MzLib: Legacy Libraries

Version 5.2.1

February 2, 2012

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

```
(require mzlib/a-signature)
```

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)
```

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)
```

Re-exports scheme/async-channel.

4 mzlib/awk

```
(require mzlib/awk)
```

```
(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)
```

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

6 mzlib/class100

```
(require mzlib/class100)
```

The class100 and class100* forms provide a syntax close to that of class and class* in Racket versions 100 through 103, but with the semantics of the current scheme/class-based class system. For a class defined with class100, keyword-based initialization arguments can be propagated to the superclass, but by-position arguments are not (i.e., the expansion of class100 to class always includes an init-rest clause).

The class100 form uses keywords (e.g., public) that are defined by the mzlib/class library, so typically scheme/class must be imported into any context that imports mzlib/class100.

```
(class100* superclass-expr (interface-expr ...) init-ids
  class100-clause
  ...)
```

```
          init-ids = id
                   | (id ... id-with-default ...)
                   | (id ... id-with-default ... . id)

    id-with-default = (id default-expr)

    class100-clause = (sequence expr ...)
                    | (public public-method-decl ...)
                    | (override public-method-decl ...)
                    | (augment public-method-decl ...)
                    | (pubment public-method-decl ...)
                    | (overment public-method-decl ...)
                    | (augride public-method-decl ...)
                    | (private private-method-decl ...)
                    | (private-field private-var-decl ...)
                    | (inherit inherit-method-decl ...)
                    | (rename rename-method-decl ...)

 public-method-decl = ((internal-id external-id) method-procedure)
                    | (id method-procedure)

private-method-decl = (id method-procedure)

   private-var-decl = (id initial-value-expr)
                    | (id)
                    | id

inherit-method-decl = id
                    | (internal-instance-id external-inherited-id)

 rename-method-decl = (internal-id external-id)
```

```
(class100 superclass-expr init-ids
    class100-clause
    ...)
```

Like `class100*`, but without `interface-expr`s.

```
(class100-asi superclass instance-id-clause ...)
```

Like `class100`, but all initialization arguments are automatically passed on to the superclass initialization procedure by position.

```
(class100*-asi superclass interfaces instance-id-clause ...)
```

Like `class100*`, but all initialization arguments are automatically passed on to the superclass initialization procedure by position.

```
(super-init init-arg-expr ...)
```

An alias for `super-make-object`.

7 mzlib/cm

```
(require mzlib/cm)
```

Re-exports compiler/cm.

8 mzlib/cm-accomplice

```
(require mzlib/cm-accomplice)
```

Re-exports compiler/cm-accomplice.

9 mzlib/cmdline

```
(require mzlib/cmdline)
```

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 re-exported 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.

10 mzlib/cml

```
(require mzlib/cml)
```

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) → thread?
  thunk : (-> any)
```

Equivalent to (thread/suspend-to-kill thunk).

```
(channel) → channel?
```

Equivalent to (make-channel).

```
(channel-recv-evt ch) → evt?
  ch : channel?
```

Equivalent to ch.

```
(channel-send-evt ch v) → evt?
  ch : channel?
  v : any/c
```

Equivalent to (channel-put-evt ch v).

```
(thread-done-evt thd) → any
  thd : thread?
```

Equivalent to (thread-dead-evt thread).

```
(current-time) → real?
```

Equivalent to (current-inexact-milliseconds).

```
(time-evt tm) → evt?
  tm : real?
```

Equivalent to (alarm-evt tm).

11 mzlib/compat

```
(require mzlib/compat)
```

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.

```
(atom? v) → any
  v : any/c
```

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

```
(define-structure (name-id field-id ...))
(define-structure (name-id field-id ...)
                  ((init-field-id init-expr) ...))
```

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

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

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`.

```
(new-cafe [eval-handler]) → any
  eval-handler : (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`.

12 mzlib/compile

```
(require mzlib/compile)
```

```
(compile-file src [dest filter]) → path?
  src : path-string?
  dest : path-string?
         (let-values ([(base name dir?) (split-path src)])
       =   (build-path base "compiled"
                       (path-add-suffix name #".zo")))
  filter : (any/c . -> . any/c) = values
```

Compiles the Scheme file *src* and saves the compiled code to *dest*. If *dest* is not provided and the "compiled" subdirectory does not already exist, the subdirectory is created. The result of compile-file is the destination file's path.

If the *filter* procedure is provided, it is applied to each source expression, and the result is compiled.

The compile-file procedure is designed for compiling modules files, in that each expression in *src* is compiled independently. If *src* does not contain a single module expression, then earlier expressions can affect the compilation of later expressions when *src* is loaded directly. An appropriate *filter* can make compilation behave like evaluation, but the problem is also solved (as much as possible) by the compile-zos procedure.

See also managed-compile-zo.

13 mzlib/contract

```
(require mzlib/contract)
```

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

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
any	make-proj-contract
any/c	natural-number/c
between/c	none/c
box-immutable/c	not/c
build-compound-type-name	one-of/c
coerce-contract	or/c
cons/c	parameter/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-expr*s.

```
(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.

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

->*rng = (contract-rng-expr ...)
       | any
```

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-rng-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)
```

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

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

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`:

```
(->d real?
     (λ (in)
       (and/c real?
              (λ (out)
                (< (abs (- (sqr out) in))
                   0.01)))))
```

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.

```
(->r ([dom-x contract-dom-expr] ...) rng)
(->r ([dom-x contract-dom-expr] ...) rest-x contract-rest-expr rng)

rng = any
    | (values contract-expr ...)
    | contract-expr
```

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).

```
(->pp ([dom-x contract-dom-expr] ...) pre-cond-expr any)
```

```
(->pp ([dom-x contract-dom-expr] ...)
      pre-cond-expr
      (values [rng-x contract-rng-expr] ...)
      post-cond-expr)
(->pp ([dom-x contract-dom-expr] ...)
      pre-cond-expr
      contract-rng-expr
      rng-x
      post-cond-expr)
```

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 paramterized over *rng-x*, which is bound to the result of the function.

```
(->pp-rest ([dom-x contract-dom-expr] ...) rest-x rest-contract-expr pre-cond-expr any)
(->pp-rest ([dom-x contract-dom-expr] ...)
           rest-x rest-contract-expr
           pre-cond-expr
           (values [rng-x contract-rng-expr] ...)
           post-cond-expr)
(->pp-rest ([dom-x contract-dom-expr] ...)
           rest-x rest-contract-expr
           pre-cond-expr
           contract-rng-expr
           rng-x
           post-cond-expr)
```

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

14 mzlib/control

```
(require mzlib/control)
```

Re-exports scheme/control.

15 mzlib/date

```
(require mzlib/date)
```

Re-exports scheme/date.

16 mzlib/deflate

```
(require mzlib/deflate)
```

Re-exports file/gzip.

17 mzlib/defmacro

```
(require mzlib/defmacro)
```

```
(define-macro id expr)
(define-macro (id . formals) body ...+)
(defmacro id formals body ...+)

formals = (id ...)
        | id
        | (id ...+ . id)
```

Defines a (non-hygienic) macro *id* through a procedure that manipulates S-expressions, as opposed to syntax objects.

In the first form, *expr* must produce a procedure. In the second form, *formals* determines the formal arguments of the procedure, as in `lambda`, and the *expr*s are the procedure body. The last form, with `defmacro`, is like the second form, but with slightly different parentheses.

In all cases, the procedure is generated in the transformer environment, not the normal environment.

In a use of the macro,

```
(id datum ...)
```

`syntax->datum` is applied to the expression, and the transformer procedure is applied to the `cdr` of the resulting list. If the number of `datum`s does not match the procedure's arity, or if *id* is used in a context that does not match the above pattern, then a syntax error is reported.

After the macro procedure returns, the result is compared to the procedure's arguments. For each value that appears exactly once within the arguments (or, more precisely, within the S-expression derived from the original source syntax), if the same value appears in the result, it is replaced with a syntax object from the original expression. This heuristic substitution preserves source location information in many cases, despite the macro procedure's operation on raw S-expressions.

After substituting syntax objects for preserved values, the entire macro result is converted to syntax with `datum->syntax`. The original expression supplies the lexical context and source location for converted elements.

Important: Although `define-macro` is non-hygienic, it is still restricted by Racket's phase separation rules. This means that a macro cannot access run-time bindings, because it is executed in the syntax-expansion phase. Translating code that involves `define-macro` or `defmacro` from an implementation without this restriction usually implies separating macro

related functionality into a `begin-for-syntax` or a module (that will be imported with `require-for-syntax`) and properly distinguishing syntactic information from run-time information.

18 mzlib/etc

```
(require mzlib/etc)
```

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

boolean=?
true
false
build-list
build-string
build-vector
compose
local
symbol=?

```
(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, letrec-syntax, 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:

```
(define-syntax-set (let-current-continuation
                    let-current-escape-continuation)
  (define (mk call-id)
    (lambda (stx)
      (syntax-case stx ()
        [(_ id body1 body ...)
         (with-syntax ([call call-id])
           #'(call (lambda (id) body1 body ...)))])))
  (define let-current-continuation/proc
    (mk (quote-syntax call/cc)))
  (define let-current-escape-continuation/proc
    (mk (quote-syntax call/ec))))
```

```
(evcase key-expr (value-expr body-expr ...) ...+)
(evcase key-expr (value-expr body-expr ...) ... [else body-expr ...])
```

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-expr*s 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.

```
(identity v) → any/c
  v : any/c
```

Returns *v*.

```
(let+ clause body-expr ...+)

clause = (val target expr)
       | (rec target expr)
       | (vals (target ...) expr)
       | (recs (target expr) ...)
       | (_ expr ...)

target = id
       | (values id ...)
```

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 `target`s are bound to the `expr`s. The environment of the `expr`s is the environment active before this clause.
- `(recs (target expr) ...)` : The `target`ss are bound to the `expr`s. The environment of the `expr`s includes all of the `target`ss.
- `(_ expr ...)` : Evaluates the `expr`s 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:

```
> (let+ ([val (values x y) (values 1 2)])
    (list x y))
'(1 2)
> (let ([x 1])
    (let+ ([val x 3]
           [val y x])
      y))
3
```

```
(loop-until start done? next f) → void?
  start : any/c
  done? : (any/c . -> . any)
  next : (any/c . -> . any/c)
  f : (any/c . -> . any)
```

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)
```

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

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*.

If *datum* has a source module, then the expansion attempts to determine the module's run-time 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.

19 mzlib/file

```
(require mzlib/file)
```

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]) → 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*.

```
(build-relative-path base sub ...) → (and/c path? relative-path?)
  base : (or/c path-string?
               (one-of/c 'up 'same))
       (or/c (and/c path-string?
                    relative-path?)
  sub :      (one-of/c 'up 'same))
```

```
(build-absolute-path base sub ...) → (and/c path? absolute-path?)
         (or/c (and/c path-string?
  base :               (not/c relative-path?))
               (one-of/c 'up 'same))
       (or/c (and/c path-string?
  sub :                (not/c complete-path?))
             (one-of/c 'up 'same))
```

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

20 mzlib/for

```
(require mzlib/for)
```

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

define-sequence-syntax
make-do-sequence
:do-in
```

21 mzlib/foreign

```
(require mzlib/foreign)
```

Re-exports scheme/foreign.

22 mzlib/include

```
(require mzlib/include)
```

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

```
(include path-spec)

path-spec = string
          | (build-path elem ...+)
          | (lib file-string collection-string ...)

     elem = string
          | up
          | same
```

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 *elem*s 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-string*s 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.

23 mzlib/inflate

```
(require mzlib/inflate)
```

Re-exports file/gunzip.

24 mzlib/integer-set

```
(require mzlib/integer-set)
```

The mzlib/integer-set library provides functions for working with finite sets of integers. This module is designed for sets that are compactly represented as groups of intervals, even when their cardinality is large. For example, the set of integers from -1000000 to 1000000 except for 0, can be represented as {[-1000000, -1], [1, 1000000]}. This data structure would not be a good choice for the set of all odd integers between 0 and 1000000, which would be {[1, 1], [3, 3], ... [999999, 999999]}.

In addition to the *integer set* abstract type, a *well-formed set* is a list of pairs of exact integers, where each pair represents a closed range of integers, and the entire set is the union of the ranges. The ranges must be disjoint and increasing. Further, adjacent ranges must have at least one integer between them. For example: '((-1 . 2) (4 . 10)) is a well-formed-set as is '((1 . 1) (3 . 3)), but '((1 . 5) (6 . 7)), '((1 . 5) (-3 . -1)), '((5 . 1)), and '((1 . 5) (3 . 6)) are not.

```
(make-integer-set wfs) → integer-set?
  wfs : well-formed-set?
```

Creates an integer set from a well-formed set.

```
(integer-set-contents s) → well-formed-set?
  s : integer-set?
```

Produces a well-formed set from an integer set.

```
(set-integer-set-contents! s wfs) → void?
  s : integer-set?
  wfs : well-formed-set?
```

Mutates an integer set.

```
(integer-set? v) → boolean?
  v : any/c
```

Returns #t if v is an integer set, #f otherwise.

```
(well-formed-set? v) → boolean?
  v : any/c
```

Returns #t if v is a well-formed set, #f otherwise.

```
(make-range) → integer-set?
(make-range elem) → integer-set?
  elem : exact-integer?
(make-range start end) → integer-set?
  start : exact-integer?
  end : exact-integer?
```

Produces, respectively, an empty integer set, an integer set containing only *elem*, or an integer set containing the integers from *start* to *end* inclusive, where (<= *start end*).

```
(intersect x y) → integer-set?
  x : integer-set?
  y : integer-set?
```

Returns the intersection of the given sets.

```
(difference x y) → integer-set?
  x : integer-set?
  y : integer-set?
```

Returns the difference of the given sets (i.e., elements in *x* that are not in *y*).

```
(union x y) → integer-set?
  x : integer-set?
  y : integer-set?
```

Returns the union of the given sets.

```
(split x y) → integer-set?
  x : integer-set?
  y : integer-set?
```

Produces three values: the first is the intersection of *x* and *y*, the second is the difference *x* remove *y*, and the third is the difference *y* remove *x*.

```
(complement s start end) → any
  s : integer-set?
  start : exact-integer?
  end : exact-integer?
```

Returns the a set containing the elements between start to end inclusive that are not in s, where (<= start-k end-k).}

```
(xor x y) → integer-set?
  x : integer-set?
  y : integer-set?
```

Returns an integer set containing every member of *x* and *y* that is not in both sets.

```
(member? k s) → boolean?
  k : exact-integer?
  s : integer-set?
```

Returns `#t` if *k* is in *s*, `#f` otherwise.

```
(get-integer integer-set) → (or/c exact-integer? false/c)
  integer-set : any/c
```

Returns a member of *integer-set*, or `#f` if *integer-set* is empty.

```
(foldr proc base-v s) → any/c
  proc : (exact-integer? any/c . -> . any/c)
  base-v : any/c
  s : integer-set?
```

Applies *proc* to each member of *s* in ascending order, where the first argument to *proc* is the set member, and the second argument is the fold result starting with *base-v*. For example, `(foldr cons null s)` returns a list of all the integers in *s*, sorted in increasing order.

```
(partition s) → (listof integer-set?)
  s : integer-set-list?
```

Returns the coarsest refinement of the sets in *s* such that the sets in the result list are pairwise disjoint. For example, partitioning the sets that represent `'((1 . 2) (5 . 10))` and `'((2 . 2) (6 . 6) (12 . 12))` produces the a list containing the sets for `'((1 . 1) (5 . 5) (7 . 10))` `'((2 . 2) (6 . 6))`, and `'((12 . 12))`.

```
(card s) → exact-nonnegative-integer?
  s : integer-set?
```

Returns the number of integers in the given integer set.

```
(subset? x y) → boolean?
  x : integer-set?
  y : integer-set?
```

Returns true if every integer in *x* is also in *y*, otherwise `#f`.

25 mzlib/kw

```
(require mzlib/kw)
```

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

```
(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)
```

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 *id*s must be unique.

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

25.1 Required Arguments

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

25.2 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.

25.3 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 §25.6 "Mode Keywords"), the value after the first occurrence is used for the variable:

Example:

```
> ((lambda/kw (#:key x [y 2] [z #:zz 3] #:allow-duplicate-keys)
     (list x y z))
   #:x 'x #:zz 'z #:x "foo")
'(x 2 z)
```

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

25.4 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 §25.5 "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:

```
> ((lambda/kw (#:key x y
               #:rest r
               #:other-keys+body rk
               #:all-keys ak
               #:other-keys ok
               #:body b)
    (list r rk b ak ok))
  #:z 1 #:x 2 2 3 4)
'((#:z 1 #:x 2 2 3 4) (#:z 1 2 3 4) (2 3 4) (#:z 1 #:x 2) (#:z 1))
```

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.

25.5 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:

```
> (define/kw (mathop #:key [op +] #:body (x y z))
    (op x y z))
```

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:

```
> (define/kw (mathop #:key [op +]
                     #:body (x y z #:key [convert values]))
    (op (convert x) (convert y) (convert z)))

> (mathop #:op * 2 4 6 #:convert exact->inexact)
48.0
```

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)
procedure mathop~body: expects at least 3 arguments, given
2: 2 4
```

25.6 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.

25.7 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.)

26 mzlib/list

```
(require mzlib/list)
```

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

```
cons?
empty?
empty
foldl
foldr
remv
remq
remove
remv*
remq*
remove*
findf
memf
assf
filter
sort
```

```
(first v) → any/c
  v : pair?
(second v) → any/c
  v : (and/c pair? ....)
(third v) → any/c
  v : (and/c pair? ....)
(fourth v) → any/c
  v : (and/c pair? ....)
(fifth v) → any/c
  v : (and/c pair? ....)
(sixth v) → any/c
  v : (and/c pair? ....)
(seventh v) → any/c
  v : (and/c pair? ....)
(eighth v) → 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) → any/c
  v : pair?
```

The same as cdr.

```
(last-pair v) → 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 *lst1* precede *lst2*.

```
(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.

27 mzlib/match

```
(require mzlib/match)
```

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.

```
(match val-expr clause ...)

clause = [pat expr ...+]
       | [pat (=> id) expr ...+]
```

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

pat	::=	*id*	match anything, bind identifier
	|	_	match anything
	|	*literal*	match literal
	|	'datum	match equal? datum
	|	(*lvp* ...)	match sequence of *lvp*s
	|	(*lvp* ... . *pat*)	match *lvp*s consed onto a *pat*
	|	#(*lvp* ...)	match vector of *pat*s
	|	#&*pat*	match boxed *pat*
	|	($ *struct-id pat* ...)	match *struct-id* instance
	|	(and *pat* ...)	match when all *pat*s 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 *pat*s
	|	`*qp*	match quasipattern
literal	::=	#t	match true
	|	#f	match false
	|	*string*	match equal? string
	|	*number*	match equal? number
	|	*character*	match equal? character
	|	*bytes*	match equal? byte string
	|	keyword	match equal? keyword
	|	regexp	match equal? regexp literal
	|	pregexp	match equal? pregexp literal
lvp	::=	*pat*	greedily match *pat* instances
	|	*pat*	match *pat*
ooo	::=	...	zero or more; ... is literal
	|	___	zero or more
	|	..*k*	*k* or more
	|	__*k*	*k* or more
qp	::=	*literal*	match literal

	id	match `equal?` symbol
	`(`*qp* `...)`	match sequences of *qp*s
	`(`*qp* `... . `*qp*`)`	match sequence of *qp*s consed onto a *qp*
	`(`*qp* `... `*qp* *ooo*`)`	match *qp*s consed onto a repeated *qp*
	`#(`*qp* `...)`	match vector of *qp*s
	`#&`*qp*	match boxed *qp*
	`,`*pat*	match *pat*
	`,@`*pat*	match *pat*, spliced

```
(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`.

28 mzlib/math

```
(require mzlib/math)
```

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

```
e : real?
```

An approximation to Euler's constant: 2.718281828459045.

29 mzlib/md5

```
(require mzlib/md5)
```

Re-exports file/md5.

30 mzlib/os

```
(require mzlib/os)
```

```
(gethostname) → 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.

WARNING: on Unix, the implementation assumes that the platform's `ftruncate` function accepts a `long long` second argument.

31 mzlib/pconvert

```
(require mzlib/pconvert)
```

The mzlib/pconvert library defines routines for printing Scheme values as evaluable S-expressions. Racket's default printing mode also prints values as expressions (in contrast to the Lisp and Scheme 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 Scheme value into another Scheme value such that the new value pretty-writes as a Scheme 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) → (any/c . -> . any/c)
(use-named/undefined-handler use-handler) → 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) → (any/c . -> . any/c)
(named/undefined-handler use-handler) → 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) → ....
  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.

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

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* ands 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.

```
                                (any/c (any/c . -> . any/c)
(current-print-convert-hook) →         (any/c . -> . any/c)
                                       . -> . any/c)
(current-print-convert-hook hook) → void?
         (any/c (any/c . -> . any/c)
  hook :        (any/c . -> . any/c)
                . -> . 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) → 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,

```
(get-shared (build-share (shared ([a (cons 1 b)]
                                  [b (cons 2 a)])
                           a)))
```

might return the list

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

```
(install-converting-printer) → void?
```

Sets the current print handler to print values using `print-convert` and sets `print-as-expression` to `#f` (since the conversion of a value is meant to be printed in `read`able form rather than `eval`uable 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.

```
(print-convert-expr share-info
                    v
                    unroll-once?) → any/c
  share-info : ....
  v : any/c
  unroll-once? : any/c
```

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.

32 mzlib/pconvert-prop

```
(require mzlib/pconvert-prop)
```

```
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.

33 mzlib/plt-match

```
(require mzlib/plt-match)
```

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.

34 mzlib/port

```
(require mzlib/port)
```

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 backslash). Since #! followed by a forward slash or space is a comment, this procedure is not needed before reading Scheme expressions.

35 mzlib/pregexp

```
(require mzlib/pregexp)
```

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) → (or/c string? bytes?)
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  input : (or/c string? bytes?)
          (or/c string? bytes?
  insert :       (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?)
          (or/c string? bytes?
  insert :       (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.

36 mzlib/pretty

```
(require mzlib/pretty)
```

Re-exports scheme/pretty.

37 mzlib/process

```
(require mzlib/process)
```

Re-exports scheme/system.

38 mzlib/restart

```
(require mzlib/restart)
```

```
(restart-mzscheme init-argv
                  adjust-flag-table
                  argv
                  init-namespace)   → boolean?
  init-argv : (vectorof string?)
  adjust-flag-table : (any/c . -> . any/c)
  argv : (vectorof string?)
  init-namespace : (-> any)
```

See `scheme/sandbox` for 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 *init-argv* is read first with the standard Racket command-line parsing. Flags that load files or evaluate expressions (e.g., `-f` and `-e`) are ignored, but flags that set Racket's modes (e.g., `-c` or `-j`) effectively set the default mode before *argv* 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.

39 mzlib/runtime-path

```
(require mzlib/runtime-path)
```

Re-exports scheme/runtime-path.

40 mzlib/sandbox

```
(require mzlib/sandbox)
```

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 ...) → (any/c . -> . any)
            (or/c module-path?
                  (one-of/c 'r5rs 'beginner 'beginner-abbr
                            'intermediate 'intermediate-lambda 'advanced)
 language :
                  (list/c (one-of/c 'special) symbol?)
                  (list/c (one-of/c 'special) symbol?)
                  (cons/c (one-of/c 'begin) list?))
            (or/c (cons/c 'begin list?)
 requires :
                  (listof (or/c module-path? path?)))
```

```
  input-program : any/c
(make-evaluator module-decl) → (any/c . -> . any)
  module-decl : (or/c syntax? pair?)
```

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`.

41 mzlib/sendevent

```
(require mzlib/sendevent)
```

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

```
(send-event  receiver-bytes
             event-class-bytes
             event-id-bytes
            [direct-arg-v
             argument-list])    → any/c
  receiver-bytes : (lambda (s) (and (bytes? s)
                                    (= 4 (bytes-length s))))
  event-class-bytes : (lambda (s) (and (bytes? s)
                                       (= 4 (bytes-length s))))
  event-id-bytes : (lambda (s) (and (bytes? s)
                                    (= 4 (bytes-length s))))
  direct-arg-v : any/c = (void)
  argument-list : list? = null
```

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

42 mzlib/serialize

```
(require mzlib/serialize)
```

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.

```
(define-serializable-struct id-maybe-super (field-id ...) maybe-inspector-expr)
(define-serializable-struct/versions id-maybe-super vers-num (field-id ...)
                                     (other-version-clause ...)
                                     maybe-inspector-expr)

      id-maybe-super = id
                     | (id super-id)

maybe-inspector-expr =
                     | inspector-expr

other-version-clause = (other-vers make-proc-expr
                                   cycle-make-proc-expr)
```

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

43 mzlib/shared

```
(require mzlib/shared)
```

Re-exports scheme/shared.

44 mzlib/string

```
(require mzlib/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.

```
(glob->regexp [str
               hide-dots?
               case-sensitive?
               simple?])        → (or/c regexp? byte-regexp?)
  str : (or/c string bytes?) = ?
  hide-dots? : any/c = #t
  case-sensitive? : any/c
                  = (eq? (system-path-convention-type)'unix)
  simple? : any/c = #f
```

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) → 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.

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

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.

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

Reads the first S-expression from *str* and returns it. The *err-handler* is as in `eval-string`.

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

45 mzlib/struct

```
(require mzlib/struct)
```

```
(copy-struct struct-id struct-expr
             (accessor-id field-expr) ...)
```

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

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.

```
(define-struct/properties id (field-id ...)
                          ((prop-expr val-expr) ...)
                          maybe-inspector-expr)

maybe-inspector-expr =
                     | expr
```

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:

```
> (define-struct/properties point (x y)
    ([prop:custom-write (lambda (p port write?)
                          (fprintf port "(~a, ~a)"
                                   (point-x p)
                                   (point-y p)))]))

> (display (make-point 1 2))
(1, 2)
```

```
(make-->vector struct-id)
```

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

46 mzlib/stxparam

```
(require mzlib/stxparam)
```

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

47 mzlib/surrogate

```
(require mzlib/surrogate)
```

Re-exports scheme/surrogate.

48 mzlib/tar

```
(require mzlib/tar)
```

Re-exports file/tar.

49 mzlib/thread

```
(require mzlib/thread)
```

```
(coroutine proc) → coroutine?
  proc : ((any/c . -> . void?) . -> . any/c)
```

Returns a coroutine object to encapsulate a thread that runs only when allowed. The *proc* procedure should accept one argument, and *proc* is run in the coroutine thread when coroutine-run is called. If coroutine-run returns due to a timeout, then the coroutine thread is suspended until a future call to coroutine-run. Thus, *proc* only executes during the dynamic extent of a coroutine-run call.

The argument to *proc* is a procedure that takes a boolean, and it can be used to disable suspends (in case *proc* has critical regions where it should not be suspended). A true value passed to the procedure enables suspends, and #f disables suspends. Initially, suspends are allowed.

```
(coroutine? v) → any
  v : any/c
```

Returns #t if *v* is a coroutine produced by coroutine, #f otherwise.

```
(coroutine-run until coroutine) → boolean?
  until : (or/c evt? real?)
  coroutine : coroutine?
```

Allows the thread associated with *coroutine* to execute for up as long as *until* milliseconds (of *until* is a real number) or *until* is ready (if *until* is an event). If *coroutine*'s procedure disables suspends, then the coroutine can run arbitrarily long until it re-enables suspends.

The coroutine-run procedure returns #t if *coroutine*'s procedure completes (or if it completed earlier), and the result is available via coroutine-result. The coroutine-run procedure returns #f if *coroutine*'s procedure does not complete before it is suspended after timeout-secs. If *coroutine*'s procedure raises an exception, then it is re-raised by coroutine-run.

```
(coroutine-result coroutine) → any
  coroutine : coroutine
```

Returns the result for *coroutine* if it has completed with a value (as opposed to an exception), #f otherwise.

```
(coroutine-kill coroutine) → void?
  coroutine : coroutine?
```

Forcibly terminates the thread associated with *coroutine* if it is still running, leaving the coroutine result unchanged.

```
(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.

```
(run-server  port-no
             conn-proc
             conn-timeout
            [handler
             listen
             close
             accept
             accept/break]) → void?
  port-no : (integer-in 1 65535)
  conn-proc : (input-port? output-port? . -> . any)
  conn-timeout : (and/c real? (not/c negative?))
  handler : (exn? . -> . any/c) = void
  listen :  ((integer-in 1 65535) (one-of/c 5) (one-of/c #t)
             . -> . listener?)
         = tcp-listen
  close : (listener? . -> . any) = tcp-close
  accept : (listener? . ->* . (input-port? output-port?))
         = tcp-accept
  accept/break : (listener? . ->* . (input-port? output-port?))
               = tcp-accept/enable-break
```

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.

50 mzlib/trace

```
(require mzlib/trace)
```

Re-exports racket/trace.

51 mzlib/traceld

```
(require mzlib/traceld)
```

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.

52 mzlib/trait

```
(require mzlib/trait)
```

Re-exports scheme/trait.

53 mzlib/transcr

```
(require mzlib/transcr)
```

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*.

54 mzlib/unit

```
(require mzlib/unit)
```

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

```
(struct id (field-id ...) omit-decl ...)

omit-decl = -type
          | -selectors
          | -setters
          | -constructor
```

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

```
(struct/ctc id ([field-id contract-expr] ...) omit-decl ...)

omit-decl = -type
          | -selectors
          | -setters
          | -constructor
```

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

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

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

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

Like struct~s and struct~s/ctc, but the constructor is named the same as the type, instead of with make- prefix.

55 mzlib/unit-exptime

```
(require mzlib/unit-exptime)
```

Re-exports scheme/unit-exptime.

56 mzlib/unit200

```
(require mzlib/unit200)
```

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.

57 mzlib/unitsig200

```
(require mzlib/unitsig200)
```

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.

58 mzlib/zip

```
(require mzlib/zip)
```

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