***Mu* Namespace**

(for libmu version*0.0.22*)

***Types***

*T* type superclass

*byte uint8\_t*

*boolean :t, :nil*

*fix fixnum* synonym

*fn function* synonym

*list*  *cons*, (), :nil

*ns*  *namespace*

*string character* vector

*type* type keyword

:char *character*

:cons *cons*

:condtn *condition*

:double 64 bit IEEE *float*

:fixnum 62 bit signed *integer*

:float 32 bit IEEE *float*

:func *lambda*, native

:keyword 7 byte *keyword*

:macro *macro* forms

:ns *symbol* bindings

:stream file, string, socket, function

:struct defstruct

:symbol Lisp-1 binding

:string **:**charvector

:vector :t :byte **:***char*

**:***fixnum* **:***float*

(**type-of** *T*) type *keyword symbol*

(**eq** *T T’***)** are *T* and *T’* identical?

(**null** *T*) is *T* () or :nil?

(**view** *T*) vector of contents

***Characters***

(**charp** *T*) *character* predicate

(**char** *T)* coerce *T* to *character*

***Symbols***

(**symbolp** *T)* *symbol* predicate

(**boundp** s*ymbol*) is *symbol* bound?

(**keywordp** s*ymbol) keyword* predicate

(**keyword** *string)* make *keyword* of *string*

(**symbol-name** *symbol)*

*symbol* name binding

(**symbol-value** *symbol)*

*symbol* value binding

(**symbol-ns** *symbol)*

*symbol* *ns* binding

(**make-symbol** *string)*

make uninterned *symbol*

***Conses/Lists***

(**consp** *T*) *cons* predicate

(**car** *list*) head of *list*

(**cdr** *list*) tail of *list*

(**cons** *T T’*) make a *cons* from *T* and *T’*

(**length** *list)* length of *list*

(**mapc** *fn list)* map *function* over *list* *cars*

(**mapcar** *fn list)* make *list* from *list* *cars*

(**mapl** *fn list)* map *function* over *list* *cdrs*

(**maplist** *fn list)* make *list* from *list* *cdrs*

(**nth** *fix**list)* nth *car* of *list*

(**nthcdr** *fix**list)* nth *cdr* of *list*

***Conditions***

*(***conditionp** *T) condition* predicate

(**condition** *keyword string T)*

make *condition*

*(***raise** *string T*) raise type *condition*

(**raise-condition** *condition)*

raise *condition*

(**with-condition** *function function*)

catch *condition*

***Printer***

(**print** *T stream boolean)*

printwith escapes to *stream*

(**terpri** *stream)* print newline to *stream*

***Heap***

(**gc** *boolean*) garbage collection

*(::****heap-view*** *T*) heap occupancy for type

***Fixnums***

***(*fixnump***T***)** *fixnum*predicate

(**fixnum** *T)* coerce *T* to *fixnum*

(**fixnum\****fix fix’*) product of *fix* and *fix’*

*(***fixnum*+*** *fix fix’*) sum of *fix* and *fix’*

(**fixnum-***fix fix’*) difference of *fix* and *fix’*

*(***fixnum*<*** *fix fix’*) is *fix* less than *fix’?*

*(***fixnum*/*** *fix fix’*) *fix* divided by *fix’* (floor)

(**logand***fix fix’*) bitwise and of *fix* and *fix’*

(**logor***fix fix’*) bitwise or *fix* and *fix’*

(**mod***fix fix’*) modulus of *fix* and *fix’*

***Floats***

***(*floatp***T***)** *float*predicate

(**float** *T)* coerce *T* to *float*

(**float\****float float’*) product of *float* and *float’*

*(***float*+*** *float float’*) sum of *float* and *float’*

(**float-***float F’*) difference of *float* and *float’*

*(***float*<*** *float float’*) is *float* less than *float’?*

*(***float*/*** *float float’*) *float* divided by *float’*

**(asin***float*) arcsine of *float* degrees

(**acos***float*) arccosine of *float* degrees

(**atan***float*) arctangent of *float* degrees

(**sin***float*) sine of *float* degrees

(**cos***float*) cosine of *float* degrees

(**tan***float*) tangent of *float* degrees

(**exp***float float’*) natural exponential

(**pow***float float’*) power function

(**log***float*) natural logarithm

(**log10***float*) base 10 logarithm

(**sqrt***float*) square root

***Vectors***

(**vectorp** *T*) *vector* predicate

(**vector-length** *vector)*

*fixnum* length of *vector*

(**vector-map** *fn vector)*

make vector from *vector*

(**vector-mapc** *fn list)*

map function over *vector*

(**vector-ref** *vector fix)*

nth element

(**vector-type** *vector)* type of *vector* elements

***Streams***

**standard-input**standard input stream

**standard-output**standard output stream

**error-output**standard error stream

(**streamp** *T) stream* predicate

(**close** *stream)* close *stream*

(**eofp** s*tream)* is *stream* at end of file?

(**get-output-string-stream** *stream)*

get *string* from *stream*

(**load** *string)* load file

(**open-input-file** *string)*

returns file *stream*

(**open-input-string** *string***)**

returns *string* *stream*

(**open-output-file** *string)*

returns file *stream*

(**open-output-string** *string***)**

returns *string* *stream*

(**open-function-stream** *fn***)**

returns *function* *stream*

(**open-socket-server** *fixnum)*

returns socket *stream*

(**open-socket-stream** *fixnum fixnum’***)**

returns socket *stream*

(**accept-socket-stream** *stream)*

accept socket *stream*

(**connect-socket-stream** *stream)*

connect socket *stream*

(**read-byte** *stream***)**

read *byte* from *stream*

(**read-char** *stream***)**

read *char* from *stream*

(**unread-char** *stream)*

push *char* onto *stream*

(**write-char** *char stream)*

write *char* to *stream*

(**write-byte byte** *stream)*

write *byte* to *stream*

***Functions***

(**codep** *T*) *code* predicate

**(functionp** *T*) *function* predicate

**(.apply***F* ***list)*** apply *function* to arg *list*

**(eval** *T****)*** evaluateform

(**closure** *fn*)reify lexical environment

**(frame-ref***fix fix’*) lexical variable of frame

**(trampoline** *fn*) trampoline

***Namespaces***

**(namespacep** *T****)*** *namespace* predicate

**(intern** *ns* **:***keyword**string**T***)**

intern in *namespace*

(**find-ns** *string*) map *string* to *namespace*

*(***find-in-ns** *ns :keyword string*)

map *string* to *symbol*

(**find-symbol** *ns**string*)

resolve *symbol* in *namespace*

(i**n-ns** *ns*) set the current *namespace*

(**ns** *string ns*) xsmake *namespace,* import *ns*

(**ns-current)** current *namespace*

(**ns-name** *ns) namespaces’s* name

(**ns-symbols** *ns*) list of *namespace’s* symbols

(**ns-import** *ns) namespace’s* import

***Internals***

(**::block** *symbol fn***)** establish named *block*

(**::return** *symbol T***)**return value from *block*

***(*::letq** *symbol T*) modify lexical value

(**::env-view**) environment values

(::clocks)

(::exit

(::frame-ref

(::invoke

(::system

***Structs***

**(structp** *T***)** *struct*predicate

**(struct** *keyword list...*) make *struct*

(**struct-type** *struct*) get *struct* type

**(struct-slots *struct*)** get*struct*slot values

***Special Forms***

(**special-operatorp** s*ymbol)*

special operator predicate

(**:defcon***symbol**form*) define constant *symbol*

**(:lambda** *list . body*) define anonymous *function*

***(*:letq** *symbol T*) modify lexical value

(**:macro** *list . body*) define *macro* expander

(**:quote** *T*) quote form

***Reader***

**(read *stream*)** read object from *stream*

**; #|...|#** comment

**(…)** list

**‘***T* quote

**“….”** *string*

**#<...>** broket

**#x** hexadecimal *fixnum*

**#d** decimal *fixnum*

**#o** octal *fixnum*

**#\***character character*

**#(***type* **…)** *vector*

**#’***function* closure

**#:***symbol* uninterned *symbol*

**#.***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** *symbol*)

extract *macro function*

**(macroexpand** *T)* expand *macro* call

(**set-macro-character** *char* *fn*)

reader interface

**Platform Reference**

(for libmu version*0.0.21*)

***Functions* (in** *mu* **namespace)**

**(exit** *fixnum***)**exit *exec* with *fixnum* rc

**(invoke** *fix string***)** call externalfunction

(**runtime**) process elapsed time

(**system** *string*) run shell command

(**systime**) 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*0.0.21*)

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*0.0.22*)

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

**::version** *symbol* constant string

(**defun** *symbol**list . body***)**

define recursive *function*

***(*defmacro***symbol**list . body****)***

define *macro* expander

***(*defconstant***symbol**T****)*** defineconstant*symbol*

**(recur***symbol**list*. *body****)***

recursive*function* binding

(**append** . lists**)**append lists, last may be atom

(**block** *symbol . body***)**named *block macro*

*(***bool***T***)**coerce *T* to *boolean*

(**return** *T***)**return from nil *block macro*

(**return-from** *symbol T***)**return from *block macro*

(**and** . *body*) **and** *macro*

(**check-type** *T T’ string*) error if *T* isn’t *T’ macro*  
(**cond** . *clauses*) **cond** *macro*

(**foldl** *fn init list*) reduce *list* left iterative

(**foldr** *fn init list*) reduce *list* right recursive  
(**gensym**) generate unique *symbol*

(**identity** *T*) identity *function*  
(**if** *fn form form’*) **if** *macro*  
(**let** *list . clauses*) parallel lexical bind *macro*

(**let**\* *list . clauses*) sequential lexical bind *macro*

(**letf** *list . clauses*) parallel lexical defun *macro*

(**letf\*** *list . clauses*) sequential lexical defun *macro*

(**listp** *T)* is *T* a *cons* or:nil*?*  
(**or** . *body*) **or** *macro*  
(**progn** . *body*) **progn** *macro*   
(**load-once** *symbol* *string*)

load file discipline

(**unless** *T* . *body*) **if** syntactic sugar *macro*  
(**when** *T .* *body*) **if** syntactic sugar *macro*  
(**list** . *body)* make *list* of *body*

(**list\*** . *body)* make dotted *list* of *body*

***Common* 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*0.0.22*)

***in mu******namespace from core/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