* (`lvb_1 ... lvb_k`) `e_1 ... e_n`)`
Sets up local variable bindings lvb_1, ..., lvb_k sequentially. Otherwise same as `let`.

11 Conditional Constructs

`(if `*test* e_1 e_2`)`
Evaluates *test* and, if not `NIL`, evaluates e_1. Otherwise, e_2 is evaluated. e_2 can be omitted.

`(when `*test* e_1 ... e_n`)`
Evaluates *test* and, if not `NIL`, evaluates e_1 through e_n. The value of e_n is returned (implicit `progn`).

`(unless `*test* e_1 ... e_n`)`
Evaluates *test* and, if `NIL`, evaluates e_1 through e_n. The value of e_n is returned (implicit `progn`).

`(cond (`$test_1$ e_{11} ... e_{1n}`) ... (`$test_k$ e_{k1} ... e_{kn}`))`
Evaluates $test_1$, ..., $test_k$ until $test_i$ evaluates to something non-`NIL`, then evaluates e_{i1}, ..., e_{in}) as an implicit `progn`.

`(case `*keyform* `(`key_1 e_{11} ... e_{1n}`) ... (`key_k e_{k1} ... e_{kn}`))`
Evaluates *keyform*, then evaluates as an implicit `progn` the forms e_{ij} whose keys key_i match the

value of *keyform*. Returns the last form evaluated. *keyform* is evaluated, but the keys are not. `case` permits a final case, `otherwise` or `t`, that handles all keys not otherwise covered.

12 Iteration Constructs

`(dolist (`*var lst* `[`*result*`]) `e_1 ... e_k`)`
Evaluates $e_1 \ldots e_k$ for each element in the list *lst*. The variable *var* is bound to the current list element value. Finally, if specified, *result* is evaluated and returned.

`(dotimes (`*var count* `[`*result*`]) `e_1 ... e_k`)`
Evaluates $e_1 \ldots e_k$ *count*-times. The variable *var* is bound to the current iteration value, starting with 0 up to $count - 1$. Finally, if specified, *result* is evaluated and returned.

`(mapcar `*fun* $arglst_1$ `[`$arglst_2$ ... $arglst_k$`])`
Applies *fun* to the successive elements of the argument lists $arglst_i$ and returns a list of the results. If there are k argument lists, then *fun* should be a function of k arguments.

13 Function Definition

`(defun `*name* `(`a_1 ... a_k`) `e_1 ... e_n`)`
Defines a named function with arguments $a_1 \ldots a_k$ and body $e_1 \ldots e_n$. *Note:* Additional options are available for specifying the arguments!

`(lambda (`a_1 ... a_k`) `e_1 ... e_n`)`
Defines a local function with arguments $a_1 \ldots a_k$ and body $e_1 \ldots e_n$. *Note:* Additional options are available for specifying the arguments!

`(function `*fn*`)`
Returns the functional object associated with *fn*. If *fn* is a symbol, its functional definition is returned. If *fn* is a lambda-expression, then `function` returns a "lexical closure".

14 Evaluation-Related

`(apply `*function* `[`a_1 ... a_k`] `*arglist*`)`
Applies the value of *function* to the list of all arguments – either *arglist* if no a_i are given, or the list of $a_1 \ldots a_k$ with *arglist* appended to it.

`(eval `e`)`
First evaluates e and then evaluates the resulting value again.

`(funcall `*function* a_1 ... a_k`)`
Applies the value of *function* (a function object) to the arguments $a_1 \ldots a_k$.

`(quote `e`)`
`quote` simply returns its argument without evaluating it. This allows any Lisp object to be written as a constant value in a program. `(quote `x`)` can be abbreviated as `'`x.

15 Miscellaneous

`(time `e`)`
Evaluates e and also outputs system-dependent timing information.

`(trace `$function_1$ ... $function_k$`)`
Enables runtime tracing of the named functions ($function_i$ need not to be quoted!).

`(untrace `$function_1$ ... $function_k$`)`
Disables runtime tracing of the named functions ($function_i$ need not to be quoted!). If no functions are given, tracing is turned off for all functions.

`(lisp-implementation-version)`
Returns a string that identifies the version of the particular Common Lisp implementation.

`(sleep `n`)`
Causes execution to be suspended for approximately n seconds. Returns `NIL`.