

Mu Namespace

(for libmu version 0.0.20)

Types

<i>T</i>	type superclass
<i>byte</i>	<i>uint8_t</i>
<i>boolean</i>	<i>:t</i> , <i>:nil</i>
<i>fix</i>	<i>fixnum</i> synonym
<i>fn</i>	<i>function</i> synonym
<i>list</i>	<i>cons</i> (), <i>:nil</i>
<i>ns</i>	<i>namespace</i>
<i>string</i>	<i>character</i> vector
<i>type</i>	type keyword
<i>:char</i>	<i>character</i>
<i>:code</i>	<i>code</i>
<i>:cons</i>	<i>cons</i>
<i>:double</i>	64 bit IEEE <i>float</i>
<i>:except</i>	<i>exception</i>
<i>:fixnum</i>	62 bit signed <i>integer</i>
<i>:float</i>	32 bit IEEE <i>float</i>
<i>:func</i>	<i>lambda</i> , native
<i>:keyword</i>	7 byte <i>keyword</i>
<i>:macro</i>	<i>macro</i> forms
<i>:ns</i>	<i>symbol</i> bindings
<i>:stream</i>	file, string, socket, function
<i>:struct</i>	<i>defstruct</i>
<i>:symbol</i>	Lisp-1 binding
<i>:string</i>	<i>:char</i> vector
<i>:vector</i>	<i>T</i> , <i>byte</i> , <i>:char</i> , <i>:fixnum</i> , <i>:float</i>
<i>(type-of T)</i>	type <i>keyword symbol</i>
<i>(eq T T')</i>	are <i>T</i> and <i>T'</i> identical?
<i>(null T)</i>	is <i>T</i> () or <i>:nil</i> ?

Characters

<i>(charp T)</i>	<i>character</i> predicate
<i>(char T)</i>	coerce <i>T</i> to <i>character</i>

Symbols

<i>(symbolp T)</i>	<i>symbol</i> predicate
<i>(boundp symbol)</i>	is <i>symbol</i> bound?
<i>(keywordp symbol)</i>	<i>keyword</i> predicate
<i>(keyword string)</i>	make <i>keyword</i> of <i>string</i>
<i>(symbol-name symbol)</i>	<i>symbol</i> name binding
<i>(symbol-value symbol)</i>	<i>symbol</i> value binding
<i>(symbol-ns symbol)</i>	<i>symbol ns</i> binding
<i>(make-symbol string)</i>	make uninterned <i>symbol</i>

Conses/Lists

<i>(consp T)</i>	<i>cons</i> predicate
<i>(car list)</i>	head of <i>list</i>
<i>(cdr list)</i>	tail of <i>list</i>
<i>(cons T T')</i>	make a <i>cons</i> from <i>T</i> and <i>T'</i>
<i>(list-length list)</i>	length of <i>list</i>
<i>(mapc fn list)</i>	map <i>function</i> over <i>list cars</i>
<i>(mapcar fn list)</i>	make <i>list</i> from <i>list cars</i>
<i>(mapl fn list)</i>	map <i>function</i> over <i>list cdrs</i>
<i>(maplist fn list)</i>	make <i>list</i> from <i>list cdrs</i>
<i>(nth fix list)</i>	<i>nth car</i> of <i>list</i>
<i>(nthcdr fix list)</i>	<i>nth cdr</i> of <i>list</i>

Exceptions

<i>(exceptionp T)</i>	<i>exception</i> predicate
<i>(exception keyword string T)</i>	make <i>exception</i>
<i>(raise string T)</i>	raise type <i>exception</i>
<i>(raise-exception exception)</i>	throw <i>exception</i>
<i>(with-exception function function)</i>	catch <i>exception</i>

Printer

<i>(.print T stream boolean)</i>	print with escapes to <i>stream</i>
<i>(print-unreadable T stream)</i>	print unreadable to <i>stream</i>
<i>(terpri stream)</i>	print newline to <i>stream</i>

Fixnums

<i>(fixnump T)</i>	<i>fixnum</i> predicate
<i>(fixnum T)</i>	coerce <i>T</i> to <i>fixnum</i>
<i>(fixnum* fix fix')</i>	product of <i>fix</i> and <i>fix'</i>
<i>(fixnum+ fix fix')</i>	sum of <i>fix</i> and <i>fix'</i>
<i>(fixnum- fix fix')</i>	difference of <i>fix</i> and <i>fix'</i>
<i>(fixnum< fix fix')</i>	is <i>fix</i> less than <i>fix'</i> ?
<i>(fixnum/ fix fix')</i>	<i>fix</i> divided by <i>fix'</i> (floor)
<i>(logand fix fix')</i>	bitwise and of <i>fix</i> and <i>fix'</i>
<i>(logor fix fix')</i>	bitwise or <i>fix</i> and <i>fix'</i>
<i>(mod fix fix')</i>	modulus of <i>fix</i> and <i>fix'</i>

Floats

<i>(floatp T)</i>	<i>float</i> predicate
<i>(float T)</i>	coerce <i>T</i> to <i>float</i>
<i>(float* float float')</i>	product of <i>float</i> and <i>float'</i>
<i>(float+ float float')</i>	sum of <i>float</i> and <i>float'</i>
<i>(float- float F')</i>	difference of <i>float</i> and <i>float'</i>
<i>(float< float float')</i>	is <i>float</i> less than <i>float'</i> ?
<i>(float/ float float')</i>	<i>float</i> divided by <i>float'</i>
<i>(asin float)</i>	arcsine of <i>float</i> degrees
<i>(acos float)</i>	arccosine of <i>float</i> degrees
<i>(atan float)</i>	arctangent of <i>float</i> degrees
<i>(sin float)</i>	sine of <i>float</i> degrees
<i>(cos float)</i>	cosine of <i>float</i> degrees
<i>(tan float)</i>	tangent of <i>float</i> degrees
<i>(exp float float')</i>	natural exponential
<i>(pow float float')</i>	power function
<i>(log float)</i>	natural logarithm
<i>(log10 float)</i>	base 10 logarithm
<i>(sqrt float)</i>	square root

Vectors

<i>(vectorp T)</i>	<i>vector</i> predicate
<i>(.vec-length vector)</i>	<i>fixnum</i> length of <i>vector</i>
<i>(.vec-map fn vector)</i>	make <i>vector</i> from <i>vector</i>
<i>(.vec-mapc fn list)</i>	map <i>function</i> over <i>vector</i>
<i>(.vec-ref vector fix)</i>	<i>nth</i> element
<i>(.vec-type vector)</i>	type of <i>vector</i> elements

Streams

standard-input	standard input stream
standard-output	standard output stream
error-output	standard error stream
(streamp <i>T</i>)	<i>stream</i> predicate
(close <i>stream</i>)	close <i>stream</i>
(eofp <i>stream</i>)	is <i>stream</i> at end of file?
(get-output-string-stream <i>stream</i>)	get <i>string</i> from <i>stream</i>
(load <i>string</i>)	load file
(open-input-file <i>string</i>)	returns file <i>stream</i>
(open-input-string <i>string</i>)	returns <i>string stream</i>
(open-output-file <i>string</i>)	returns file <i>stream</i>
(open-output-string <i>string</i>)	returns <i>string stream</i>
(open-function-stream <i>fn</i>)	returns <i>function stream</i>
(open-socket-server <i>fixnum</i>)	returns socket <i>stream</i>
(open-socket-stream <i>fixnum fixnum</i>)	returns socket <i>stream</i>
(accept-socket-stream <i>stream</i>)	accept socket <i>stream</i>
(connect-socket-stream <i>stream</i>)	connect socket <i>stream</i>
(read-byte <i>stream</i>)	read <i>byte</i> from <i>stream</i>
(read-char <i>stream</i>)	read <i>char</i> from <i>stream</i>
(unread-char <i>stream</i>)	push <i>char</i> onto <i>stream</i>
(write-char <i>char stream</i>)	write <i>char</i> to <i>stream</i>
(write-byte <i>byte stream</i>)	write <i>byte</i> to <i>stream</i>

Functions

(codep <i>T</i>)	<i>code</i> predicate
(functionp <i>T</i>)	<i>function</i> predicate
(.apply <i>F list</i>)	apply <i>function</i> to arg <i>list</i>
(eval <i>T</i>)	evaluate form
(closure <i>fn</i>)	reify lexical environment
(frame-ref <i>fix fix</i>)	lexical variable of frame
(.trampoline <i>fn</i>)	trampoline

Namespaces

(namespacep <i>T</i>)	<i>namespace</i> predicate
(intern <i>ns :keyword string T</i>)	intern in <i>namespace</i>
(find-ns <i>string</i>)	map <i>string</i> to <i>namespace</i>
(find-in-ns <i>ns :keyword string</i>)	map <i>string</i> to <i>symbol</i>
(find-symbol <i>ns string</i>)	resolve <i>symbol</i> in <i>namespace</i>
(in-ns <i>ns</i>)	set the current <i>namespace</i>
(ns <i>string ns</i>)	xsmake <i>namespace</i> , import <i>ns</i>
(ns-current)	current <i>namespace</i>
(ns-name <i>ns</i>)	<i>namespace</i> 's name
(ns-symbols <i>ns</i>)	list of <i>namespace</i> 's symbols
(ns-import <i>ns</i>)	<i>namespace</i> 's import

Miscellaneous

(.block <i>symbol fn</i>)	establish named <i>block</i>
(.return <i>symbol T</i>)	return value from <i>block</i>
(.if <i>fn fn' fn</i>)	support <i>if macro</i>
(.letq <i>symbol T</i>)	modify lexical value
(.env-stack <i>fix fix</i>)	stack as a list of frames
(.env-stack-depth)	stack depth as a <i>fixnum</i>
(gc <i>boolean</i>)	garbage collection
(heap-info <i>T</i>)	heap occupancy for type
(heap-log <i>boolean</i>)	enable heap logging
(view <i>T</i>)	make view of <i>T</i>

Structs

(structp <i>T</i>)	<i>struct</i> predicate
(struct <i>keyword list...</i>)	make <i>struct</i>
(struct-type <i>struct</i>)	get <i>struct</i> type
(struct-slots <i>struct</i>)	get <i>struct</i> slot values

Special Forms

(special-operatorp <i>symbol</i>)	special operator predicate
(:defcon <i>symbol form</i>)	define constant <i>symbol</i>
(:lambda <i>list . body</i>)	define anonymous <i>function</i>
(:letq <i>symbol T</i>)	modify lexical value
(:macro <i>list . body</i>)	define <i>macro</i> expander
(:quote <i>T</i>)	quote form

Reader

(read <i>stream</i>)	read object from <i>stream</i>
; #!...!#	comment
(...)	list
'<i>T</i>	quote
"..."	<i>string</i>
#<...>	broket
#x	hexadecimal <i>fixnum</i>
#d	decimal <i>fixnum</i>
#o	octal <i>fixnum</i>
#\character	<i>character</i>
#: (type ...)	<i>vector</i>
#'function	closure
#:symbol	uninterned <i>symbol</i>
#.T	read time eval
\	single escape (in strings)
" ' ()	terminating macro char
` , ;	terminating macro char
#	non-terminating macro char
! \$ % & *	constituent
+ - . / :	constituent
< = > ? @	constituent
[] ^ _	constituent
{ } ~ -	constituent
A--Z a--z	constituent
0--9	constituent
Backspace	constituent
Rubout	constituent
Linefeed	whitespace
Newline	whitespace
Page	whitespace
Return	whitespace
Space	whitespace
Tab	whitespace

Macros (see :macro special operator)

(macro-function <i>macro</i>)	extract <i>macro function</i>
(macroexpand <i>T</i>)	expand <i>macro</i> call
(set-macro-character <i>char fn</i>)	reader interface

Platform Reference

(for libmu version 0.0.20)

Functions (in *mu* namespace)

(**exit** *fixnum*) exit *exec* with *fixnum* rc
(**invoke** *fix string*) call external function
(**runtime**) process elapsed time
(**system string**) run shell command
(**systemtime**) system (wall clock) time
(**system-env**) user environment

C++ API

```
Platform(std::list<OptMap>> options,  
         std::list<std::string>> optargs)
```

Streams

```
const StreamId STREAM_ERROR  
  
enum STD_STREAM { STDIN, STDOUT, STDERR }  
  
bool IsClosed(StreamId)  
bool IsEof(StreamId)  
bool IsFile(StreamId)  
bool IsInput(StreamId)  
bool IsOutput(StreamId)  
bool IsStdStream(StreamId)  
bool IsString(StreamId)  
  
void Close(StreamId)  
  
StreamId AcceptSocketStream(StreamId)  
StreamId ConnectSocketStream(StreamId)  
StreamId OpenClientSocketStream(int, int)  
StreamId OpenInputFile(std::string)  
StreamId OpenInputString(std::string)  
StreamId OpenOutputFile(std::string)  
StreamId OpenOutputString(std::string)  
StreamId OpenServerSocketStream(int)  
StreamId OpenStandardStream(STD_STREAM)  
std::string GetStdString(StreamId)  
void Flush(StreamId)  
int ReadByte(StreamId)  
int UnReadByte(int, StreamId)  
void WriteByte(int, StreamId)
```

System

```
void SystemTime(unsigned long *)  
void ProcessTime(unsigned long *)
```

Libmu API

(for libmu version 0.0.20)

```
char** Environment()  
int System(const std::string)  
std::string Invoke(uint64_t, std::string)  
void* libmu_t();  
void* libmu_nil();  
const char* libmu_version();  
void* libmu_eval(void*, void*);  
void* libmu_read_stream(void*, void*);  
void* libmu_read_string(void*, std::string);  
void* libmu_read_cstr(void*, const char*);  
void libmu_print(void*, void*, void*, bool)  
const char* libmu_print_cstr(void*, void*,  
                             bool);  
void libmu_terpri(void*, void*);  
void libmu_withException(void*,  
                         std::function<void(void*)>);  
void* libmu_env_default(Platform*);  
void* libmu_env(Platform*, Platform::StreamId,  
Platform::StreamId, Platform::StreamId);
```

Mu Defined Forms

(for mu version 0.0.17)

in *mu* namespace from *core/mu.l*

.version	<i>symbol</i> constant string
(defun <i>symbol list . body</i>)	define recursive <i>function</i>
(defmacro <i>symbol list . body</i>)	define <i>macro</i> expander
(defconstant <i>symbol T</i>)	define constant <i>symbol</i>
(.recur <i>symbol list . body</i>)	recursive <i>function</i> binding
(append . <i>lists</i>)	append lists, last may be atom
(block <i>symbol . body</i>)	named <i>block macro</i>
(bool <i>T</i>)	coerce <i>T</i> to <i>boolean</i>
(return <i>T</i>)	return from nil <i>block macro</i>
(return-from <i>symbol T</i>)	return from <i>block macro</i>
(and . <i>body</i>)	and <i>macro</i>
(check-type <i>T T' string</i>)	error if <i>T</i> isn't <i>T'</i> <i>macro</i>
(cond . <i>clauses</i>)	cond <i>macro</i>
(foldl <i>fn init list</i>)	reduce <i>list</i> left iterative
(foldr <i>fn init list</i>)	reduce <i>list</i> right recursive
(gensym)	generate unique <i>symbol</i>
(identity <i>T</i>)	identity <i>function</i>
(if <i>fn form form'</i>)	if <i>macro</i>
(let <i>list . clauses</i>)	parallel lexical bind <i>macro</i>
(let* <i>list . clauses</i>)	sequential lexical bind <i>macro</i>
(letf <i>list . clauses</i>)	parallel lexical defun <i>macro</i>
(letf* <i>list . clauses</i>)	sequential lexical defun <i>macro</i>
(listp <i>T</i>)	is <i>T</i> a <i>cons</i> or :nil ?
(or . <i>body</i>)	or <i>macro</i>
(progn . <i>body</i>)	progn <i>macro</i>
(load-once <i>symbol string</i>)	load file discipline
(unless <i>T . body</i>)	if syntactic sugar <i>macro</i>
(when <i>T . body</i>)	if syntactic sugar <i>macro</i>
(list . <i>body</i>)	make <i>list</i> of <i>body</i>
(list* . <i>body</i>)	make dotted <i>list</i> of <i>body</i>

Canon Namespace

(for version 0.0.18)

.version *symbol* constant string

Common

(**atom** *T*) atom predicate
(**typep** *type T*) type predicate
(**1+** *fixnum*) increment *fixnum*
(**1-** *fixnum*) decrement *fixnum*
(**evenp** *fixnum*) even *fixnum* predicate
(**oddp** *fixnum*) odd *fixnum* predicate
(**zerop** *fixnum*) zero *fixnum* predicate
(**evenp** *fixnum*) even *fixnum* predicate
(**eql** *T T'*) *eql* predicate
(**lambda** *list . body*) :lambda syntactic sugar
(**compile** *T*) compile form
(**error** *T string . T*) error as *fmt*
(**pprint** *T stream*) pretty print object
(**describe** *T*) describe object
(**break** *T*) break loop
(**dotimes** *symbol T . body*)
dotimes loop macro
(**dolist** *symbol list . body*)
dolist loop macro
(**copy-list** *list*) copy list

Lists

(**zip-1** *list*) cons pairs in list
(**zip-2** *list list'*) cons pairs from lists

Namespaces

(**with-ns-symbols** *fn ns*)
apply function to symbols

REPL

(**repl**) read-eval-print loop

Sequences

(**reverse** *list*) reverse sequence
(**reduce** *fn list T*) reduce sequence
(**concatenate** *type . sequences*)
concatenate sequences
(**count-if** *fn sequence*) count in sequence
(**length** *sequence*) length of sequence
(**elt** *fixnum sequence*) sequence ref
(**find-if** *fn sequence*) find in sequence

Printer

(**prin1** *T stream*) print with escapes
(**princ** *T stream*) print without escapes
(**prin1-to-string** *T*) print object to string
(**princ-to-string** *T*) print with escapes
(**read-from-string** *string*)
read from string
(**read-from-string-stream** *stream*)
print with escapes
(**with-open-stream** *keyword type string fn*)
with open stream

Core Defined Forms

(for mu version 0.0.17)

in canon namespace from canon/core.l

(**constantp** *T*) constant predicate
(**sequencep** *T*) sequence predicate
(**pairp** *T*) dotted pair predicate
(**schar** *string fixnum*) string accessor
(**string=** *string string'*)
string comparison
(**string** *T*) string coercion
(**stringp** *T*) string predicate
(**vector** *T . rest*) vector from body
(**vector-to-list** *vector*) constant predicate
(**letrec** *list . body*) letrec macro
(**assoc** *T a-list*) assoc lookup
(**pairlis** *keys values*) make alist from lists
(**acons** *key datum a-list*)
cons to alist
(**mapc** *fn . lists*) mapc on lists
(**mapcar** *fn . lists*) mapcar on lists
(**mapl** *fn . lists*) mapl on lists
(**maplist** *fn . lists*) maplist on lists
(**mapc** *fn . lists*) mapc on lists
(**check-type** *T type string*)
raise error on type mismatch
(**fmt** *string . T*) formatted output
(**defstruct** *name . slots*)
defstruct