

Core Library Reference

core name space, version 0.0.10

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

%format <i>T string list</i>	string	formatted output
load-file <i>string</i>	bool	load file through core reader
%make-keyword <i>string</i>	keyword	make keyword
%quote <i>T</i>	cons	quote form
eval <i>T</i>	T	eval form
apply <i>fn list</i>	T	apply <i>fn</i> to <i>list</i>
compile <i>T</i>	T	compile <i>T</i> in null environment
gensym	sym	create unique uninterned symbol
eql <i>T T</i>	bool	eql predicate

Special Forms

%defmacro <i>sym list . body</i>	symbol	define macro
%lambda <i>list . body</i>	fn	define closure
%if <i>T 'T</i>	T	conditional
%if <i>T 'T 'T</i>	T	conditional

Fixnum

1+ <i>fix</i>	fix	increment <i>fix</i>
1- <i>fix</i>	fix	decrement <i>fix</i>
logand <i>fix 'fix</i>	fix	bitwise and
lognot <i>fix</i>	fix	bitwise negate
logor <i>fix 'fix</i>	fix	bitwise or
logxor <i>fix 'fix</i>	fix	bitwise xor

List

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

String

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

Vector

%make-vector <i>list</i>	vector	specialized vector from <i>list</i>
%map-vector <i>fn vector</i>	vector	mapc for vectors
make-vector <i>list</i>	vector	general vector from <i>list</i>
bit-vector-p <i>vector</i>	bool	bit vector?
vector-displaced-p <i>vector</i>	bool	a displaced vector?
vector-length <i>vector</i>	fix	length of <i>vector</i>
vector-ref <i>vector fix</i>	T	element of <i>vector</i> at index <i>fix</i>
vector-slice <i>vector fix 'fix</i>	vector	displaced vector from start for length
vector-type <i>vector</i>	symbol	vector type

Macro

define-symbol-macro <i>sym T</i>	symbol	define symbol macro
macro-function <i>sym list</i>	T	extract macro function with environment
macroexpand <i>T list</i>	T	expand macro expression in environment
macroexpand-1 <i>T list</i>	T	expand macro expression once in environment

Predicate			Exception			Modules		
minusp <i>fix</i>	<i>bool</i>	negative <i>fix</i>	%exceptionf <i>stream string bool struct</i>			modules <i>list</i>		module definitions
numberp <i>T</i>	<i>bool</i>	<i>float</i> or <i>fixnum</i>		<i>string</i>	format exception	module-version <i>string</i>		module version
%uninternedp <i>sym</i>	<i>bool</i>	<i>symbol</i> interned	%make-exception <i>sym T string sym list</i>				<i>string</i>	module version
charp <i>T</i>	<i>bool</i>	<i>char</i>		<i>struct</i>	create exception	module-namespace <i>string</i>		module
consp <i>T</i>	<i>bool</i>	<i>cons</i>	error <i>T symbol list</i>	<i>string</i>	error format		<i>ns</i>	namespace
fixnump <i>T</i>	<i>bool</i>	<i>fixnum</i>	exceptionp <i>struct</i>	<i>bool</i>	predicate	provide <i>string list</i>	<i>T</i>	define module
floatp <i>T</i>	<i>bool</i>	<i>float</i>	raise <i>T sym str</i>		raise exception	require <i>string</i>	<i>bool</i>	load module
functionp <i>T</i>	<i>bool</i>	<i>fn</i> tion	raise-env <i>T sym str</i>		raise exception			
keywordp <i>T</i>	<i>bool</i>	<i>keyword</i>	warn <i>T string</i>	<i>T</i>	warning			
listp <i>T</i>	<i>bool</i>	<i>cons</i> or <i>()</i>	with-exception <i>fn fn T</i>		catch exception			
namespacep <i>T</i>	<i>bool</i>	<i>namespace</i>						
null <i>T</i>	<i>bool</i>	<i>:nil</i> or <i>()</i>						
streamp <i>T</i>	<i>bool</i>	<i>stream</i>						
stringp <i>T</i>	<i>bool</i>	<i>char</i> vector						
structp <i>T</i>	<i>bool</i>	<i>struct</i>						
symbolp <i>T</i>	<i>bool</i>	<i>symbol</i>						
vectorp <i>T</i>	<i>bool</i>	<i>vector</i>						
Type System			Macro Definitions			Reader Syntax		
%core-type-p <i>T</i>	<i>bool</i>	a core type?	and &rest ...	<i>T</i>	and of ...	<i>;</i>		comment to end of line
def-type <i>symbol list</i>	<i>struct</i>	create core type of name <i>symbol</i>	cond &rest ...	<i>T</i>	cond switch	<i># ... #</i>		block comment
type-of <i>T</i>	<i>sym</i>	core type symbol	let <i>list</i> &rest ...	<i>T</i>	lexical bindings	<i>`form</i>		quoted form
			let* <i>list</i> &rest ...	<i>T</i>	dependent list of bindings	<i>`form</i>		backquoted form
			or &rest ...	<i>T</i>	or of ...	<i>`(...)</i>		backquoted list (proper lists)
			progn &rest ...	<i>T</i>	evaluate rest list, return last evaluation	<i>,form</i>		eval backquoted form
			unless <i>T</i> &rest ...	<i>T</i>	if <i>T</i> is <i>()</i> , (progn ...)	<i>,@form</i>		eval-splice backquoted form
			when <i>T</i> &rest ...	<i>T</i>	if <i>T</i> is an <i>atom</i> , (progn ...) otherwise <i>()</i>	<i>(...)</i>		constant <i>list</i>
						<i>()</i>		empty <i>list</i> , prints as <i>:nil</i>
						<i>(... . .)</i>		dotted <i>list</i>
						<i>"..."</i>		<i>string</i> , <i>char</i> vector
						<i> </i>		single escape in strings
						<i>#*...</i>		bit vector
						<i>#x...</i>		hexadecimal <i>fixnum</i>
						<i>#.</i>		read-time eval
						<i>#\.</i>		<i>char</i>
						<i>#(:type ...)</i>		<i>vector</i>
						<i>#s(:type ...)</i>		<i>struct</i>
						<i>#:symbol</i>		uninterned <i>symbol</i>
						<i>"` , ;</i>		terminating macro char
						<i>#</i>		non-terminating macro char
						<i>!\$%&*+- .</i>		symbol constituents
						<i><=>?[</i>		
						<i>:^_{ }~ /</i>		
						<i>A..Za..z</i>		
						<i>0..9</i>		
						<i>0x09 #\tab</i>		whitespace
						<i>0x0a #\linefeed</i>		
						<i>0x0c #\page</i>		
						<i>0x0d #\return</i>		
						<i>0x20 #\space</i>		
Stream			Closures					
%peek-char <i>stream char</i>		read char from stream, unread	append &rest ...	<i>list</i>	append lists			
%format <i>T string list T</i>		formatted output to stream	format <i>T string</i> &rest ...	<i>T</i>	formatted output			
read <i>stream bool T</i>	<i>T</i>	read from stream with EOF handling	funcall <i>fn</i> &rest ...	<i>T</i>	apply <i>fn</i> to ...			
write <i>T bool stream</i>	<i>T</i>	write escaped object to stream	list &rest ...	<i>list</i>	<i>list</i> of ...			
			list* &rest ...	<i>list</i>	append ...			
			mapc <i>fn</i> &rest ...	<i>list</i>	mapc of ...			
			mapcar <i>fn</i> &rest ...	<i>list</i>	mapcar of ...			
			mapl <i>fn</i> &rest ...	<i>list</i>	mapl of ...			
			maplist <i>fn</i> &rest ...	<i>list</i>	maplist of ...			