Common Lisp

Blake McBride (blake@mcbridemail.com)

Contents

1 Data Types

Type	Comment	Example
null	null list & false	nil
boolean	true	t
integer	decimal	48
	binary	#b1011
	binary	#2r1011
	octal	#o755
	hex	#xD5
ratio		2/3
	hex	#xDE/FF
float	default precision	3.141
	short precision	3.141s0
	single precision	3.141f0
	double precision	3.141d0
	long precision	3.141L0
character	letter 'M'	#\M
		#\Space
		#\Newline
	semi-standard	#\Backspace
		#\Tab
		#\Linefeed
		#\Return
		#\Page
symbol	case insensitive (upcased)	'symb
	case sensitive	'|symb|
	embedded '('	'sy\(mb
	in keyword package	:symb
	external symbol in ppp package	ppp:symb
	internal symbol in ppp package	ppp::symb
string	with embedded "	"hello\" there"
cons	list	'(hello there)

Data Types (continued)

Type	Comment	Example
vector		#(3 4 hello) (vector 3 4 'hello)
bit vector		#*1011
array	2 dimensional array	#2A((1 2 3) (4 5 6))

(type–of obj) returns the type of the object passed

2 Numeric Hierarchy

t
- number
  - real
    - rational
      - integer
        - fixnum (most–positive–fixnum)
        - bignum
      - ratio
    - float
      - short–float
      - single–float (4 bytes?)
      - double–float (8 bytes?)
      - long–float

3 Comments

Semi-colon ';' introduces a comment. Comments extend from the semi-colon to the end of the line.

4 List Operations

Type	Comment	Sp	D	Example	Result
car	first element	fast	no	(car '(A B C))	A
				(car nil)	nil
cdr	rest of list	fast	no	(cdr '(A B C))	(B C)
				(cdr nil)	nil
c????r	shorthand for multiple car & cdr's	fast	no		
cons	add a list node	fast	no	(cons 'A '(B C))	(A B C)
list	create a list	fast	no	(list 'A 'B 'C)	(A B C)
append	append lists	slow	no	(append '(A B) '(D) '(G H))	(A B D G H)
length		slow	no	(length '(A B C))	3
nth		slow	no	(nth 1 '(A B C))	B
				(nth 8 '(A B C))	nil
rest	same as cdr				
first	same as (nth 0 x)				
second	same as (nth 1 x)				
third	same as (nth 2 x)				
etc.					
last		slow	no	(last '(A B C))	(C)
	Given: (setq lst '(A B C))				
rplaca	replace car cell	fast	yes	(replaca lst 'X)	(X B C)
rplacd	replace cdr cell	fast	yes	(replacd lst 'X)	(A . X)

Sp = Speed

D = Destructive

5 Evaluation and Quotes

Lists are recursively evaluated. The first element of a list is the function and the remainder are arguments to the function. For example:

```
(func arg1 arg2 ...)
```

Quoting stops evaluation, so

```
(quote (abc def ghi))
```

would not attempt to evaluate *abc*, *def*, or *ghi*. The "`'`" symbol acts as a shorthand for *quote*, so the following is equivelent to the previous statement.

```
'(abc def ghi)
```

6 String Operations

"This is\" a string"
(string "new string")
(length str)
(concatenate 'string str1 str2)
(schar str idx) or (char str idx) or (aref str idx)
(setf (aref str idx) #\X)
(subseq str start [end])
(search "abc" str)
(parse–integer str [:start x] [:end y])
(string #\x)

7 Predicates

function	Returns *NIL*, or *T* for
null	null
symbolp	symbols
atom	non-list items
consp	list items (excluding *NIL*)
listp	list items (including *NIL*)
numberp	all numbers
integerp	integers
rationalp	rational numbers
floatp	floating point numbers
characterp	characters
stringp	strings
bit-vector-p	bit vectors
eq	same object
eql	same object, character, or numbers of the same type
equal	structurally similar objects, string compare
equalp	case insensitive string compare, elements of an array
string=	string equal
string/=	string not equal
string>	string greater than
string<	string less than

8 Math Predicates

These also work if the numbers aren't the same type.

Function	Description
(numberp n)	is *n* a number
(zerop n)	
(plusp n)	
(minusp n)	
(oddp n)	
(evenp n)	
(integerp n)	
(= a b)	
(/= a b)	
(< a b)	
(> a b)	
(<= a b)	
(>= a b)	

9 Logical Operators

function	Example	Comment
not	(not exp)	Logical inverse
and	(and exp1 exp2 ...)	Stops evalution on first null expression
or	(or exp1 exp2 ...)	Stops evalution on first non-null expression

and and *or* can be used as conditionals too. For example:

```
(and exp1 exp2 exp3)
```

is the same as

```
(if exp1
    (if exp2 exp3))
```

10 Program Control

(progn
exp1
exp2
...)

PROGN: Execute each expression and return the value of the last expression.

(prog1
exp1
exp2
...)

PROG1: Execute each expression and return the value of the first expression.

(prog2
exp1
exp2
...)

PROG2: Execute each expression and return the value of the second expression.

(prog ((var1 val1) ; initialize *var1* to *val1*
var2 ; initialize *var2* to *nil*
...)
exp1
lbl
exp2
(go lbl) ; branch to label *lbl*
(return val) ; exit *prog*, returns *val*
...)

PROG: Create local variables, execute each expression and return the value of the last or as specified in the *RETURN* statement. Allows *GO* and *RETURN*.

11 Local Variables

```
(let ((var1 val1) ; initialize var1 to val1
      var2 ; initialize var2 to nil
      ...)
     exp1
     exp2
     ...)
```

LET: Creates local variables and executes the expressions in their context. Returns the value of the last expression.

```
(let* (vars) exp1 exp2 ...)
```

LET:* Sames as *LET* except the variables are assigned sequentially. Previously defined variables may be used in subsequent variable initializations.

12 Defining Functions

```
(defun function-name (arg1 arg2 arg3 ...)
     exp1
     exp2
     ...)
```

DEFUN: Define function named *function-name* with specified arguments, and run the expressions in that context.

12.1 Lambda Argument Types

Example argument list	Comment
(arg1 &optional arg2)	*arg2* is optional, defaults to *null*
(arg1 &optional (arg2 5))	*arg2* is optional, defaults to *5*
(arg1 &rest rst)	*rst* is a list containing the remaining arguments
(arg1 &key abc def)	two key-word arguments, usage: (func :def 55) or (func :abc 44 :def 77)
(arg1 &key (abc 88))	keyword argument with default *88*
(arg1 &aux var1 var2)	two auxiliary or local varables (not arguments)

13 Conditionals

In Lisp, conditions are considered true if they are any value other than *NIL*. Therefore, *NIL* is considered as *false*, and all other values are treated as *true* conditions.

```
(if test
     exp1
     [exp2])
```

IF: If *test* expression is *true*, return *exp1*. *exp2* is optional. If *test* is *false*, return *exp2* or *NIL*.

(cond
(test1 exp1 exp2 ...)
(test2 exp1 exp2 ..)
...
(T exp1 exp2 ...))

COND: Evalute each *testN* until one is *true*. If a *true* test is found, its expressions are evaluated and the result of the last one is returned. The *T* test is always *true*.

14 Numeric Operators

Function	Description
+ – * /	math functions
(1+ n)	add one
(1– n)	subtract one
(– n)	negate number
(abs n)	absolute value
(floor n)	return integer rounded down
(ceiling n)	return integer rounded up
(truncate n)	return integer portion of number
(sqrt n)	square root
(expt x y)	x to power of y
(min n1 n2 ...)	minimum
(max n1 n2 ...)	maximum

15 Iteration

(loop exp1 exp2 ...) *LOOP:* Loop indefinitely until a *(return exp)* is evaluated. This is a simple loop. Cor

(loop exp1 exp2 ...)

LOOP: Loop indefinitely until a *(return exp)* is evaluated. This is a simple loop. Common Lisp has another very full-featured loop.

(dolist (var lst [rtn]) exp1 exp2 ...)

DOLIST: Create new variable *var*, evaluate *lst* and set *var* to successive elements of it. Return *rtn* when done.

(dotimes (var end-val [rtn]) exp1 exp2 ...)

DOTIMES: Create new variable *var* and initialize it to 0. Perform body *end-val* times incrementing *var* with each iteration. Return *rtn* when done.

(do ((var1 val1 rep1) ...)
(test exp1 ...)
body1 ...)

DO: Create new variable *varn* and initialize it to *valn*. *repn* is used to provide successive values to *varn* on each iteration. If *test* returns non-nil, execute *expn* and exit *DO* with the last value. If test fails, execute *bodyn* and repeat.

Also, see *PROG* and *Loop*

16 Loop

(loop for x to 10 do (print x) ...)	Loop from 0 to 10 setting x equal to each number.
(loop for x to 10 do (print x) ...)	Loop from 0 to 10 setting x equal to each number.
(loop for x from 5 to 10 do (print x) ...)	Loop from 5 to 10 setting x equal to each number.
(loop for x in '(a b c) do (print x) ...)	Loop over each element of the list '(a b c) setting x equal to each atom.
(loop while [condition] do ...)	While loop

Loop has many clauses that may be mixed and matched as follows.

16.1 Loop Variables

- for <var> from [number]
- for <var> to [number] starts at 0
- for <var> from [number] to [number]
- for <var> below [number] while <var> is below [number]
- for <var> in [list]
- for <var> across [vector]
- for <var> = <exp>
- for (<var1> <var2>) in '((val1 val2) (val1 val2) (val1 val2) ...)
- with <var> [= val]

16.2 Termination Clause

- while <exp>

- until <exp>
- (loop-finish) used anywhere, terminates loop
- (return <exp>)
- (return-from [block] <exp>)

16.3 Body of the loop

- do <exp1> <exp2> <exp3> ...

16.4 Prolog and Epilogue

- initially <exp>
- finally <exp>)

16.5 Value Collection

- collect <exp> [into <var>]
- append <exp> [into <var>])
- nconc <exp> [into <var>])
- summing <exp> [into <var>])
- maximizing <exp> [into <var>])
- minimizing <exp> [into <var>])

17 Displaying

(print x)	newline, lisp-object, space
(prin1 x)	lisp-object - show as a lisp object
(princ x)	object - i.e. no quotes around strings
(write x)	like prin1 except has a lot of options
(pprint x)	newline, lisp-object BUT pretty prints lisp-object
(terpri)	new-line
(finish-output)	flush output
(write-to-string x)	
(prin1-to-string x)	
(princ-to-string x)	

18 Format

(format dest fmt arg1 ...)

dest

t	stdout
nil	return string
fp	output to file
string	output to string with fill pointer
error-output	output to stderr

fmt

~N%	N newlines (default 1)
~c	character
~Na	string, N wide, left justify, no quotes around strings
~s	display as a lisp object
~w	use write
~~	~
~$	2 digit floating point
~Nd	decimal, N wide, right justify
~N:d	decimal, N wide, right justify, add commas
~N,Pf	floating point, N wide, P decimal places, right justify
~N,'0d	decimal, N wide, zero fill

19 Arrays

(make–array '(2 3))	create general array with 2x3 dimensions
(make–array '(2 3) :element-type 'integer)	create 2x3 integer array
(aref a 2 1)	get value at [2,1] index origin 0
(setf (aref a 2 1) val)	set array at index [2,1] to val
(array-rank a)	return the number of dimensions in array
(array-dimensions a)	get dimensions of array
(array-dimension a 0)	get the 0th array dimension

20 Hash Tables

(make–hash–table)	create a new hash table
(gethash key ht)	get value of key in hash table
(setf (gethash key ht) val)	add a new entry into the hash table
(gethash key ht)	test if a key is present
(remhash key ht)	remove key / value from hash table
(maphash fun ht)	enumerate over hash table, fun gets passed key and value
(hash-table-count ht)	return count of entries in ht
(clrhash ht)	clear contents of hash table

21 Structures

(defstruct *structname elm1 elm2* ...)	create a structure *structname* with the defined elements
(make-*structname keyword-args*)	create a new instance of *structname* with possible initialization of the specified elements
(*structname–elmN* instance)	access *elmN* of *instance* (use *setf* to set)
(*structname*–p val)	test if *val* is an instance of *structname*

22 Function References

Obtaining a function reference:

#'function–name
#'(lambda ...
(function function–name)
(function (lambda ...

Calling a function reference:

(funcall function–reference arg1 arg2 ...)
(apply function–reference '(arg1 arg2 ...))

23 Mapping

Mapping is the process of iterating a function reference over a group of items.

(map result–type function–reference arg1–list arg2–list ...)

result–type	nil	no result
	'list	return a list of result values
function–reference	#'myfunc	
	#'(lambda ...	
argN–list	list of arguments. Each argN corresponds to a consecutive argument of *function*. Each list must be the same length.	

(mapcar function arg1–list arg2–list ...)

Execute *function* with consecutive elements of *argN* returning a list of result values.

(mapc function arg1–list arg2–list ...)

Execute *function* with consecutive elements of *argN* returning *arg1–list*. It does not keep the return values.

24 Packages & Modules

Defining a package:

```
(defpackage "X"
  (:nicknames "Y")
  (:use "COMMON-LISP")
  (:export "FUN1" "FUN2")
  (:shadow "LOAD" "COMPILE-FILE" "EVAL"))
(in-package "X")
(provide "X")
```

Using a package:

> A package (*pkg*) contains exported (*esym*) and non-exported names (*nesym*). Package contents are not accessible until they are loaded (i.e. via *load* or *require*). After the package is loaded, the symbols in that package are accessible in two different ways depending on whether an *use-package* has been executed or not.
>
> `Before use-package:`
>
> ```
> pkg:esym
> pkg::nesym
> ```
>
> `After use-package:`
>
> ```
> esym
> pkg::nesym
> ```

Package and module facilities

package	the current package
(list-all-packages)	
COMMON–LISP–USER	default user package
(package–name pkg)	return the name of the package
(find-package name)	find package with given name
(symbol-package 'symbol)	return package symbol is a member of
(defpackage name ...)	define a new package
(in–package name)	change to the package name
(use–package name)	import the exported symbols from package name
modules	a list of the currently loaded modules
(provide name)	adds name to *modules* to prevent duplicate loads
(require name &optional path)	load path if name is not in *modules*

25 Global Variables

(defparameter var val [doc–string])	set *var* to *val* always
(defvar var val [doc–string])	set *var* to *val* only if not already defined

26 Macros

Macros are similar to functions except:

1. the arguments are not evaluated
2. instead of defining operations to execute, macros define lisp code to be executed
3. macros have no runtime overhead because they are resolved at compile time
4. macros take the same variety of arguments types as functions

26.1 defmacro

(defmacro macro-name (arg1 arg2 arg3 ...) `(template))	Define a macro named *macro-name* with unevaluated arguments, create the lisp code, and then execute after creating the code.

26.2 Macro Templates

The ` character works like a quote except that the code inside the ` (the template) can have real values inserted into the quoted result. Within the template, the following insertions are in affect:

,var	insert the *value* of var
,@var	splice in the *value* of var

27 Debugging

• (break [msg])	add a breakpoint
• (trace <function-name>)	trace input and output of <function>
• (untrace <function-name>)	stop tracing <function>
• (macroexpand '<exp>)	show full macro expansion of <exp>
• (macroexpand-1 '<exp>)	show expansion of <exp>

28 Exceptions

```
(ignore-errors
  <exp>)
```

```
(error <error-data>)                    throw an error
                                        (<error-data> can be anything)
```

```
(handler-case                           catch the error
  <exp>
  (t (error-data)
     <return-on-error-exp1>
     <return-on-error-exp2>
     ...))
```

29 Block

```
(block <block-name>
  <exp1>
  <exp2>
  ...)
```

```
(return <exp>)                          return from the current block

(return-from <block-name> <exp>)        can be nested block
```