Core Library Reference

core* name space, version *0.0.5

type identifiers

`%lambda`	closure lambda
`%exception`	exception
`%vector`	vector
`%closure`	lexical closure
bool	false if (), otherwise true
char	
cons	
fixnum	*fix*
float	
func	
keyword	
ns	
null	
stream	
string	
struct	
symbol	*sym*
vector	

Core

+version+	*string*	version string
%format *T string list*	*string*	formatted output
load-file *string*	*bool*	load file through core reader
%make-keyword string		make keyword
%quote *T*	*cons*	quote form
apply *func list*	*T*	apply *func* to *list*
compile *T*	*T*	compile T in null environment
gensym	*sym*	create unique uninterned symbol

Special Form

%defmacro *sym list . body*	*symbol*	define macro
%lambda *list* . body	*func*	define closure
if *T 'T*	*T*	conditional
if *T 'T ''T*	*T*	conditional

Fixnum

1+ *fix*	*fix*	increment *fix*
1- *fix*	*fix*	decrement *fix*
logand *fix 'fix*	*fix*	bitwise and
lognot *fix*	*fix*	bitwise negate
logor *fix 'fix*	*fix*	bitwise or
logxor *fix 'fix*	*fix*	bitwise xor

List

%dropl *list fixnum*	*list*	drop left
%dropr *list fixnum*	*list*	drop right
%findl-if *func list*	*T*	element if applied function returns an atom, () otherwise
%foldl *func T list*	*list*	left fold
%foldr *func T list*	*list*	right fold
%mapc *func list*		apply *func* to *list* cars, return *list*
%mapcar *func list*	*list*	new list from applying *func* to *list* cars
%mapl *func list*	*list*	apply *func* to *list* cdrs, return *list*
%maplist *func list*	*list*	new list from applying *func* to *list* cdrs
%positionl-if *func list*	*T*	index of element if *func* returns an atom, otherwise ()
%append *list*	*list*	append lists
reverse *list*	*list*	reverse *list*

String

%string-position *char string*	*fix*	index of char in *string*, nil if not found
%substr *string fix 'fix*	*string*	substring of *string* from start to end

Vector

%make-vector *list*	*vector*	specialized vector from list
%map-vector *func vector*	*vector*	make vector of *func* applications on *vector* elements
make-vector *list*	*vector*	general vector from list
bit-vector-p *vector*	*bool*	bit vector?
vector-displaced-p *vector*	*bool*	a displaced vector?
vector-length *vector*	*fix*	length of *vector*
vector-ref *vector fix*	*T*	element of *vector* at index *fix*
vector-slice *vector fix 'fix*	*vector*	displaced vector from start to end
vector-type *vector*	*symbol*	vector type

Macro

define-symbol-macro *sym T*	*symbol*	define symbol macro
macro-function *sym list*	*T*	extract macro function with environment
macroexpand *T list*	*T*	expand macro expression in environment
macroexpand-1 *T list*	*T*	expand macro expression once in environment

Predicate

minusp *fix*	*bool*	negative *fix*
numberp *T*	*bool*	*float* or *fixnum*
%uninternedp *sym*	*bool*	*symbol* interned
charp *T*	*bool*	*char*
consp *T*	*bool*	*cons*
fixnump *T*	*bool*	*fixnum*
floatp *T*	*bool*	*float*
functionp *T*	*bool*	function
keywordp *T*	*bool*	keyword
listp *T*	*bool*	*cons* or ()
namespacep *T*	*bool*	*namespace*
null *T*	*bool*	:nil or ()
streamp *T*	*bool*	*stream*
stringp *T*	*bool*	*char vector*
structp *T*	*bool*	*struct*
symbolp *T*	*bool*	*symbol*
vectorp *T*	*bool*	vector

Type System

%core-type-p *T*	*bool*	a core type?
def-type *symbol list*	*struct*	create core type of name *symbol*
type-of *T*	*sym*	core type symbol
typep *T typespec*	*bool*	does *T* conform to typespec?

Stream

%peek-char *stream*	*char*	read char from stream, unread
%format *T string list*	*T*	formatted output to stream
read *stream bool T*	*T*	read from stream with EOF handling
write *T bool stream*		write escaped object to stream

Exception

%exceptionf *stream string bool struct*	*string*	format exception
%make-exception *sym T string sym list*	*struct*	create exception
error *T symbol list*	*string*	error format
exceptionp *struct*	*bool*	predicate
raise *T symbol list*		raise exception
raise-env *T symbol list*		raise exception
warn *T string*	*T*	warning
with-exception *func func*	*T*	catch exception

Macro Definitions

and &rest ...	*T*	and of ...
cond &rest ...	*T*	cond switch
let *list* &rest ...	*T*	lexical bindings
let* *list* &rest ...	*T*	dependent list of bindings
or &rest ...	*T*	or of ...
progn &rest ...	*T*	evaluate rest list, return last evaluation
unless *T* &rest ...	*T*	if T is (), (**progn** ...) otherwise ()
when *T* &rest ...	*T*	if T is an *atom*, (**progn** ...) otherwise ()

Closures

append &rest ...	*list*	append lists
format *T string* &rest ...	*T*	formatted output
funcall *func* &rest ...	*T*	apply *func to* ...
list &rest ...	*list*	*list of* ...
list* &rest ...	*list*	append ...
vector &rest	*vector*	*vector* of ...

Modules

provide *string list*	*T*	module definition
require *string*	*bool*	module load
require-lib *string*	*bool*	lib module load

Reader Syntax

`;`	comment to end of line
`#|...|#`	block comment
‘*form*	quoted form
`` ` ``*form*	backquoted form
`` ` ``(...)	backquoted list (proper lists)
,*form*	eval backquoted form
,@*form*	eval-splice backquoted form
(...)	constant *list*
`()`	empty *list*, prints as `:nil`
(... . .)	dotted *list*
“...”	*string, char vector*
`\`	single escape in strings
`#*...`	bit vector
`#x...`	hexadecimal *fixnum*
`#.`	read-time eval
`#\.`	*char*
`#(:type ...)`	*vector*
`#s(:type ...)`	*struct*
`#:symbol`	uninterned *symbol*
“`` ` `` , ;	terminating macro char
`#`	non-terminating macro char
`!$%&*+-.` `<>=?@[]|` `:^_{}~/` `A..Za..z` `0..9`	symbol constituents
`0x09 #\tab` `0x0a #\linefeed` `0x0c #\page` `0x0d #\return` `0x20 #\space`	whitespace