Common Lisp Reference Card

Conventions

one or more	`+`
zero or more	`*`
optional	`[]`
optional with default	`[arg=def]`

Evaluation Control

prevent evaluation	`quote`
evaluate form	`eval`
apply function to object and list	`apply`
(apply *function [objects] list*) ⇒ result	
apply function to objects	`funcall`
fully expands a form	`macroexpand`
return values computed by forms	`values`
	`multiple-value-bind`

Flow of Control

list delimeters	`(  )`
comment delimeters	`;`
return value from function or loop	`(return expr)`
	`return-from`
	`prog`
	`prog1`
	`prog2`
	`progn`
	`block`

Conditional Flow Control (`if/when/unless/cond/case`)

```
(if test then-form [else-form])
(when test then-form+)
(unless test else-form+)
```

```
(cond (test1 consequent1+)
      (test2 consequent2+)
      ...
      (testn consequentn+))
```

```
(case key-form
  (key1 consequent1+)
  (key2 consequent2+)
  ...
  (keyn consequentn+))
```

Iteration Flow Control (`dotimes/dolist/do/loop`)

```
(dotimes (count upper-bound result)
         body*)
```

```
(dolist (element list result)
        body*)
```

```
(do ((parameter1 initial-value1 update-form1)
     (parameter2 initial-value2 update-form2)
     ...
     (parametern initial-valuen update-formn))
  (termination-test intermediate-forms* result)
  body*)
```

```
(loop body+) ;; (return [value]) to quit loop
```

Function Definition

define a function:

(defun *fname (args+) (body)+*) ⇒ func

define a macro	`defmacro`
	`defsetf`
	`lambda`
	`function`

Numeric Functions

sum, difference, product, quotient	`+, -, *, /`
Example. `(- (* 2 4 3) 5 2)` ⇒ `17`	
increment, decrement	`1+, 1-`
increment, decrement place:	
(incf/decf *place [delta]*) ⇒ new-value	
$\sqrt{x}$	`sqrt x`
x^y	`expt x y`
min, max	`min, max`
Example. `(min -1 2 -3 4 -5 6)` ⇒ `-5`	
absolute value	`abs`
numerator/denominator of rational	`numerator, denominator`
modulus, remainder	`mod, rem`
nearest integer	`round, fround`
if between, rounds towards even integer	
truncate towards 0	`truncate, ftruncate`
truncate towards $+\infty$	`ceiling, fceiling`
truncate towards $-\infty$	`floor, ffloor`
greatest common denominator	`gcd`
least common multiple	`lcm`
sign (-1, 0, or 1)	`signum`
(log *number [base=e]*) ⇒ logarithm	

Trig Functions `arguments in radians` *not* `degrees`

sin, cos, tan	`sin, cos, tan`
arc sin, cos, tan	`asin, acos, atan`
hyperbolic	`sinh, cosh, tanh, asinh, acosh, atanh`
3.141592653589793. . .	`pi`

Predicates

General Predicates

identical objects/symbols	`eq`
eq and same char or number	`eql`
eql does *not* work on lists or strings	
structurally similar (isomorphic) objects	`equal`
same components	`equalp`
nil	`null`
object is a cons	`consp`
(consp nil) is nil	
end of list	`endp`
	`member`

Numeric Predicates

equal, $\neq$, magnitude/order	`=, /=, <, >, <=, >=`
Example. `(> 7 5 1)` ⇒ `t`	
parity	`evenp, oddp`
negative, zero, positive	`minusp, zerop, plusp`

String Predicates

string/char	`string=, /=, <, >, <=, >=`
`string-equal, -not-equal`	
`string-lessp, -greaterp`	
`string-not-greaterp, -not-lessp`	

Character Predicates

Use **char** everywhere **string** is used above.

Logical Predicates

boolean logic	`and, or, not`

Type-checking Predicates

a list	`listp`
a number	`numberp`
an integer	`integerp`
a float	`floatp`
a string	`stringp`
an atom	`atom`
a symbol	`symbolp`
a hash table	`hash-table-p`
of certain type	`typep`

Selection Functions

(every *predicate sequences+*) ⇒ bool

(some *predicate sequences+*) ⇒ bool

(notevery *predicate sequences+*) ⇒ bool

(notany *predicate sequences+*) ⇒ bool

Assignment

	`setq`
sequential set place to value	`setf`
parallel set places to values	`psetf`
parallel	`let`
sequential	`let*`
	`set`

Arrays

create new array:

(make-array *'(dimension+)*) ⇒ new-array

retrieve array element:

(aref *array-name dimension+*) ⇒ place

set an array element:

`(setf (aref a1 2 5) 7 )` ⇒ 7

create vector from list of values:

`(vector 34 22 30 )` ⇒ #(32 22 30)

Hash Tables

create new hash table:

(make-hash-table *:test #'equal*) ⇒ new-ht

retrieve a hash value:

(gethash *'key ht-name*) ⇒ value

set a hash value:

`(setf (gethash 'key htbl) 'val )` ⇒ val

Structures/Objects

define a structure/object type:

(defstruct *str-t slot1 slot2*) ⇒ str-t

create an object of type str-t:

(make-str-t *:slot1 val . . .*) ⇒ new-obj

retrieve slot1 of obj:

(str-t-slot1 *obj*) ⇒ val

set slot1 of obj to val:

`(setf (str-t-slot1 obj) 'val )` ⇒ val

obtain type of object	`type-of`

Symbols

retrieve prop1 of symb1:
`(get 'symb1 'prop1 )` ⇒ value
set prop1 of symb1 to val:
`(setf (get 'symb1 'prop1) val )` ⇒ val

Format Codes

new line	`~%`
n new lines	`~n%`
fresh line	`~&`
page break	`~|`
tab stop	`~T`
n tabs	`~nT`
justification	`~< ~>`
right justification in field w wide	`~w< ~>`
character	`~C`
spell out non-printable char	`~:C`
convert to lower case	`~( ~)`
convert to upper case	`~:@( ~)`
capitalize each word	`~:( ~)`
capitalize first word	`~@( ~)`
decimal integer	`~D`
right justified dec int	`~wD`
decimal integer with +	`~@D`
decimal integer with ,	`~:D`
binary integer	`~B`
octal integer	`~O`
hexadecimal integer	`~X`
base-b integer	`~bR`
spell an integer cardinal (one)	`~R`
spell an integer ordinal (first)	`~:R`
print with Roman numerals	`~@R`
plural with 's'	`~P`
plural with 'y/ies'	`~@P`
reuse previous arg for plural	`~:P ~:@P`
floating point	`~F`
scientific notation	`~E`
the shorter of ~F or ~E	`~G`
monetary	`~$`
strings, etc..., legibly	`~A`
READably, with escapes	`~S`
~	`~~`

Sequence Operations

concatenates sequences as *result-type*	`concatenate` *rt seq+*
indexes into sequence (0-based)	`elt` *seq idx*
	`subseq`
reverses elements in sequence	`reverse`
number of elements in sequence	`length`
number of items occurences in sequence	`count` *it seq*
removes occurences of *it* in *seq*	`remove` *it seq*
substitute *new* for *old*	`[n]substitute` *new old seq*
delete *it* from *seq*	`delete` *it seq*
	`remove-duplicates`
	`delete-duplicates`
replace all *seq* items with *it*	`fill` *seq it*
replace items in s1 with s2 keywords: start1, end1, start2, end2	`replace` *s1 s2*
	`position`
first occurence of item in seq	`find` *it seq*
	`search`
	`mismatch`
destructively sort based on predicate	`sort` *seq pred*
	`merge`

List Functions

	`car, first`
	`cdr, rest`
	`cadr`
	`first, second, ... tenth`
	`last`
	`nth`
	`nthcdr`
	`cons`
	`append`
	`list`
destructive append	`nconc`
	`assoc`
	`subst`

Maping

`mapcar`
`mapc`
`mapcan`
`maplist`
`mapl`
`mapcon`
`map`
`map-into`
`reduce`

Streams

`open`
`close`
`with-open-file`

Documentation

`apropos`
`describe`
`inspect`
`documentation`

I/O Functions

	`read`
	`load`
LISP-readable output	`print`
output w/o delimiters	`princ`
	`pprint`
	`terpri`
	`format`

Compiler Functions

`compile`
`compile-file`
`defvar`
`eval-when`

Packages

`intern`
`unintern`
`export`
`import`
`provide`
`require`
`in-package`
`use-package`

Set Functions

`adjoin`
`[n]intersection`
`[n]set-exclusive-or`
`[n]set-difference`
`[n]union`
`subsetp`

Send comments or corrections to Gabriel B. Burca, ⟨gburca@yahoo.com⟩