Mu Library Reference

mu* name space, version *0.1.83

type keywords and aliases

supertype	*T*	
bool	`()`, `:nil` are false, otherwise true	
condition	*keyword,* see ***Exception***	
list	`:cons` or `()`, `:nil`	
`:null`	`()`, `:nil`	
`:char`	*char*	
`:cons`	*cons*	
`:fixnum`	*fixnum, fix*	56 bit signed integer
`:float`	*float, fl*	32 bit IEEE float
`:func`	*function, fn*	function
`:keyword`	*keyword, key*	symbol
`:ns`	*namespace, ns*	namespace
`:stream`	*stream*	file or string type
`:struct`	*struct*	typed vector
`:symbol`	*symbol, sym*	LISP-1 symbol
`:vector`	*vector, string, str*	

`:char :t :byte :fixnum :float`

Heap

heap-info *vector* heap information
`#(:t type pages pagesize)`

heap-stat *vector* heap allocations
`#(:t :type size total free ...)`

heap-size *T* *fixnum* heap occupancy

Frames

%frame-stack *list* active *frame*s
%frame-pop *fn* *fn* pop *function's* top frame binding
frame binding: `(fn . #(:t …))`

%frame-push *frame* *cons* push frame
%frame-ref *fn fix* *T* *function*, offset

Symbols

boundp *symbol* *bool* is *symbol* bound?
make-symbol *string* *symbol* uninterned *symbol*
symbol-namespace *symbol* *key* *namespace*
symbol-name *symbol* *string* name binding
symbol-value *symbol* *T* value binding

Special Forms

:lambda *list . List'* *function* anonymous *function*
:quote *form* *list* quoted form
:if *form T T'* *T* conditional

Core

apply *fn list* *T* apply *function* to *list*
eval *form* *T* evaluate *form*
eq *T T'* *bool* *T* and *T'* identical?
type-of *T* *key* type keyword
compile *form* *T* *mu* form compiler
view *form* *vector* vector of object

%if *T T' T"* *key* **:if** implementation

repr *type T* *T* tag representation

type `:t :vector`

if `type` is `:vector`, return 8 byte byte vector of argument tag bits, otherwise convert argument byte vector to tag.

fix *fn T* *T* fixpoint of *function*
gc *bool* garbage collection

+version+ *string* version string

Futures

defer *fn list* *struct* future application
detach *fn list* *struct* future application

force *struct* *T* force completion
poll *struct* *bool* poll completion

Fixnum

mul *fix fix'* *fixnum* product
add *fix fix'* *fixnum* sum
sub *fix fix'* *fixnum* difference
less-than *fix fix'* *bool* *fix < fix'?*
div *fix fix'* *fixnum* quotient
ash *fix fix'* *fixnum* arithmetic shift
logand *fix fix'* *fixnum* bitwise and
logor *fix fix'* *fixnum* bitwise or
lognot *fix* *fixnum* bitwise complement

Float

fmul *fl fl'* *float* product
fadd *fl fl'* *float* sum
fsub *float* difference
fless-than *fl fl'* *bool* *fl < fl'?*
fdiv *fl fl'* *float* quotient

Conses/Lists

append *list* *list* append lists
car *list* *list* head of *list*
cdr *list* *T* tail of *list*
cons *T T'* *cons* (*form . form*')
length *list* *fixnum* length of *list*
nth *fix list* *T* *n*th *car* of *list*
nthcdr *fix list* *T* *n*th *cdr* of *list*

Vectors

make-vector *key list* *vector* specialized vector from list
vector-length *vector* *fixnum* length of *vector*
vector-type *vector* *key* type of *vector*
svref *vector fix* *T* *n*th element

Reader/Printer

read *stream bool T* *T* read stream object
write *T bool stream* *T* write escaped object

Structs

make-struct *key list* *struct* of type *key* from *list*
struct-type *struct* *key* *struct* type *keyword*
struct-vec *struct* *vector* of *struct* members

Exception

with-exception *fn fn' T* — catch exception

fn - `(:lambda (obj cond src) . body)`
fn' - `(:lambda () . body)`

raise *T keyword* — raise exception on *T* with condition:

```
:arity   :div0    :eof     :error   :except
:future  :ns      :open    :over    :quasi
:range   :read    :return  :sigint  :stream
:syntax  :syscall :type    :unbound :under
:write
```

Streams

standard-input	*stream*	std input *stream*
standard-output	*stream*	std output *stream*
error-output	*stream*	std error *stream*
open *type dir string*	*stream*	open *stream*

type `:file` `:string`
dir `:input` `:output` `:bidir`

close *stream*	*bool*	close *stream*
openp *stream*	*bool*	is *stream* open?
flush *stream*	*bool*	flush output *steam*
get-string *stream*	*string*	from *string stream*
read-byte *stream bool T*	*byte*	read *byte* from *stream,* error on eof, *T:* eof value
read-char *stream bool T*	*char*	read *char* from *stream,* error on `eof`, *T:* `eof` value
unread-char *char stream*	*char*	push *char* onto *stream*
write-byte *byte stream*	*byte*	write *byte* to *stream*
write-char *char stream*	*char*	write *char* to *stream*

Namespace

make-namespace *str*	*ns*	make *namespace*
namespace-map	*list*	list of mapped *namespaces*
namespace-name *ns*	*string*	*namespace* name
intern *ns str value*	*symbol*	intern bound symbol
find-namespace *str*	*ns*	map *string* to *namespace*
find *ns string*	*symbol*	map *string* to *symbol*
namespace-symbols *ns list*		*namespace symbols*

Features

```
[dependencies]
default = [ "std", "nix", "ffi", "sysinfo" ]
```

cpu-time	process-time (*u*sec)
nix	uname
std	command, exit
sysinfo	sysinfo (disabled on macOS)
ffi	Rust FFI
prof	mu profiling

mu library API

```
[dependencies]
mu = {
   git = "https://github.com/Software-Knife-and-Tool/mu.git",
   branch=main
}

use mu::{
    Condition, Config, Env, Exception, Result, Tag
};

config string format: "npages:N,gcmode:GCMODE"
                      GCMODE - { none, auto, demand }

impl Env {
  const VERSION: &str
  fn signal_exception()  // enable ^C :sigint exception
  fn config(config: Option<String>) → Option<Config>
  fn new(config: &Config, Option<Vec<u8>>) → Env
  fn apply(&self, func: Tag, args: Tag) → Result<Tag>
  fn compile(&self, form: Tag) → Result<Tag>
  fn eq(&self, func: Tag, args: Tag) → bool;
  fn exception_string(&self, ex: Exception) → String
  fn eval(&self, exp: Tag) → Result<Tag>
  fn eval_str(&self, exp: &str) → Result<Tag>
  fn load(&self, file_path: &str) → Result<bool>
  fn read(&self, st: Tag, eofp: bool, eof: Tag) → Result<Tag>
  fn read_str(&self, str: &str) → Result<Tag>
  fn image(&self) → Result<Vec<u8>>
  fn err_out(&self) → Tag
  fn std_in(&self) → Tag
  fn std_out(&self) → Tag
  fn write(&self, exp: Tag, esc: bool, st: Tag) → Result<()>
  fn write_str(&self, str: &str, st: Tag) → Result<()>
  fn write_to_string(&self, exp: Tag, esc: bool) → String
```

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

mu-sys

```
mu-sys: 0.0.2: [celq] [file…]

c: [name:value,…]
e: eval [form] and print result
l: load [path]
q: eval [form] quietly
```