# MzLib: Legacy Libraries

Version 7.1

October 26, 2018

The "mzlib" collection contains wrappers and libraries for compatibility with older versions of Racket. In many ways, the libraries of the "mzlib" collection go with the mzscheme legacy language. Newer variants of many libraries reside in the "scheme" collection.

## **Contents**

1	mzlib/a-signature	5
2	mzlib/a-unit	6
3	mzlib/async-channel	7
4	mzlib/awk	8
5	mzlib/class	10
6	mzlib/cm	11
7	mzlib/cm-accomplice	12
8	mzlib/cmdline	13
9	mzlib/cml	14
10	mzlib/compat	15
11	mzlib/compile	17
12	mzlib/contract	18
13	mzlib/control	23
14	mzlib/date	24
15	mzlib/deflate	25
16	mzlib/defmacro	26
17	mzlib/etc	27
18	mzlib/file	32
19	mzlib/for	34
20	mzlib/foreign	35
21	mzlib/include	36
22	mzlib/inflate	38
23	mzlib/integer-set	39

24	mzlib/kw  24.1 Required Arguments  24.2 Optional Arguments  24.3 Keyword Arguments  24.4 Rest and Rest-like Arguments  24.5 Body Argument  24.6 Mode Keywords  24.7 Property Lists	40 41 41 42 43 44 45 47
25	mzlib/list	48
26	mzlib/match	50
27	mzlib/math	52
28	mzlib/md5	53
29	mzlib/os	54
30	mzlib/pconvert	55
31	mzlib/pconvert-prop	60
32	mzlib/plt-match	61
33	mzlib/port	62
34	mzlib/pregexp	63
35	mzlib/pretty	65
36	mzlib/process	66
37	mzlib/restart	67
38	mzlib/runtime-path	69
39	mzlib/sandbox	70
40	mzlib/sendevent	72
41	mzlib/serialize	73
42	mzlib/shared	74
43	mzlib/string	75
44	mzlib/struct	78

45 mzlib/stxparam	<b>79</b>
46 mzlib/surrogate	80
47 mzlib/tar	81
48 mzlib/thread	82
49 mzlib/trace	84
50 mzlib/traceld	85
51 mzlib/trait	86
52 mzlib/transcr	87
53 mzlib/unit	88
54 mzlib/unit-exptime	89
55 mzlib/unit200	90
<b>56</b> mzlib/unitsig200	91
57 mzlib/zip	92
Bibliography	93
Index	94
Index	94

## 1 mzlib/a-signature

```
(require mzlib/a-signature) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/signature, instead.

Like scheme/signature in #lang form for defining a single signature within a module, but based on mzscheme instead of scheme/base.

### 2 mzlib/a-unit

```
(require mzlib/a-unit) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/unit, instead.

Like scheme/unit in #lang form for defining a single unit within a module, but based on mzscheme instead of scheme/base.

# 3 mzlib/async-channel

```
(require mzlib/async-channel)
    package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/async-channel, instead.

Re-exports scheme/async-channel.

#### 4 mzlib/awk

```
(require mzlib/awk)
                       package: compatibility-lib
(awk next-record-expr
     (record field-id ...)
    maybe-counter
    ((state-variable init-expr) ...)
    maybe-continue
 clause ...)
 maybe-counter =
                id
maybe-continue =
               id
        clause = (test body ...+)
                (test => procedure-expr)
                (/ regexp-str / (id-or-false ...+) body ...+)
                (range excl-start-test excl-stop-test body ...+)
                (:range incl-start-test excl-stop-test body ...+)
                (range: excl-start-test incl-stop-test body ...+)
                (:range: incl-start-test incl-stop-test body ...+)
                 (else body ...+)
                (after body ...+)
           test = integer
                regexp-string
                expr
excl-start-test = test
excl-stop-test = test
incl-start-test = test
incl-stop-test = test
   id-or-false = id
               #f
```

The awk macro from Scsh [Shivers06]. In addition to awk, the Scsh-compatible procedures match:start, match:end, match:substring, and regexp-exec are defined. These match: procedures must be used to extract match information in a regular expression clause

when using the => form.

```
(match:start rec [which]) → exact-nonnegative-integer?
  rec : ....
  which : exact-nonnegative-integer? = 0
(match:end rec [which]) → exact-nonnegative-integer?
  rec : ....
  which : exact-nonnegative-integer? = 0
(match:substring rec [which]) → string?
  rec : ....
  which : exact-nonnegative-integer? = 0
```

Extracts a start position, end position, or substring corresponding to a match. The first argument is the value supplied to the procedure after => in a awk clause or the result of regexp-exec.

```
(regexp-exec re s) → (or/c .... false/c)
  re : (or/c string? regexp?)
  s : string?
```

Matches a regexp to a string, returning a record compatible with match:start, etc.

## 5 mzlib/class

```
(require mzlib/class) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/class, instead.

Re-exports scheme/class, except for the contract constructors.

## 6 mzlib/cm

(require mzlib/cm) package: compatibility-lib

**NOTE:** This library is deprecated; use compiler/cm, instead.

Re-exports compiler/cm.

# 7 mzlib/cm-accomplice

```
(require mzlib/cm-accomplice)
    package: compatibility-lib
```

**NOTE:** This library is deprecated; use compiler/cm-accomplice, instead.

Re-exports compiler/cm-accomplice.

### 8 mzlib/cmdline

```
(require mzlib/cmdline) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/cmdline, instead.

Provides a command-line from that is similar to the one in racket/cmdline, but without using keywords. The parse-command-line procedure from racket/cmdline is reexported directly.

```
(command-line program-name-expr argv-expr clause ...)
     clause = (multi flag-spec ...)
            (once-each flag-spec ...)
            once-any flag-spec ...)
            | (final flag-spec ...)
            (help-labels string ...)
            | (args arg-formals body-expr ...+)
            (=> finish-proc-expr arg-help-expr help-proc-expr
                 unknown-proc-expr)
 flag-spec = (flags id ... help-str ...+ body-expr ...+)
           (flags => handler-expr help-expr)
     flags = flag-string
          (flag-string ...+)
arg-formals = id
            | (id ...)
            (id ...+ . id)
```

Like command-line from racket/cmdline, but without keywords in the syntax.

### 9 mzlib/cml

```
(require mzlib/cml) package: compatibility-lib
```

The mzlib/cml library defines a number of procedures that wrap Racket concurrency procedures. The wrapper procedures have names and interfaces that more closely match those of Concurrent ML [Reppy99].

```
(spawn thunk) \rightarrow thread?
   thunk : (-> any)
Equivalent to (thread/suspend-to-kill thunk).
(channel) \rightarrow channel?
Equivalent to (make-channel).
 (channel-recv-evt ch) \rightarrow evt?
   ch : channel?
Equivalent to ch.
 (channel-send-evt ch \ v) \rightarrow evt?
   ch : channel?
   v : any/c
Equivalent to (channel-put-evt ch v).
 (thread-done-evt thd) \rightarrow any
   thd : thread?
Equivalent to (thread-dead-evt thread).
(current-time) \rightarrow real?
Equivalent to (current-inexact-milliseconds).
 (time-evt tm) \rightarrow evt?
   tm : real?
Equivalent to (alarm-evt tm).
```

### 10 mzlib/compat

```
(require mzlib/compat) package: compatibility-lib
```

The mzlib/compat library defines a number of procedures and syntactic forms that are commonly provided by other Scheme implementations. Most of the procedures are aliases for mzscheme procedures.

```
(=? n ...+) → boolean?
  n : number?
(<? n ...+) → boolean?
  n : real?
(>? n ...+) → boolean?
  n : real?
(<=? n ...+) → boolean?
  n : real?
(>=? n ...+) → boolean?
  n : real?
```

Same as =, <, etc.

```
(1+ n) → number?
  n : number?
(1- n) → number?
  n : number?
```

Same as add1 and sub1.

```
(gentmp [base]) → symbol?
base : (or/c string? symbol?) = "g"
```

Same as gensym.

```
(flush-output-port [o]) → void?
  o : output-port? = (current-output-port)
```

Same as flush-output.

```
(real-time) → exact-integer?
```

Same as current-milliseconds.

```
\begin{array}{c} (atom? \ v) \rightarrow any \\ v : any/c \end{array}
```

Same as (not (pair? v)) (which does not actually imply an atomic value).

Like define-struct, except that the *name-id* is moved inside the parenthesis for fields. In addition, *init-field-ids* can be specified with automatic initial-value expression.

The <code>init-field-ids</code> do not have corresponding arguments for the <code>make-name-id</code> constructor. Instead, each <code>init-field-id</code>'s <code>init-expr</code> is evaluated to obtain the field's value when the constructor is called. The <code>field-ids</code> are bound in <code>init-exprs</code>, but not other <code>init-field-ids</code>.

#### Examples:

```
> (define-structure (add left right) ([sum (+ left right)]))
> (add-sum (make-add 3 6))
9

(getprop sym property [default]) → any/c
   sym : symbol?
   property : symbol?
   default : any/c = #f
(putprop sym property value) → void?
   sym : symbol?
   property : symbol?
   value : any/c
```

The getprop function gets a property value associated with sym. The property argument names the property to be found. If the property is not found, default is returned.

The properties obtained with getprop are the ones installed with putprop.

```
(\text{new-cafe [eval-handler]}) \rightarrow \text{any}

eval-handler : (\text{any/c . -> . any}) = \#f
```

Emulates Chez Scheme's new-cafe by installing eval-handler into the current-eval parameter while running read-eval-print. In addition, current-exit is set to escape from the call to new-cafe.

# 11 mzlib/compile

(require mzlib/compile) package: compatibility-lib

 $\label{lem:compile-file} Re\text{-exports compile-file.}$ 

#### 12 mzlib/contract

```
(require mzlib/contract) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/contract, instead. This library is designed as a backwards compatible library for old uses of contracts. It should not be used for new libraries.

The main differences: the function contract syntax is more regular and function contracts now support keywords, and union is now or/c.

The mzlib/contract library re-exports many bindings from racket/contract:

```
</c
                              flat-rec-contract
<=/c
                              guilty-party
=/c
                              integer-in
>/c
                              list/c
>=/c
                              listof
and/c
                              make-none/c
                              make-proj-contract
any
any/c
                              natural-number/c
between/c
                              none/c
box-immutable/c
                              not/c
                              one-of/c
build-compound-type-name
coerce-contract
                              or/c
                              parameter/c
cons/c
contract
                              printable/c
contract-first-order-passes? promise/c
contract-violation->string
                              provide/contract
contract?
                              raise-contract-error
define-contract-struct
                              real-in
false/c
                              recursive-contract
flat-contract
                              string/len
flat-contract-predicate
                              symbols
flat-contract?
                              syntax/c
flat-murec-contract
                              vector-immutable/c
flat-named-contract
                              vector-immutableof
```

It also provides the old version of the following contracts:

```
(define/contract id contract-expr init-value-expr)
```

Attaches the contract contract-expr to init-value-expr and binds that to id.

The define/contract form treats individual definitions as units of blame. The definition itself is responsible for positive (co-variant) positions of the contract and each reference to

id (including those in the initial value expression) must meet the negative positions of the contract.

Error messages with define/contract are not as clear as those provided by provide/contract, because define/contract cannot detect the name of the definition where the reference to the defined variable occurs. Instead, it uses the source location of the reference to the variable as the name of that definition.

```
(box/c c) → flat-contract?
  c : flat-contract?
```

Returns a flat contract that recognizes boxes. The content of the box must match c.

```
(vectorof c) → flat-contract?
c : flat-contract?
```

Accepts a flat contract and returns a flat contract that checks for vectors whose elements match the original contract.

```
(vector/c c ...) → flat-contract?
  c : flat-contract?
```

Accepts any number of flat contracts and returns a flat contract that recognizes vectors. The number of elements in the vector must match the number of arguments supplied to vector/c, and each element of the vector must match the corresponding flat contract.

```
(struct/c struct-id flat-contract-expr ...)
```

Produces a flat contract that recognizes instances of the structure type named by *struct-id*, and whose field values match the flat contracts produced by the *flat-contract-exprs*.

```
(build-flat-contract name predicate) → flat-contract?
  name : symbol?
  predicate : (-> any/c any)
```

Builds a flat contract out of *predicate*, giving it the name *name*. Nowadays, just using *predicate* directly is preferred.

```
(-> contract-dom-expr ... any)
(-> contract-dom-expr ... contract-rng-expr)
```

This is a restricted form of racket/contract's -> contract that does not handle keyword arguments or multiple value results.

The ->\* form matches up to racket/contract's -> and ->\*, according to the following rules; each equation on the left refers to a mzlib/contract combinator; on the right are the racket/contract equivalents.

```
(->* (contract-dom-expr ...) any) =
(-> contract-dom-expr ... any)

(->* (contract-dom-expr ...) (contract-rng-expr ...)) =
(-> contract-dom-expr ... (values contract-rng-expr))

(->* (contract-expr ...) contract-rest-expr any) =
(->* (contract-expr ...) #:rest contract-rest-expr any)

(->* (contract-expr ...) contract-rest-expr (contract-rng-expr ...)) =
(->* (contract-expr ...)
    #:rest contract-rest-expr
    (values contract-reg-expr ...))

(opt-> (contract-req-expr ...) (contact-opt-expr ...) any)
(opt-> (contract-req-expr ...) (contact-opt-expr ...) contract-rng-expr)
```

The opt-> form is a simplified verison of racket/contract's ->\* and appearances of opt-> can be simply replaced with ->\*.

```
(opt->* (contract-req-expr ...) (contact-opt-expr ...) any)
(opt->* (contract-req-expr ...) (contact-opt-expr ...) (contract-rng-expr ...))
```

The opt->\* form matches up to racket/contract's ->\*, according to the following rules; each equation on the left refers to a mzlib/contract combinator; on the right are the racket/contract equivalents.

```
(opt->* (contract-req-expr ...) (contract-opt-expr ...) any) =
(->* (contract-req-expr ...) (contract-opt-expr ...) any)
```

The ->d contract constructor is just like ->, except that the range position is expected to be a function that accepts the actual arguments passed to the function, and returns a contract for the range. For example, this is one contract for sqrt:

It says that the input must be a real number, and so must the result, and that the square of the result is within 0.01 of input.

```
(->d* (contract-dom-expr ...) contract-rng-fun-expr)
(->d* (contract-dom-expr ...) contract-rest-expr contract-rng-fun-expr)
```

The ->d\* contract constructor is a generalization of ->d to support multiple values and rest arguments.

In the two sub-expression case, the first sequence of contracts are contracts on the domain of the function and the second subexpression is expected to evaluate to a function that accepts as many arguments as there are expressions in the first position. It should return multiple values: one contract for each result of the function.

In the three sub-expression case, the first and last subexpressions are just like the sub-expressions in the two sub-expression case; the middle sub-expression si expected to evaluate to a contract on the rest argument.

The ->r form is a simplified version of racket/contract's ->i, where each *contract-dom-expr* is parameterized over all of the *dom-x* variables (and does lax checking; see ->d for details).

The ->pp form, like ->r is a simplified version of racket/contract's ->i, where each contract-dom-expr is parameterized over all of the dom-x variables (and does lax checking; see racket/contract's ->d for details). Unlike ->r, it also has pre- and post-condition expressions; these expressions are also implicitly parameterized over all of the dom-x variables and the post-condition is also parameterized over rng-x, which is bound to the result of the function.

Like ->pp, but with an additional contract for the rest arguments of the function.

```
(case-> mzlib/contract-arrow-contract-expr ...)
```

Builds a contract analogous to case-lambda, where each case comes from one of the contract expression arguments (tried in order).

```
(object-contract [id mzlib/contract-arrow-contract-expr] ...)
```

Builds a contract for objects where each *id* is expected to be a method on the object living up to the corresponding contract

## 13 mzlib/control

**NOTE:** This library is deprecated; use racket/control, instead.

Re-exports scheme/control.

## 14 mzlib/date

(require mzlib/date) package: compatibility-lib

**NOTE:** This library is deprecated; use racket/date, instead.

Re-exports scheme/date.

## 15 mzlib/deflate

**NOTE:** This library is deprecated; use file/gzip, instead.

Re-exports file/gzip.

## 16 mzlib/defmacro

```
(require mzlib/defmacro) package: compatibility-lib
```

**NOTE:** This library is deprecated; use compatibility/defmacro, instead.

Re-exports compatibility/defmacro.

### 17 mzlib/etc

```
(require mzlib/etc) package: compatibility-lib
```

The mzlib/etc library re-exports the following from scheme/base and other libraries:

```
boolean=?
true
false
build-list
build-string
build-vector
compose
local
symbol=?
nand
nor

(begin-lifted expr ...+)
```

Lifts the exprs so that they are evaluated once at the "top level" of the current context, and the result of the last expr is used for every evaluation of the begin-lifted form.

When this form is used as a run-time expression within a module, the "top level" corresponds to the module's top level, so that each expr is evaluated once for each invocation of the module. When it is used as a run-time expression outside of a module, the "top level" corresponds to the true top level. When this form is used in a define-syntax, letrecsyntax, etc. binding, the "top level" corresponds to the beginning of the binding's right-hand side. Other forms may redefine "top level" (using local-expand/capture-lifts) for the expressions that they enclose.

```
(begin-with-definitions defn-or-expr ...)

The same as (block defn-or-expr ...).

(define-syntax-set (id ...) defn ...)
```

Similar to define-syntaxes, but instead of a single body expression, a sequence of definitions follows the sequence of defined identifiers. For each identifier, the *defns* should include a definition for *id*/proc. The value for *id*/proc is used as the (expansion-time) value for *id*.

The define-syntax-set form is useful for defining a set of syntax transformers that share helper functions, though begin-for-syntax now serves essentially the same purposes.

Examples:

The evcase form is similar to case, except that expressions are provided in each clause instead of a sequence of data. After key-expr is evaluated, each value-expr is evaluated until a value is found that is eqv? to the key value; when a matching value is found, the corresponding body-exprs are evaluated and the value(s) for the last is the result of the entire evcase expression.

The else literal is recognized either as unbound (like in the mzscheme language) or bound as else from scheme/base.

```
\begin{array}{c} \text{(identity } v) \to \text{any/c} \\ v : \text{any/c} \end{array}
```

Returns v.

A binding construct that specifies scoping on a per-binding basis instead of a per-expression basis. It helps eliminate rightward-drift in programs. It looks similar to let, except each clause has an additional keyword tag before the binding variables.

Each clause has one of the following forms:

- (val target expr): Binds target non-recursively to expr.
- (rec target expr): Binds target recursively to expr.
- (vals (target expr) ...): The targets are bound to the exprs. The environment of the exprs is the environment active before this clause.
- (recs (target expr) ...): The targetss are bound to the exprs. The environment of the exprs includes all of the targetss.
- (\_ expr ...) : Evaluates the exprs without binding any variables.

The clauses bind left-to-right. When a target is (values id ...), multiple values returned by the corresponding expression are bound to the multiple variables.

#### Examples:

Repeatedly invokes the f procedure until the done? procedure returns #t:

```
(define (loop-until start done? next f)
  (let loop ([i start])
     (unless (done? i)
          (f i)
          (loop (next i)))))
(namespace-defined? sym) → boolean?
  sym : symbol?
```

Returns #t if namespace-variable-value would return a value for sym, #f otherwise.

```
(nand expr ...)
```

```
Same as (not (and expr ...)).
(nor expr ...)
Same as (not (or expr ...)).
(opt-lambda formals body ...+)
Supports optional (but not keyword) arguments like lambda from scheme/base.
(recur id bindings body ...+)
Equivalent to (let id bindings body ...+).
 (rec id value-expr)
 (rec (id arg-id ...) expr)
 (rec (id arg-id ... rest-id) expr)
Equivalent, respectively, to
  (letrec ([id value-expr]) id)
  (letrec ([id (lambda (arg-id ...) value-expr)]) id)
  (letrec ([id (lambda (arg-id ... . rest-id) value-expr)]) id)
 (this-expression-source-directory)
 (this-expression-source-directory datum)
```

Expands to an expression that evaluates to the directory of the file containing the source datum. If datum is not supplied, then the entire (this-expression-source-directory) expression is used as datum.

See scheme/runtime-path for a definition form that works better when creating executables.

If datum has a source module, then the expansion attempts to determine the module's runtime location. This location is determined by preserving the lexical context of datum in a syntax object, extracting its source module path at run time, and then resolving the module path.

Otherwise, datum's source file is determined through source location information associated with datum, if it is present. As a last resort, current-load-relative-directory is used if it is not #f, and current-directory is used if all else fails.

A directory path derived from source location is always stored in bytes in the expanded code, unless the file is within the result of find-collects-dir, in which case the expansion records the path relative to (find-collects-dir) and then reconstructs it using (find-collects-dir) at run time.

```
(this-expression-file-name)
(this-expression-file-name datum)
```

Similar to this-expression-source-directory, except that only source information associated with *datum* or (this-expression-file-name) is used to extract a filename. If no filename is available, the result is #f.

```
(hash-table (quote flag) ... (key-expr val-expr) ...)
```

Creates a new hash-table providing the quoted flags (if any) to make-hash-table, and then mapping each key to the corresponding values.

#### 18 mzlib/file

```
(require mzlib/file) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/file, instead.

The mzlib/file library mostly re-exports from scheme/file:

```
find-relative-path
explode-path
normalize-path
filename-extension
file-name-from-path
path-only
delete-directory/files
copy-directory/files
make-directory*
make-temporary-file
get-preference
put-preferences
fold-files
find-files
pathlist-closure
(call-with-input-file* file proc [mode]) → any
 file : path-string?
 proc : (input-port? -> any)
 mode : (one-of/c 'text 'binary) = 'binary
(call-with-output-file* file
                         proc
                        mode
                         exists) \rightarrow any
 file : path-string?
 proc : (output-port? -> any)
 mode : (one-of/c 'text 'binary) = 'binary
 exists : (one-of/c 'error 'append 'update
                      'replace 'truncate 'truncate/replace)
         = 'error
```

Like call-with-input-fileand call-with-output-file, except that the opened port is closed if control escapes from the body of *proc*.

Like build-path, but with extra constraints to ensure a relative or absolute result.

### 19 mzlib/for

(require mzlib/for)

```
NOTE: This library is deprecated; use racket/base, instead.
The mzlib/for library re-exports from scheme/base:
 for/fold for*/fold
 for for*
 for/list for*/list
 for/lists for*/lists
 for/and for*/and
 for/or for*/or
 for/first for*/first
 for/last for*/last
 for/fold/derived for*/fold/derived
 in-range
 in-naturals
 in-list
 in-vector
 in-string
 in-bytes
 in-input-port-bytes
 in-input-port-chars
 in-hash-table
 in-hash-table-keys
 in-hash-table-values
 in-hash-table-pairs
 in-parallel
 stop-before
 stop-after
 in-indexed
 sequence?
 sequence-generate
 {\tt define-sequence-syntax}
 make-do-sequence
 :do-in
```

package: compatibility-lib

# 20 mzlib/foreign

```
(require mzlib/foreign) package: compatibility-lib
```

**NOTE:** This library is deprecated; use ffi/unsafe, instead.

Re-exports scheme/foreign.

### 21 mzlib/include

```
(require mzlib/include) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/include, instead.

Similar to scheme/include, but with a different syntax for paths.

Inlines the syntax in the designated file in place of the include expression. The *path-spec* can be any of the following:

- A literal string that specifies a path to include, parsed according to the platform's conventions (which means that it is not portable).
- A path construction of the form (build-path elem ...+), where build-path is module-identifier=? either to the build-path export from mzscheme or to the top-level build-path, and where each elem is a path string, up (unquoted), or same (unquoted). The elems are combined in the same way as for the build-path function to obtain the path to include.
- A path construction of the form (lib file-string collection-string ...), where lib is free or refers to a top-level lib variable. The collection-strings are passed to collection-path to obtain a directory; if no collection-stringss are supplied, "mzlib" is used. The file-string is then appended to the directory using build-path to obtain the path to include.

If path-spec specifies a relative path to include, the path is resolved relative to the source for the include expression, if that source is a complete path string. If the source is not a complete path string, then path-spec is resolved relative to the current load relative directory if one is available, or to the current directory otherwise.

The included syntax is given the lexical context of the include expression.

```
(include-at/relative-to context source path-spec)
(include-at/relative-to/reader context source path-spec reader-expr)
(include/reader path-spec reader-expr)
```

Variants of include analogous to the variants of scheme/include.

## 22 mzlib/inflate

```
(require mzlib/inflate) package: compatibility-lib
```

**NOTE:** This library is deprecated; use file/gunzip, instead.

Re-exports file/gunzip.

## 23 mzlib/integer-set

```
(require mzlib/integer-set) package: compatibility-lib
```

**NOTE:** This library is deprecated; use data/integer-set, instead.

The mzlib/integer-set library re-exports bindings from data/integer-set except that it renames symmetric-difference to xor, subtract to difference, and count to card.

### 24 mzlib/kw

```
(require mzlib/kw) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/base, instead. The Racket base language supports keyword arguments. Using the built-in keyword arguments in Racket is highly recommended.

```
(lambda/kw kw-formals body ...+)
(define/kw (head args) body ...+)
   kw-formals = id
               | (id ... [#:optional optional-spec ...]
                         [#:key key-spec ...]
                         [rest/mode-spec ...])
               (id ... id)
 optional-spec = id
               (id default-expr)
     key-spec = id
               (id default-expr)
               (id keyword default-expr)
rest/mode-spec = #:rest id
                #:other-keys id
                #:other-keys+body id
                #:all-keys id
                #:body kw-formals
                #:allow-other-keys
                #:forbid-other-keys
                #:allow-duplicate-keys
                #:forbid-duplicate-keys
                #:allow-body
                #:forbid-body
                 #:allow-anything
                #:forbid-anything
         head = id
               (head . kw-formals)
```

The lambda and procedure-application forms of scheme/base support keyword arguments, and it is not compatible with the mzlib/kw library.

Like lambda, but with optional and keyword-based argument processing. This form is similar to an extended version of Common Lisp procedure arguments (but note the differences below). When used with plain variable names, lambda/kw expands to a plain lambda, so

lambda/kw is suitable for a language module that will use it to replace lambda. Also, when used with only optionals, the resulting procedure is similar to opt-lambda (but a bit faster).

In addition to lambda/kw, define/kw is similar to define, except that the formals are as in lambda/kw. Like define, this form can be used with nested parenthesis for curried functions (the MIT-style generalization of define).

The syntax of lambda/kw is the same as lambda, except for the list of formal argument specifications. These specifications can hold (zero or more) plain argument names, then an optionals (and defaults) section that begins after an #:optional marker, then a keyword section that is marked by #:keyword, and finally a section holding rest and "rest"-like arguments which are described below, together with argument processing flag directives. Each section is optional, but the order of the sections must be as listed. Of course, all binding ids must be unique.

The following sections describe each part of the kw-formals.

### 24.1 Required Arguments

Required arguments correspond to ids that appear before any keyword marker in the argument list. They determine the minimum arity of the resulting procedure.

### 24.2 Optional Arguments

The optional-arguments section follows an #:optional marker in the kw-formals. Each optional argument can take the form of a parenthesized variable and a default expression; the latter is used if a value is not given at the call site. The default expression can be omitted (along with the parentheses), in which case #f is the default.

The default expression's environment includes all previous arguments, both required and optional names. With k optionals after n required arguments, and with no keyword arguments or rest-like arguments, the resulting procedure accept between n and n+k arguments, inclusive.

The treatment of optionals is efficient, with an important caveat: default expressions appear multiple times in the resulting case-lambda. For example, the default expression for the last optional argument appears k-1 times (but no expression is ever evaluated more than once in a procedure call). This expansion risks exponential blow-up is if lambda/kw is used in a default expression of a lambda/kw, etc. The bottom line, however, is that lambda/kw is a sensible choice, due to its enhanced efficiency, even when you need only optional arguments.

Using both optional and keyword arguments is possible, but note that the resulting behavior differs from traditional keyword facilities (including the one in Common Lisp). See the

following section for details.

### 24.3 Keyword Arguments

A keyword argument section is marked by a #:key. If it is used with optional arguments, then the keyword specifications must follow the optional arguments (which mirrors the use in call sites; where optionals are given before keywords).

When a procedure accepts both optional and keyword arguments, the argument-handling convention is slightly different than in traditional keyword-argument facilities: a keyword after required arguments marks the beginning of keyword arguments, no matter how many optional arguments have been provided before the keyword. This convention restricts the procedure's non-keyword optional arguments to non-keyword values, but it also avoids confusion when mixing optional arguments and keywords. For example, when a procedure that takes two optional arguments and a keyword argument #:x is called with #:x 1, then the optional arguments get their default values and the keyword argument is bound to 1. (The traditional behavior would bind #:x and 1 to the two optional arguments.) When the same procedure is called with 1 #:x 2, the first optional argument is bound to 1, the second optional argument is bound to its default, and the keyword argument is bound to 2. (The traditional behavior would report an error, because 2 is provided where #:x is expected.)

Like optional arguments, each keyword argument is specified as a parenthesized variable name and a default expression. The default expression can be omitted (with the parentheses), in which case #f is the default value. The keyword used at a call site for the corresponding variable has the same name as the variable; a third form of keyword arguments has three parts—a variable name, a keyword, and a default expression—to allow the name of the locally bound variable to differ from the keyword used at call sites.

When calling a procedure with keyword arguments, the required argument (and all optional arguments, if specified) must be followed by an even number of arguments, where the first argument is a keyword that determines which variable should get the following value, etc. If the same keyword appears multiple times (and if multiple instances of the keyword are allowed; see §24.6 "Mode Keywords"), the value after the first occurrence is used for the variable:

#### Example:

Default expressions are evaluated only for keyword arguments that do not receive a value for a particular call. Like optional arguments, each default expression is evaluated in an environment that includes all previous bindings (required, optional, and keywords that were specified on its left).

See §24.6 "Mode Keywords" for information on when duplicate or unknown keywords are allowed at a call site.

### 24.4 Rest and Rest-like Arguments

The last *kw-formals* section—after the required, optional, and keyword arguments—may contain specifications for rest-like arguments and/or mode keywords. Up to five rest-like arguments can be declared, each with an *id* to bind:

- #:rest The variable is bound to the list of "rest" arguments, which is the list of all values after the required and the optional values. This list includes all keyword-value pairs, exactly as they are specified at the call site.
  - Scheme's usual dot-notation is accepted in *kw-formals* only if no other meta-keywords are specified, since it is not clear whether it should specify the same binding as a #:rest or as a #:body. The dot notation is allowed without meta-keywords to make the lambda/kw syntax compatible with lambda.
- #:body The variable is bound to all arguments after keyword-value pairs. (This is different from Common Lisp's &body, which is a synonym for &rest.) More generally, a #:body specification can be followed by another kw-formals, not just a single id; see §24.5 "Body Argument" for more information.
- #:all-keys the variable is bound to the list of all keyword-values from the call site, which is always a proper prefix of a #:rest argument. (If no #:body arguments are declared, then #:all-keys binds the same as #:rest.) See also keyword-get.
- #:other-keys The variable is bound like an #:all-keys variable, except that all keywords specified in the kw-formals are removed from the list. When a keyword is used multiple times at a call cite (and this is allowed), only the first instances is removed for the #:other-keys binding.
- #:other-keys+body the variable is bound like a #:rest variable, except that all keywords specified in the kw-formals are removed from the list. When a keyword is used multiple times at a call site (and this is allowed), only the first instance us removed for the #:other-keys+body binding. (When no #:body variables are specified, then #:other-keys+body is the same as #:other-keys.)

In the following example, all rest-like arguments are used and have different bindings:

Example:

Note that the following invariants always hold:

```
rest = (append all-keys body)other-keys+body = (append other-keys body)
```

To write a procedure that uses a few keyword argument values, and that also calls another procedure with the same list of arguments (including all keywords), use #:other-keys (or #:other-keys+body). The Common Lisp approach is to specify :allow-other-keys, so that the second procedure call will not cause an error due to unknown keywords, but the :allow-other-keys approach risks confusing the two layers of keywords.

#### 24.5 Body Argument

The most notable divergence from Common Lisp in lambda/kw is the #:body argument, and the fact that it is possible at a call site to pass plain values after the keyword-value pairs. The #:body binding is useful for procedure calls that use keyword-value pairs as sort of an attribute list before the actual arguments to the procedure. For example, consider a procedure that accepts any number of numeric arguments and will apply a procedure to them, but the procedure can be specified as an optional keyword argument. It is easily implemented with a #:body argument:

#### Examples:

```
> (define/kw (mathop #:key [op +] #:body b)
        (apply op b))
> (mathop 1 2 3)
6
> (mathop #:op max 1 2 3)
3
```

(Note that the first body value cannot itself be a keyword.)

A #:body declaration works as an arbitrary kw-formals, not just a single variable like b in the above example. For example, to make the above mathop work only on three arguments that follow the keyword, use (x y z) instead of b:

#### Example:

In general, #:body handling is compiled to a sub procedure using lambda/kw, so that a procedure can use more then one level of keyword arguments. For example:

#### Examples:

Obviously, nested keyword arguments works only when non-keyword arguments separate the sets.

Run-time errors during such calls report a mismatch for a procedure with a name that is based on the original name plus a ~body suffix:

#### Example:

```
> (mathop #:op * 2 4)
mathop~body: arity mismatch;
the expected number of arguments does not match the given
number
    expected: at least 3
    given: 2
    arguments...:
    2
    4
```

### 24.6 Mode Keywords

Finally, the argument list of a lambda/kw can contain keywords that serve as mode flags to control error reporting.

• #:allow-other-keys — The keyword-value sequence at the call site *can* include keywords that are not listed in the keyword part of the lambda/kw form.

- #:forbid-other-keys The keyword-value sequence at the call site *cannot* include keywords that are not listed in the keyword part of the lambda/kw form, otherwise the exn:fail:contract exception is raised.
- #:allow-duplicate-keys The keyword-value list at the call site *can* include duplicate values associated with same keyword, the first one is used.
- #:forbid-duplicate-keys The keyword-value list at the call site *cannot* include duplicate values for keywords, otherwise the exn:fail:contract exception is raised. This restriction applies only to keywords that are listed in the keyword part of the lambda/kw form if other keys are allowed, this restriction does not apply to them.
- #:allow-body Body arguments *can* be specified at the call site after all keyword-value pairs.
- #:forbid-body Body arguments *cannot* be specified at the call site after all keyword-value pairs.
- #:allow-anything Allows all of the above, and treat a single keyword at the end of an argument list as a #:body, a situation that is usually an error. When this is used and no rest-like arguments are used except #:rest, an extra loop is saved and calling the procedures is faster (around 20%).
- #:forbid-anything Forbids all of the above, ensuring that calls are as restricted as possible.

These above mode markers are rarely needed, because the default modes are determined by the declared rest-like arguments:

- The default is to allow other keys if a #:rest, #:other-keys+body, #:all-keys, or #:other-keys variable is declared (and an #:other-keys declaration requires allowing other keys).
- The default is to allow duplicate keys if a #:rest or #:all-keys variable is declared.
- The default is to allow body arguments if a #:rest, #:body, or #:other-keys+body variable is declared (and a #:body argument requires allowing them).

Here's an alternate specification, which maps rest-like arguments to the behavior that they imply:

- #:rest: Everything is allowed (a body, other keys, and duplicate keys);
- #:other-keys+body: Other keys and body are allowed, but duplicates are not;
- #:all-keys: Other keys and duplicate keys are allowed, but a body is not;

- #:other-keys: Other keys must be allowed (on by default, cannot use with #:forbid-other-keys), and duplicate keys and body are not allowed;
- #:body: Body must be allowed (on by default, cannot use with #:forbid-body) and other keys and duplicate keys and body are not allowed;
- Except for the previous two "must"s, defaults can be overridden by an explicit #:allow-... or a #:forbid-... mode.

### 24.7 Property Lists

```
(keyword-get args kw not-found) → any
  args : (listof (cons/c keyword? any/c))
  kw : keyword?
  not-found : (-> any)
```

Searches a list of keyword arguments (a "property list" or "plist" in Lisp jargon) for the given keyword, and returns the associated value. It is the facility that is used by lambda/kw to search for keyword values.

The args list is scanned from left to right, if the keyword is found, then the next value is returned. If the kw was not found, then the not-found thunk is used to produce a value by applying it. If the kw was not found, and not-found thunk is not given, #f is returned. (No exception is raised if the args list is imbalanced, and the search stops at a non-keyword value.)

### 25 mzlib/list

cons?

```
(require mzlib/list) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/list, instead.

The mzlib/list library re-exports several functions from scheme/base and scheme/list:

```
empty?
empty
foldl
foldr
remv
remq
remove
remv*
remq*
remove*
findf
memf
assf
filter
sort
(first v) \rightarrow any/c
  v : pair?
(second v) \rightarrow any/c
  v : (and/c pair? ....)
(third v) \rightarrow any/c
  v : (and/c pair? ....)
(fourth v) \rightarrow any/c
  v : (and/c pair? ....)
(fifth v) \rightarrow any/c
  v : (and/c pair? ....)
(sixth v) \rightarrow any/c
  v : (and/c pair? ....)
(seventh v) \rightarrow any/c
  v : (and/c pair? ....)
(eighth v) \rightarrow any/c
  v : (and/c pair? ....)
```

Accesses the first, second, etc. elment of "list" v. The argument need not actually be a list; it is inspected only as far as necessary to obtain an element (unlike the same-named functions from scheme/list, which do require the argument to be a list).

```
(rest v) \rightarrow any/c
 v : pair?
```

The same as cdr.

```
(last-pair v) \rightarrow pair?
v : pair?
```

Returns the last pair in v, raising an error if v is not a pair (but v does not have to be a proper list).

```
(merge-sorted-lists lst1 lst2 less-than?) → list?
  lst1 : list?
  lst2 : lst?
  less-than? : (any/c any/c . -> . any/c)
```

Merges the two sorted input lists, creating a new sorted list. The merged result is stable: equal items in both lists stay in the same order, and these in 1st1 precede 1st2.

```
(mergesort lst less-than?) → list?
  lst : list?
  less-than? : (any/c any/c . -> . any/c)
```

The same as sort.

```
(quicksort lst less-than?) → list?
  lst : list?
  less-than? : (any/c any/c . -> . any/c)
```

The same as sort.

### 26 mzlib/match

```
(require mzlib/match) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/match, instead.

The mzlib/match library provides a match form similar to that of racket/match, but with an different (older and less extensible) syntax of patterns.

See match from racket/match for a description of matching. The grammar of pat for this match is as follows:

```
::= id
                                      match anything, bind identifier
pat
                                      match anything
            literal
                                      match literal
          | 'datum
                                      match equal? datum
          | (1vp ...)
                                      match sequence of 1vps
          | (1vp ... pat)
                                      match lvps consed onto a pat
          | #(lvp ...)
                                      match vector of pats
          | #&pat
                                      match boxed pat
          ($ struct-id pat ...) match struct-id instance
            (and pat ...)
                                      match when all pats match
            (or pat ...)
                                      match when any pat match
            (not pat ...)
                                      match when no pat match
             (= expr pat)
                                      match (expr value) to pat
             (? pred-expr pat ...) match if (expr value) and pats
                                      match quasipattern
                                      match true
literal ::= #t
                                      match false
          | #f
                                      match equal? string
          | string
          number
                                      match equal? number
          character
                                      match equal? character
          | bytes
                                      match equal? byte string
          | keyword
                                      match equal? keyword
                                      match equal? regexp literal
          | regexp
                                      match equal? pregexp literal
          | pregexp
                                      greedily match pat instances
lvp
         ::= pat
                                      match pat
          | pat
                                      zero or more; ... is literal
000
         ::= ...
                                      zero or more
          l ___
```

```
..k
                                   k or more
         -_k
                                   k or more
        ::= literal
                                   match literal
qp
         | id
                                   match equal? symbol
         | (qp ...)
                                   match sequences of qps
         | (qp ... qp)
                                   match sequence of qps consed onto a qp
                                   match qps consed onto a repeated qp
         | (qp ... qp ooo)
         | #(qp ...)
                                   match vector of qps
         | #&qp
                                   match boxed qp
         | ,pat
                                   match pat
           ,@pat
                                   match pat, spliced
 (define/match (head args) match*-clause ...)
 (match-lambda clause ...)
 (match-lambda* clause ...)
 (match-let ([pat expr] ...) body ...+)
 (match-let* ([pat expr] ...) body ...+)
 (match-letrec ([pat expr] ...) body ...+)
 (match-define pat expr)
```

Analogous to the combined forms from racket/match.

```
(define-match-expander id proc-expr)
(define-match-expander id proc-expr proc-expr)
(define-match-expander id proc-expr proc-expr proc-expr)
(match-equality-test) → (any/c any/c . -> . any)
(match-equality-test comp-proc) → void?
  comp-proc : (any/c any/c . -> . any)
```

Analogous to the form and parameter from racket/match. The define-match-expander form, however, supports an extra *proc-expr* as the middle one: an expander for use with match from mzlib/match.

## 27 mzlib/math

```
(require mzlib/math) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/math, instead.

Re-exports scheme/math, and also exports e.

```
e : real?
```

An approximation to Euler's constant: 2.718281828459045.

## 28 mzlib/md5

(require mzlib/md5) package: compatibility-lib

**NOTE:** This library is deprecated; use file/md5, instead.

Re-exports file/md5.

### 29 mzlib/os

```
(require mzlib/os) package: compatibility-lib (gethostname) \rightarrow string?
```

Returns a string for the current machine's hostname (including its domain).

```
(getpid) → exact-integer?
```

Returns an integer identifying the current process within the operating system.

```
(truncate-file file [n-bytes]) → void?
  file : path-string?
  n-bytes : exact-nonnegative-integer? = 0
```

Truncates or extends the given file so that it is n-bytes long. If the file does not exist, or if the process does not have sufficient privilege to truncate the file, the exn:fail exception is raised.

The truncate-file function is implemented in terms of racket/base's file-truncate.

## 30 mzlib/pconvert

```
(require mzlib/pconvert) package: pconvert-lib
```

The mzlib/pconvert library defines routines for printing Racket values as evaluable S-expressions. Racket's default printing mode also prints values as expressions (in contrast to the Lisp and Racket tradition of printing readable S-expressions), but mzlib/pconvert is more configurable and approximates expressions for a wider range of values. For example, procedures print using lambda instead of #procedure>.

The print-convert procedure does not print values; rather, it converts a Racket value into another Racket value such that the new value pretty-writes as a Racket expression that evaluates to the original value. For example, (pretty-write (print-convert `(9, (box 5) #(6 7)))) prints the literal expression (list 9 (box 5) (vector 6 7)) to the current output port.

To install print converting into the read-eval-print loop, require mzlib/pconvert and call the procedure install-converting-printer.

In addition to print-convert, this library provides print-convert, build-share, get-shared, and print-convert-expr. The last three are used to convert sub-expressions of a larger expression (potentially with shared structure).

See also prop:print-convert-constructor-name.

```
(abbreviate-cons-as-list) → boolean?
(abbreviate-cons-as-list abbreviate?) → void?
abbreviate? : any/c
```

A parameter that controls how lists are represented with constructor-style conversion. If the parameter's value is #t, lists are represented using list. Otherwise, lists are represented using cons. The initial value of the parameter is #t.

```
(booleans-as-true/false) → boolean?
(booleans-as-true/false use-name?) → void?
  use-name? : any/c
```

A parameter that controls how #t and #f are represented. If the parameter's value is #t, then #t is represented as true and #f is represented as false. The initial value of the parameter is #t.

```
(use-named/undefined-handler) \rightarrow (any/c . -> . any/c) (use-named/undefined-handler use-handler) \rightarrow void? use-handler : (any/c . -> . any/c)
```

A parameter that controls how values that have inferred names are represented. The procedure is passed a value. If the procedure returns true, the procedure associated with

named/undefined-handler is invoked to render that value. Only values that have inferred names but are not defined at the top-level are used with this handler.

The initial value of the parameter is (lambda (x) #f).

```
(named/undefined-handler) \rightarrow (any/c . -> . any/c)
(named/undefined-handler use-handler) \rightarrow void?
use-handler : (any/c . -> . any/c)
```

Parameter for a procedure that controls how values that have inferred names are represented. The procedure is called only if use-named/undefined-handler returns true for some value. In that case, the procedure is passed that same value, and the result of the parameter is used as the representation for the value.

The initial value of the parameter is (lambda (x) #f).

```
(add-make-prefix-to-constructor) → boolean?
(add-make-prefix-to-constructor add-prefix?) → void?
add-prefix? : any/c
```

A parameter that controls whether a make- prefix is added to a constructor name for a structure instance. The initial value of the parameter is #f.

```
(build-share v) \rightarrow \dots
v : any/c
```

Takes a value and computes sharing information used for representing the value as an expression. The return value is an opaque structure that can be passed back into get-shared or print-convert-expr.

```
(constructor-style-printing) → boolean?
(constructor-style-printing use-constructors?) → void?
use-constructors? : any/c
```

Parameter that controls how values are represented after conversion. If this parameter's value is #t, then constructors are used; e.g., pair containing 1 and 2 is represented as (cons 1 2). Otherwise, quasiquote-style syntax is used; e.g., the pair containing 1 and 2 is represented as `(1 . 2). The initial value of the parameter is #f.

The constructor used for mutable pairs is mcons, unless print-mpair-curly-braces is set to #f, in which case cons and list are used. Similarly, when using quasiquote style and print-mpair-curly-braces is set to #f, mutable pair constructions are represented using quote, quasiquote, etc.

See also quasi-read-style-printing and prop:print-convert-constructor-name.

Parameter that sets a procedure used by print-convert and build-share to assemble sharing information. The procedure *hook* takes three arguments: a value v, a procedure basic-share, and a procedure sub-share; the return value is ignored. The basic-share procedure takes v and performs the built-in sharing analysis, while the sub-share procedure takes a component of v and analyzes it. Sharing information is accumulated as values are passed to basic-share and sub-share.

A current-build-share-hook procedure usually works together with a current-print-convert-hook procedure.

```
(current-build-share-name-hook)
  → (any/c . -> . (or/c symbol? false/c))
(current-build-share-name-hook hook) → void?
hook : (any/c . -> . (or/c symbol? false/c))
```

Parameter that sets a procedure used by print-convert and build-share to generate a new name for a shared value. The *hook* procedure takes a single value and returns a symbol for the value's name. If *hook* returns #f, a name is generated using the form "-n-, where n is an integer.

```
(\text{current-print-convert-hook}) \rightarrow (\text{any/c (any/c .} -> . \text{ any/c}) \\ (\text{any/c .} -> . \text{ any/c}) \\ . -> . \text{ any/c}) \\ (\text{current-print-convert-hook hook}) \rightarrow \text{void?} \\ \text{hook : (any/c (any/c .} -> . \text{any/c}) \\ (\text{any/c .} -> . \text{any/c}) \\ . -> . \text{any/c})
```

Parameter that sets a procedure used by print-convert and print-convert-expr to convert values. The procedure *hook* takes three arguments—a value v, a procedure *basic-convert*, and a procedure sub-convert—and returns the converted representation of v. The basic-convert procedure takes v and returns the default conversion, while the sub-convert procedure takes a component of v and returns its conversion.

A current-print-convert-hook procedure usually works together with a current-build-share-hook procedure.

```
(current-read-eval-convert-print-prompt) → string?
```

```
(current-read-eval-convert-print-prompt str) \rightarrow void? str : string?
```

Parameter that sets the prompt used by install-converting-printer. The initial value is " | - ".

```
(get-shared share-info [cycles-only?])
  → (list-of (cons/c symbol? any/c))
  share-info : ....
  cycles-only? : any/c = #f
```

The shared-info value must be a result from build-share. The procedure returns a list matching variables to shared values within the value passed to build-share.

The default value for *cycles-only*? is #f; if it is not #f, get-shared returns only information about cycles.

For example,

might return the list

```
'((-1- (cons 1 -2-)) (-2- (cons 2 -1-)))

(install-converting-printer) \rightarrow void?
```

Sets the current print handler to print values using print-convert and sets print-asexpression to #f (since the conversion of a value is meant to be printed in readable form rather than evaluable form). The current read handler is also set to use the prompt returned by current-read-eval-convert-print-prompt.

```
(print-convert v [cycles-only?]) → any/c
  v : any/c
  cycles-only? : any/c = (show-sharing)
```

Converts the value v. If cycles-only? is not #f, then only circular objects are included in the output.

Converts the value v using sharing information share-info, which was previously returned by build-share for a value containing v. If the most recent call to get-shared with share-info requested information only for cycles, then print-convert-expr will only display sharing among values for cycles, rather than showing all value sharing.

The unroll-once? argument is used if v is a shared value in share-info. In this case, if unroll-once? is #f, then the return value will be a shared-value identifier; otherwise, the returned value shows the internal structure of v (using shared value identifiers within v's immediate structure as appropriate).

```
(quasi-read-style-printing) → boolean?
(quasi-read-style-printing on?) → void?
on?: any/c
```

Parameter that controls how vectors and boxes are represented after conversion when the value of constructor-style-printing is #f. If quasi-read-style-printing is set to #f, then boxes and vectors are unquoted and represented using constructors. For example, the list of a box containing the number 1 and a vector containing the number 1 is represented as `(,(box 1),(vector 1)). If the parameter's value is #t, then #&... and #(...) are used, e.g., `(#&1 #(1)). The initial value of the parameter is #t.

```
(show-sharing) → boolean?
(show-sharing show?) → void?
show? : any/c
```

Parameter that determines whether sub-value sharing is conserved (and shown) in the converted output by default. The initial value of the parameter is #t.

```
(whole/fractional-exact-numbers) → boolean?
(whole/fractional-exact-numbers whole-frac?) → void?
whole-frac? : any/c
```

Parameter that controls how exact, non-integer numbers are converted when the numerator is greater than the denominator. If the parameter's value is #t, the number is converted to the form (+ integer fraction) (i.e., a list containing '+, an exact integer, and an exact rational less than 1 and greater than -1). The initial value of the parameter is #f.

## 31 mzlib/pconvert-prop

```
(require mzlib/pconvert-prop) package: pconvert-lib
prop:print-converter : property?
(print-converter? v) → any
  v : any/c
(print-converter-proc v)
  → (any/c (any/c . -> . any/c) . -> . any/c)
  v : print-converter?
```

The prop:print-converter property can be given a procedure value for a structure type. In that case, for constructor-style print conversion via print-convert, instances of the structure are converted by calling the procedure that is the property's value. The procedure is called with the value to convert and a procedure to recursively convert nested values. The result should be an S-expression for the converted value.

The print-converter? predicate recognizes instances of structure types that have the prop:print-converter property, and print-converter-proc extracts the property value.

```
prop:print-convert-constructor-name : property?
(print-convert-named-constructor? v) → any
  v : any/c
(print-convert-constructor-name v) → any
  v : print-convert-named-constructor?
```

The prop:print-convert-constructor-name property can be given a symbol value for a structure type. In that case, for constructor-style print conversion via print-convert, instances of the structure are shown using the symbol as the constructor name.

The prop:print-converter property takes precedence over prop:print-convert-constructor-name. If neither is attached to a structure type, its instances are converted using a constructor name that is make-prefixed onto the result of object-name.

The print-convert-named-constructor? predicate recognizes instances of structure types that have the prop:print-convert-constructor-name property, and print-convert-constructor-name extracts the property value.

## 32 mzlib/plt-match

```
(require mzlib/plt-match) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/match, instead.

The mzlib/plt-match library mostly re-provides scheme/match.

```
(define-match-expander id proc-expr)
(define-match-expander id proc-expr proc-expr)
(define-match-expander id proc-expr proc-expr proc-expr)
```

The same as the form from mzlib/match.

## 33 mzlib/port

```
(require mzlib/port) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/port, instead.

The mzlib/port library mostly re-provides racket/port.

```
(strip-shell-command-start in) → void?
  in : input-port?
```

Reads and discards a leading #! in in (plus continuing lines if the line ends with a back-slash). Since #! followed by a forward slash or space is a comment, this procedure is not needed before reading Scheme expressions.

## 34 mzlib/pregexp

```
(require mzlib/pregexp) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/base, instead.

The mzlib/pregexp library provides wrappers around regexp-match, etc. that coerce string and byte-string arguments to pregexp matchers instead of regexp matchers.

The library also re-exports: pregexp, and it re-exports regexp-quote as pregexp-quote.

```
(pregexp-match pattern
               input
              [start-pos
               end-pos
               output-port])
 → (or/c (listof (or/c (cons (or/c string? bytes?))
                              (or/c string? bytes?))
                       false/c))
         false/c)
 pattern : (or/c string? bytes? regexp? byte-regexp?)
 input : (or/c string? bytes? input-port?)
 start-pos : exact-nonnegative-integer? = 0
 end-pos : (or/c exact-nonnegative-integer? false/c) = #f
 output-port : (or/c output-port? false/c) = #f
(pregexp-match-positions pattern
                         input
                         [start-pos
                         end-pos
                         output-port])
 → (or/c (listof (or/c (cons exact-nonnegative-integer?
                              exact-nonnegative-integer?)
                       false/c))
         false/c)
 pattern : (or/c string? bytes? regexp? byte-regexp?)
 input : (or/c string? bytes? input-port?)
 start-pos : exact-nonnegative-integer? = 0
 end-pos : (or/c exact-nonnegative-integer? false/c) = #f
 output-port : (or/c output-port? false/c) = #f
```

```
(pregexp-split pattern
               input
              [start-pos
               end-pos]) → (listof (or/c string? bytes?))
 pattern : (or/c string? bytes? regexp? byte-regexp?)
 input : (or/c string? bytes? input-port?)
 start-pos : exact-nonnegative-integer? = 0
 end-pos : (or/c exact-nonnegative-integer? false/c) = #f
(pregexp-replace pattern input insert) \rightarrow (or/c string? bytes?)
 pattern : (or/c string? bytes? regexp? byte-regexp?)
 input : (or/c string? bytes?)
 insert : (or/c string? bytes?
                (string? . -> . string?)
                (bytes? . -> . bytes?))
(pregexp-replace* pattern input insert) → (or/c string? bytes?)
 pattern : (or/c string? bytes? regexp? byte-regexp?)
 input : (or/c string? bytes?)
 insert : (or/c string? bytes?
                (string? . -> . string?)
                (bytes? . -> . bytes?))
```

Like regexp-match, etc., but a string pattern argument is compiled via pregexp, and a byte string pattern argument is compiled via byte-pregexp.

# 35 mzlib/pretty

```
(require mzlib/pretty) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/pretty, instead.

Re-exports scheme/pretty.

# 36 mzlib/process

```
(require mzlib/process) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/system, instead.

Re-exports scheme/system.

### 37 mzlib/restart

```
(require mzlib/restart) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/sandbox, instead. The racket/sandbox library provides a more general way to simulate running a new Racket process.

Simulates starting Racket with the vector of command-line strings argv. The init-argv, adjust-flag-table, and init-namespace arguments are used to modify the default settings for command-line flags, adjust the parsing of command-line flags, and customize the initial namespace, respectively.

The vector of strings <code>init-argv</code> is read first with the standard Racket command-line parsing. Flags that load files or evaluate expressions (e.g., <code>-f</code> and <code>-e</code>) are ignored, but flags that set Racket's modes (e.g., <code>-c</code> or <code>-j</code>) effectively set the default mode before <code>argv</code> is parsed.

Before argv is parsed, the procedure adjust-flag-table is called with a command-line flag table as accepted by parse-command-line. The return value must also be a table of command-line flags, and this table is used to parse argv. The intent is to allow adjust-flag-table to add or remove flags from the standard set.

After *argv* is parsed, a new thread and a namespace are created for the "restarted" Racket. (The new namespace is installed as the current namespace in the new thread.) In the new thread, restarting performs the following actions:

- The *init-namespace* procedure is called with no arguments. The return value is ignored.
- Expressions and files specified by *argv* are evaluated and loaded. If an error occurs, the remaining expressions and files are ignored, and the return value for restart-mzscheme is set to #f.
- The read-eval-print-loop procedure is called, unless a flag in *init-argv* or *argv* disables it. When read-eval-print-loop returns, the return value for restart-mzscheme is set to #t.

Before evaluating command-line arguments, an exit handler is installed that immediately returns from restart-mzscheme with the value supplied to the handler. This exit handler remains in effect when read-eval-print-loop is called (unless a command-line argument changes it). If restart-mzscheme returns normally, the return value is determined as described above.

Note that an error in a command-line expression followed by read-eval-print-loop produces a #t result. This is consistent with Racket's stand-alone behavior.

# 38 mzlib/runtime-path

```
(require mzlib/runtime-path) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/runtime-path, instead.

Re-exports scheme/runtime-path.

### 39 mzlib/sandbox

```
(require mzlib/sandbox) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/sandbox, instead.

The mzlib/sandbox library mostly re-exports racket/sandbox, but it provides a slightly different make-evaluator function.

The library re-exports the following bindings:

```
sandbox-init-hook
sandbox-reader
sandbox-input
sandbox-output
sandbox-error-output
sandbox-propagate-breaks
sandbox-coverage-enabled
sandbox-namespace-specs
sandbox-override-collection-paths
sandbox-security-guard
sandbox-path-permissions
sandbox-network-guard
sandbox-make-inspector
sandbox-eval-limits
kill-evaluator
break-evaluator
set-eval-limits
put-input
get-output
get-error-output
get-uncovered-expressions
call-with-limits
with-limits
exn:fail:resource?
exn:fail:resource-resource
(make-evaluator language
                requires
                input-program ...) \rightarrow (any/c . -> . any)
 language : (or/c module-path?
                   (one-of/c 'r5rs 'beginner 'beginner-abbr
                             'intermediate 'intermediate-lambda 'advanced)
                   (list/c (one-of/c 'special) symbol?)
                   (list/c (one-of/c 'special) symbol?)
                   (cons/c (one-of/c 'begin) list?))
```

Like make-evaluator or make-module-evaluator, but with several differences:

- The language argument can be one of a fixed set of symbols: 'r5rs, etc. They are converted by adding a (list 'special ....) wrapper.
- If requires starts with 'begin, then each element in the remainder of the list is effectively evaluated as a prefix to the program. Otherwise, it corresponds to the #:requires argument of make-evaluator.
- For each of *language* and *requires* that starts with 'begin, the expressions are inspected to find top-level require forms (using symbolic equality to detect require), and the required modules are added to the #:allow list for make-evaluator.

### 40 mzlib/sendevent

```
(require mzlib/sendevent) package: compatibility-lib
```

The mzlib/sendevent library provides a send-event function that works only on Mac OS, and only when running in GRacket (though the library can be loaded in Racket).

Calls send-event scheme/gui/base, if available, otherwise raises exn:fail:unsupported.

#### 41 mzlib/serialize

```
(require mzlib/serialize) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/serialize, instead.

The mzlib/serialize library provides the same bindings as racket/serialize, except that define-serializable-struct and define-serializable-struct/versions are based on the syntax of define-struct from mzscheme.

Like define-serializable-struct and define-serializable-struct/versions, but with the syntax of closer to define-struct of mzscheme.

## 42 mzlib/shared

(require mzlib/shared) package: compatibility-lib

**NOTE:** This library is deprecated; use racket/shared, instead.

Re-exports scheme/shared.

### 43 mzlib/string

```
(require mzlib/string) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/base, instead. Also see racket/string

The mzlib/string library re-exports several functions from scheme/base:

```
real->decimal-string
regexp-quote
regexp-replace-quote
regexp-match*
regexp-match-positions*
regexp-match-peek-positions*
regexp-split
regexp-match-exact?
```

It also re-exports regexp-try-match as regexp-match/fail-without-reading.

Produces a regexp for a an input "glob pattern" str. A glob pattern is one that matches \* with any string, ? with a single character, and character ranges are the same as in regexps (unless simple? is true). In addition, the resulting regexp does not match strings that begin with ..., unless str begins with ... or hide-dots? is #f. The resulting regexp can be used with string file names to check the glob pattern. If the glob pattern is provided as a byte string, the result is a byte regexp.

The case-sensitive? argument determines whether the resulting regexp is case-sensitive.

If simple? is true, then ranges with [...] in str are treated as literal character sequences.

```
(string-lowercase! str) \rightarrow void?

str: (and/c string? (not/c immutable?))
```

Destructively changes str to contain only lowercase characters.

```
(string-uppercase! str) → void?
  str : (and/c string? (not/c immutable?))
```

Destructively changes str to contain only uppercase characters.

Reads and evaluates S-expressions from str, returning results for all of the expressions in the string. If any expression produces multiple results, the results are spliced into the resulting list. If str contains only whitespace and comments, an empty list is returned, and if str contains multiple expressions, the result will be contain multiple values from all subexpressions.

The err-handler argument can be:

- #f (the default) which means that errors are not caught;
- a one-argument procedure, which will be used with an exception (when an error occurs) and its result will be returned
- a thunk, which will be used to produce a result.

```
(expr->string expr) → string?
expr : any/c
```

Prints expr into a string and returns the string.

Reads the first S-expression from str and returns it. The err-handler is as in evalstring.

```
(read-from-string-all str [err-handler]) → list?
str : (or/c string? bytes?)
err-handler : (or/c false/c = #f
(any/c . -> . any/c)
(-> any/c))
```

Reads all S-expressions from the string (or byte string) str and returns them in a list. The err-handler is as in eval-string.

#### 44 mzlib/struct

"Functional update" for structure instances. The result of evaluating <code>struct-expr</code> must be an instance of the structure type named by <code>struct-id</code>. The result of the copy-struct expression is a fresh instance of <code>struct-id</code> with the same field values as the result of <code>struct-expr</code>, except that the value for the field accessed by each <code>accessor-id</code> is replaced by the result of <code>field-expr</code>.

The result of struct-expr might be an instance of a sub-type of struct-id, but the result of the copy-struct expression is an immediate instance of struct-id. If struct-expr does not produce an instance of struct-id, the exn:fail:contract exception is raised.

If any accessor-id is not bound to an accessor of struct-id (according to the expansion-time information associated with struct-id), or if the same accessor-id is used twice, then a syntax error is raised.

Like define-struct from mzscheme, but properties can be attached to the structure type. Each *prop-expr* should produce a structure-type property value, and each *val-expr* produces the corresponding value for the property.

#### Examples:

Builds a function that accepts a structure type instance (matching struct-id) and provides a vector of the fields of the structure type instance.

## 45 mzlib/stxparam

(require mzlib/stxparam) package: compatibility-lib

**NOTE:** This library is deprecated; use racket/stxparam, instead. Also see racket/stxparam-exptime.

Re-exports scheme/stxparam and scheme/stxparam-exptime (both at phase level 0).

# 46 mzlib/surrogate

```
(require mzlib/surrogate) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/surrogate, instead.

Re-exports scheme/surrogate.

## 47 mzlib/tar

```
(require mzlib/tar) package: compatibility-lib
```

**NOTE:** This library is deprecated; use file/tar, instead.

Re-exports file/tar.

#### 48 mzlib/thread

```
(require mzlib/thread) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/engine, instead.

Re-exports the bindings from racket/engine under different names and also provides two extra bindings. The renamings are:

- engine as coroutine
- engine? as coroutine?
- engine-run as coroutine-run
- engine-result as coroutine-result
- engine-kill as coroutine-kill

```
(consumer-thread f [init]) → thread? procedure?
  f : procedure?
  init : (-> any) = void
```

Returns two values: a thread descriptor for a new thread, and a procedure with the same arity as f.

When the returned procedure is applied, its arguments are queued to be passed on to f, and #<void> is immediately returned. The thread created by consumer-thread dequeues arguments and applies f to them, removing a new set of arguments from the queue only when the previous application of f has completed; if f escapes from a normal return (via an exception or a continuation), the f-applying thread terminates.

The *init* argument is a procedure of no arguments; if it is provided, *init* is called in the new thread immediately after the thread is created.

Executes a TCP server on the port indicated by *port-no*. When a connection is made by a client, conn is called with two values: an input port to receive from the client, and an output port to send to the client.

Each client connection is managed by a new custodian, and each call to conn occurs in a new thread (managed by the connection's custodian). If the thread executing conn terminates for any reason (e.g., conn returns), the connection's custodian is shut down. Consequently, conn need not close the ports provided to it. Breaks are enabled in the connection thread if breaks are enabled when run-server is called.

To facilitate capturing a continuation in one connection thread and invoking it in another, the parameterization of the run-server call is used for every call to handler. In this parameterization and for the connection's thread, the current-custodian parameter is assigned to the connection's custodian.

If *conn-timeout* is not #f, then it must be a non-negative number specifying the time in seconds that a connection thread is allowed to run before it is sent a break signal. Then, if the thread runs longer than (\* *conn-timeout* 2) seconds, then the connection's custodian is shut down. If *conn-timeout* is #f, a connection thread can run indefinitely.

If *handler* is provided, it is passed exceptions related to connections (i.e., exceptions not caught by *conn-proc*, or exceptions that occur when trying to accept a connection). The default handler ignores the exception and returns #<void>.

The run-server function uses listen, close, accept and accept/break in the same way as it might use tcp-listen, tcp-close, tcp-accept, and tcp-accept/enable-break to accept connections. Provide alternate procedures to use an alternate communication protocol (such as SSL) or to supply optional arguments in the use of tcp-listen. The listener? part of the contract indicates that the procedures must all work on the same kind of listener value.

The run-server procedure loops to serve client connections, so it never returns. If a break occurs, the loop will cleanly shut down the server, but it will not terminate active connections.

## 49 mzlib/trace

(require mzlib/trace) package: compatibility-lib

**NOTE:** This library is deprecated; use racket/trace, instead.

Re-exports racket/trace.

#### 50 mzlib/traceld

(require mzlib/traceld) package: compatibility-lib

The mzlib/traceld library does not provide any bindings. Instead, mzlib/traceld is required for its side-effects.

The mzlib/traceld library installs a new load handler (see current-load) and load-extension handler (see current-load-extension) to print information about the files that are loaded. These handlers chain to the current handlers to perform the actual loads. Trace output is printed to the port that is the current error port (see current-error-port) when the library is instantiated.

Before a file is loaded, the tracer prints the file name and "time" (as reported by the procedure current-process-milliseconds) when the load starts. Trace information for nested loads is printed with indentation. After the file is loaded, the file name is printed with the "time" that the load completed.

## 51 mzlib/trait

(require mzlib/trait) package: compatibility-lib

**NOTE:** This library is deprecated; use racket/trait, instead.

Re-exports scheme/trait.

### 52 mzlib/transcr

```
(require mzlib/transcr) package: compatibility-lib
```

The transcript-on and transcript-off procedures of mzscheme always raise exn:fail:unsupported. The mzlib/transcr library provides working versions of transcript-on and transcript-off.

```
(transcript-on filename) → any
  filename : any/c
(transcript-off) → any
```

Starts/stops recording a transcript at filename.

#### 53 mzlib/unit

```
(require mzlib/unit) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/unit, instead.

The mzlib/unit library mostly re-provides racket/unit, except for struct and struct/ctc from racket/unit.

A signature form like struct from racket/base, but with a different syntax for options that limit exports.

A signature form like struct/ctc from racket/unit, but with a different syntax for the options that limit exports.

```
struct~r
struct~r/ctc
```

The same as struct from racket/base and struct/ctc from racket/unit.

```
struct~s
struct~s/ctc
```

Like  $struct \sim r$  and  $struct \sim r/ctc$ , but the constructor is named the same as the type, instead of with make- prefix.

# 54 mzlib/unit-exptime

```
(require mzlib/unit-exptime) package: compatibility-lib
```

**NOTE:** This library is deprecated; use racket/unit-exptime, instead.

Re-exports scheme/unit-exptime.

### 55 mzlib/unit200

(require mzlib/unit200) package: compatibility-lib

**NOTE:** This library is deprecated; use racket/unit, instead.

The mzlib/unit200 library provides an old implementation of units. See archived version 360 documentation on the "unit.ss" library of the "mzlib" collection for information about this library.

## **56** mzlib/unitsig200

(require mzlib/unitsig200) package: compatibility-lib

**NOTE:** This library is deprecated; use racket/unit, instead.

The mzlib/unit200 library provides an old implementation of units. See archived version 360 documentation on the "unitsig.ss" library of the "mzlib" collection for information about this library.

# 57 mzlib/zip

```
(require mzlib/zip) package: compatibility-lib
```

**NOTE:** This library is deprecated; use file/zip, instead.

Re-exports file/zip.

# Bibliography

[Shivers06] Olin Shivers, Brian D. Carlstrom, Martin Gasbichler, and Mike Sperber, "Scsh Reference Manual." 2006. [Reppy99] John H. Reppy, "Concurrent Programming in ML." 1999.

Index	call-with-input-file*, 32
	call-with-output-file*, 32
#:all-keys, 43	case->, 22
#:allow-anything, 46	channel, 14
#:allow-body, 46	channel-recv-evt, 14
#:allow-duplicate-keys,46	channel-send-evt, 14
#:allow-other-keys, 45	command-line, 13
#:body, 43	constructor-style-printing, 56
#:forbid-anything, 46	consumer-thread, 82
#:forbid-body, 46	copy-struct, 78
#:forbid-duplicate-keys,46	current-build-share-hook, 57
#:forbid-other-keys, 46	current-build-share-name-hook, 57
#:key, 42	current-print-convert-hook, 57
#:optional,41	current-read-eval-convert-print-
#:rest, 43	prompt, 57
->, 19	current-time, 14
->*, 20	define-match-expander, 61
->d, 21	define-match-expander, 51
->d*, 21	define-serializable-struct,73
->pp, 22	define-serializable-
->pp-rest, 22	struct/versions, 73
->r, 21	define-struct/properties, 78
1+, 15	define-structure, 16
1-, 15	define-syntax-set, 27
<=?, 15	define/contract, 18
, 15</th <th><math>{\tt define/kw}, 40</math></th>	${\tt define/kw}, 40$
=?, 15	define/match, 51
>=?, 15	e, 52
>?, 15	eighth, 48
abbreviate-cons-as-list, 55	eval-string,76
add-make-prefix-to-constructor, 56	evcase, 28
atom?, 15	expr->string, 76
awk, 8	fifth, 48
begin-lifted, 27	first, 48
begin-with-definitions, 27	flush-output-port, 15
Body Argument, 44	fourth, 48
booleans-as-true/false, 55	gentmp, 15
box/c, 19	get-shared, 58
build-absolute-path, 33	gethostname, 54
build-flat-contract, 19	getpid, 54
build-relative-path, 33	getprop, 16
build-share, 56	glob->regexp, 75

	/- 24
hash-table, 31	mzlib/date, 24
identity, 28	mzlib/deflate, 25
include, 36	mzlib/defmacro, 26
include-at/relative-to, 36	mzlib/etc, 27
include-at/relative-to/reader, 36	mzlib/file, 32
include/reader, 36	mzlib/for, 34
install-converting-printer, 58	mzlib/foreign, 35
Keyword Arguments, 42	mzlib/include, 36
keyword-get, 47	mzlib/inflate, 38
lambda/kw, 40	mzlib/integer-set, 39
last-pair, 49	mzlib/kw, 40
let+, 28	mzlib/list, 48
loop-until, 29	mzlib/match, 50
make>vector, 78	mzlib/math, 52
make-evaluator, 70	mzlib/md5, 53
match, 50	mzlib/os, 54
match-define, 51	mzlib/pconvert, 55
match-equality-test, 51	mzlib/pconvert-prop, 60
match-lambda, 51	mzlib/plt-match, 61
match-lambda*, 51	mzlib/port, 62
match-let, 51	mzlib/pregexp, 63
match-let*, 51	mzlib/pretty,65
match-letrec, 51	mzlib/process, 66
match:end, 9	mzlib/restart,67
match:start,9	mzlib/runtime-path, 69
match:substring,9	mzlib/sandbox, 70
merge-sorted-lists, 49	mzlib/sendevent,72
mergesort, 49	mzlib/serialize,73
Mode Keywords, 45	mzlib/shared,74
mzlib/a-signature, 5	mzlib/string,75
mzlib/a-unit,6	mzlib/struct, 78
mzlib/async-channel,7	mzlib/stxparam,79
mzlib/awk, 8	mzlib/surrogate, 80
mzlib/class, 10	mzlib/tar,81
mzlib/cm, 11	mzlib/thread, 82
mzlib/cm-accomplice, 12	mzlib/trace, 84
mzlib/cmdline, 13	mzlib/traceld, 85
mzlib/cml, 14	mzlib/trait,86
mzlib/compat, 15	mzlib/transcr, 87
mzlib/compile, 17	mzlib/unit,88
mzlib/contract, 18	mzlib/unit-exptime, 89
mzlib/control, 23	mzlib/unit200,90

```
mzlib/unitsig200,91
                                       run-server, 82
                                       second, 48
mzlib/zip, 92
MzLib: Legacy Libraries, 1
                                       send-event, 72
named/undefined-handler, 56
                                       seventh, 48
namespace-defined?, 29
                                       show-sharing, 59
nand, 29
                                       sixth, 48
new-cafe, 16
                                       spawn, 14
nor, 30
                                       string-lowercase!,75
object-contract, 22
                                       string-uppercase!,76
opt->, 20
                                       strip-shell-command-start, 62
opt->*, 20
                                       struct, 88
opt-lambda, 30
                                       struct/c, 19
                                       struct/ctc, 88
Optional Arguments, 41
pregexp-match, 63
                                       struct~r,88
                                       struct~r/ctc, 88
pregexp-match-positions, 63
pregexp-replace, 64
                                       struct \sim s, 88
pregexp-replace*, 64
                                       struct~s/ctc, 88
pregexp-split, 64
                                       third, 48
print-convert, 58
                                       this-expression-file-name, 31
print-convert-constructor-name, 60
                                       this-expression-source-directory,
print-convert-expr, 58
                                       thread-done-evt, 14
print-convert-named-constructor?,
                                       time-evt, 14
print-converter-proc, 60
                                       transcript-off, 87
print-converter?, 60
                                       transcript-on, 87
prop:print-convert-constructor-
                                       truncate-file, 54
 name, 60
                                       use-named/undefined-handler, 55
prop:print-converter, 60
                                       vector/c, 19
Property Lists, 47
                                       vectorof, 19
putprop, 16
                                       whole/fractional-exact-numbers, 59
quasi-read-style-printing, 59
quicksort, 49
read-from-string, 76
read-from-string-all, 76
real-time, 15
rec, 30
recur, 30
regexp-exec, 9
Required Arguments, 41
rest, 49
Rest and Rest-like Arguments, 43
restart-mzscheme, 67
```