Core Library Reference

core* name space, version *0.0.11

type identifiers

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

Core

load-file *string*	*bool*	load file through core reader
%make-keyword *string*	*keyword*	make keyword
eval *T*	*T*	eval form
apply *fn list*	*T*	apply *fn* to *list*
compile *T*	*T*	compile *T* in null environment
gensym	*sym*	create unique uninterned symbol
eql *T T*	*bool*	eql predicate

Special Forms

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

List

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

String

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

Vector

%make-vector *list*	*vec*	specialized vector from *list*
%map-vector *fn vector*	*vec*	mapc for vectors
make-vector *list*	*vec*	general vector from list
bit-vector-p *vector*	*bool*	bit vector?
vector-displaced-p *vec*	*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*	*vec*	displaced vector from start for length
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 null environment
macroexpand *T list ()*	*T*	expand macro expression in null environment
macroexpand-1 *T list ()*	*T*	expand macro expression once in null environment

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

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*	*T*	write escaped object to stream

Predicates

minusp *fix*	*bool*	negative value
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

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*	*T*	write escaped object to stream

Exception

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

Macro Definitions

and &rest ...	*T*	logical and of ...
cond &rest ...	*T*	cond switch
let *list* &rest ...	*T*	lexical bindings
let* *list* &rest ...	*T*	dependent list of bindings
or &rest ...	*T*	logical or of ...
progn &rest ...	*T*	evaluate rest list, return final 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 *fn* &rest ...	*T*	apply *fn to* ...
list &rest ...	*list*	*list of* ...
list* &rest ...	*list*	append ...
mapc *fn* &rest ...	*list*	mapc of ...
mapcar *fn* &rest ...	*list*	mapcar of ...
mapl *fn* &rest ...	*list*	mapl of ...
maplist *fn* &rest ...	*list*	maplist of ...

Modules

modules	*list*	module definitions
module-version *string*	*string*	module version
module-namespace *string*	*ns*	module namespace
provide *string list*	*T*	define module
require *string*	*bool*	load module

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
`!$%&*+-.`	symbol constituents
`<>=?@[]|`	
`:^_{}~/`	
`A..Za..z`	
`0..9`	
`0x09 #\tab`	whitespace
`0x0a #\linefeed`	
`0x0c #\page`	
`0x0d #\return`	
`0x20 #\space`	