MzScheme: Legacy Language

Version 5.3.6

August 9, 2013

```
(require mzscheme)
```

The mzscheme language provides nearly the same bindings as the mzscheme module of PLT Scheme version 372 and earlier.

Unlike version 372, the mzscheme language does not include set-car! or set-cdr!, and cons makes immutable pairs, as in scheme/base; those changes make modules built on mzscheme reasonably compatible with modules built on scheme/base.

Otherwise, the mzscheme language shares many bindings with scheme/base. It renames a few bindings, such as syntax-object->datum instead of syntax->datum, and it provides old versions of some syntactic forms, such as lambda without support for keyword and optional arguments. In addition, mzscheme includes all of the exports of racket/tcp and racket/udp.

Contents

1 Old Syntactic Forms

```
(#%module-begin form ...)
```

Like `#%plain-module-begin` from `scheme/base`, but `(require-for-syntax mzscheme)` is added to the beginning of the *form* sequence, thus importing `mzscheme` into the transformer environment for the module body. (In contrast, `scheme/base` exports `for-syntax` minimal transformer support, while `scheme` exports all of `scheme/base` `for-syntax`.)

```
(#%plain-module-begin form ...)
```

The same binding as `#%plain-module-begin` from `scheme/base`.

```
(#%plain-lambda formals body ...+)
```

The same binding as `#%plain-lambda` in `scheme/base`. (This binding was not present in version 372 and earlier.)

```
(lambda formals body ...+)
(λ formals body ...+)
```

The same bindings as `#%plain-lambda`.

```
(#%app proc-expr arg-expr ...)
(#%app)
```

The same binding as `#%plain-app` from `scheme/base`.

```
(#%plain-app proc-expr arg-expr ...)
(#%plain-app)
```

The same binding as `#%app`. (This binding was not present in version 372 and earlier.)

```
(define id expr)
(define (head args) body ...+)

head = id
     | (head args)

args = arg-id ...
     | arg-id ... . rest-id
```

Like `define` in `scheme/base`, but without support for keyword arguments or optional arguments.

```
(define-syntax id expr)
(define-syntax (head args) body ...+)
(define-for-syntax id expr)
(define-for-syntax (head args) body ...+)
```

Like `define-syntax` and `define-for-syntax` in `scheme/base`, but without support for keyword arguments or optional arguments (i.e., *head* is as for `define`).

```
(if test-expr then-expr else-expr)
(if test-expr then-expr)
```

Like `if` in `scheme/base`, but *else-expr* defaults to `(void)`.

```
(cond cond-clause ...)
(case val-expr case-clause ...)
```

Like `cond` and `case` in `scheme/base`, but `else` and `=>` are recognized as unbound identifiers, instead of as the `scheme/base` bindings.

```
(fluid-let ([id expr] ...) body ...+)
```

Provides a kind of dynamic binding via mutation of the *id*s.

The `fluid-let` form first evaluates each *expr* to obtain an *entry value* for each *id*. As evaluation moves into *body*, either though normal evaluation or a continuation jump, the current value of each *id* is swapped with the entry value. On exit from *body*, then the current value and entry value are swapped again.

```
(define-struct id-maybe-super (field-id ...) maybe-inspector-expr)

maybe-inspector-expr =
                     | expr
```

Like `define-struct` from `scheme/base`, but with fewer options. Each field is implicitly mutable, and the optional *expr* is analogous to supplying an `#:inspector` expression.

```
(let-struct id-maybe-super (field-id ...) body ...+)
```

Expands to

```
(let ()
  (define-struct id-maybe-super (field-id ...))
  body ...+)
```

```
(require raw-require-spec)
(require-for-syntax raw-require-spec)
(require-for-template raw-require-spec)
(require-for-label raw-require-spec)
(provide raw-provide-spec)
(provide-for-syntax raw-provide-spec)
(provide-for-label raw-provide-spec)
```

Like `#%require` and `#%provide`. The `-for-syntax`, `-for-template`, and `-for-label` forms are translated to `#%require` and `#%provide` using `for-syntax`, `for-template`, and `for-label` sub-forms, respectively.

```
(#%datum . datum)
```

Expands to `'datum`, even if *datum* is a keyword.

```
(#%top-interaction . form)
```

The same as `#%top-interaction` in `scheme/base`.

2 Old Functions

```
(apply proc v ... lst) → any
  proc : procedure?
  v : any/c
  lst : list?
```

Like `apply` from `scheme/base`, but without support for keyword arguments.

```
prop:procedure : struct-type-property?
```

Like `prop:procedure` from `scheme/base`, but even if the property's value for a structure type is a procedure that accepts keyword arguments, then instances of the structure type still do not accept keyword arguments. (In contrast, if the property's value is an integer for a field index, then a keyword-accepting procedure in the field for an instance causes the instance to accept keyword arguments.)

```
(open-input-file file [mode]) → input-port?
  file : path-string?
  mode : (one-of/c 'text 'binary) = 'binary
(open-output-file file [mode exists]) → input-port?
  file : path-string?
  mode : (one-of/c 'text 'binary) = 'binary
  exists : (one-of/c 'error 'append 'update
                     'replace 'truncate 'truncate/replace)
         = 'error
(open-input-output-file file [mode exists])
 → input-port? output-port?
  file : path-string?
  mode : (one-of/c 'text 'binary) = 'binary
  exists : (one-of/c 'error 'append 'update
                     'replace 'truncate 'truncate/replace)
         = 'error
(with-input-from-file file thunk [mode]) → any
  file : path-string?
  thunk : (-> any)
  mode : (one-of/c 'text 'binary) = 'binary
(with-output-to-file file thunk [mode exists]) → any
  file : path-string?
  thunk : (-> any)
  mode : (one-of/c 'text 'binary) = 'binary
  exists : (one-of/c 'error 'append 'update
                     'replace 'truncate 'truncate/replace)
         = 'error
```

```
(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 `open-input-file`, etc. from `scheme/base`, but *mode* and *exists* arguments are not keyword arguments. When both *mode* and *exists* are accepted, they are accepted in either order.

```
(syntax-object->datum stx) → any
  stx : syntax?
(datum->syntax-object ctxt v srcloc [prop cert]) → syntax?
  ctxt : (or/c syntax? false/c)
  v : any/c
          (or/c syntax? false/c
                (list/c any/c
                        (or/c exact-positive-integer? false/c)
                        (or/c exact-nonnegative-integer? false/c)
                        (or/c exact-nonnegative-integer? false/c)
  srcloc :              (or/c exact-positive-integer? false/c))
                (vector/c any/c
                          (or/c exact-positive-integer? false/c)
                          (or/c exact-nonnegative-integer? false/c)
                          (or/c exact-nonnegative-integer? false/c)
                          (or/c exact-positive-integer? false/c)))
  prop : (or/c syntax? false/c) = #f
  cert : (or/c syntax? false/c) = #f
```

The same as `syntax->datum` and `datum->syntax`.

```
(module-identifier=? a-id b-id) → boolean?
  a-id : syntax?
  b-id : syntax?
(module-transformer-identifier=? a-id b-id) → boolean?
  a-id : syntax?
  b-id : syntax?
(module-template-identifier=? a-id b-id) → boolean?
  a-id : syntax?
  b-id : syntax?
```

```
(module-label-identifier=? a-id b-id) → boolean?
  a-id : syntax?
  b-id : syntax?
(free-identifier=? a-id b-id) → boolean?
  a-id : syntax?
  b-id : syntax?
```

The module-identifier=?, etc. functions are the same as free-identifier=?, etc. in scheme/base.

The free-identifier=? procedure returns

```
(and (eq? (syntax-e a) (syntax-e b))
     (module-identifier=? a b))
```

```
(make-namespace [mode]) → namespace?
  mode : (one-of/c 'initial 'empty) = 'initial
```

Creates a namespace with mzscheme attached. If the *mode* is empty, the namespace's top-level environment is left empty. If *mode* is 'initial, then the namespace's top-level environment is initialized with (namespace-require/copy 'mzscheme). See also make-base-empty-namespace.

```
(namespace-transformer-require req) → void?
  req : any/c
```

Equivalent to (namespace-require '(for-syntax ,*req*)).

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

Raises exn:fail, because the operations are not supported.

```
(hash-table? v) → hash-table?
  v : any/c
(hash-table? v flag) → hash-table?
  v : any/c
  flag : (one-of/c 'weak 'equal 'eqv)
(hash-table? v flag flag2) → hash-table?
  v : any/c
  flag : (one-of/c 'weak 'equal 'eqv)
  flag2 : (one-of/c 'weak 'equal 'eqv)
```

Returns `#t` if *v* is a hash table created by `make-hash-table` or `make-immutable-hash-table` with the given *flag*s (or more), `#f` otherwise. If *flag2* is provided, it must be distinct from *flag* and `'equal` cannot be used with `'eqv`, otherwise the `exn:fail:contract` exception is raised.

```
(make-hash-table) → hash-table?
(make-hash-table flag) → hash-table?
  flag : (one-of/c 'weak 'equal 'eqv)
(make-hash-table flag flag2) → hash-table?
  flag : (one-of/c 'weak 'equal 'eqv)
  flag2 : (one-of/c 'weak 'equal 'eqv)
```

Creates and returns a new hash table. If provided, each *flag* must one of the following:

- `'weak` — creates a hash table with weakly-held keys via `make-weak-hash`, `make-weak-hasheq`, or `make-weak-hasheqv`.
- `'equal` — creates a hash table that compares keys using `equal?` instead of `eq?` using `make-hash` or `make-weak-hash`.
- `'eqv` — creates a hash table that compares keys using `eqv?` instead of `eq?` using `make-hasheqv` or `make-weak-hasheqv`.

By default, key comparisons use `eq?` (i.e., the hash table is created with `make-hasheq`). If *flag2* is redundant or `'equal` is provided with `'eqv`, the `exn:fail:contract` exception is raised.

```
(make-immutable-hash-table assocs)
 → (and/c hash-table? immutable?)
  assocs : (listof pair?)
(make-immutable-hash-table assocs flag)
 → (and/c hash-table? immutable?)
  assocs : (listof pair?)
  flag : (one-of/c 'equal 'eqv)
```

Like `make-immutable-hash`, `make-immutable-hasheq`, or `make-immutable-hasheqv`, depending on whether an `'equal` or `'eqv` *flag* is provided.

```
hash-table-get : procedure?
hash-table-put! : procedure?
hash-table-remove! : procedure?
hash-table-count : procedure?
hash-table-copy : procedure?
hash-table-map : procedure?
hash-table-for-each : procedure?
```

```
hash-table-iterate-first : procedure?
hash-table-iterate-next : procedure?
hash-table-iterate-value : procedure?
hash-table-iterate-key : procedure?
```

The same as `hash-ref`, `hash-set!`, `hash-remove!`, `hash-count`,`hash-copy`, `hash-map`, `hash-for-each`, `hash-iterate-first`, `hash-iterate-next`, `hash-iterate-value`, and `hash-iterate-key`, respectively.

```
expand-path : procedure?
```

The same as `cleanse-path`.

```
list-immutable : procedure?
```

The same as `list`.

```
(collection-file-path file collection ...+) → path?
  file : path-string?
  collection : path-string?
(collection-path collection ...+) → path?
  collection : path-string?
```

Like `collection-file-path` and `collection-path`, but without the `#:fail` option.

3 Extra Libraries

The `mzscheme` library re-exports `racket/promise`, `racket/tcp`, and `racket/udp`.

4 Omitted Forms and Functions

In addition to forms and functions that have replacements listed in §1 “Old Syntactic Forms” and §2 “Old Functions”, the following forms and functions are exported by `racket/base` but not `mzscheme`:

```
compose filter sort foldl foldr
remv remq remove remv* remq* remove* memf assf findf
build-vector build-string build-list
hash-keys hash-values hash->list hash-set* hash-set*!
hash-update hash-update!
vector-copy!
thread-send thread-receive thread-try-receive thread-receive-evt
log-fatal log-error log-warning log-info log-debug
log-message log-level? make-logger logger?
current-logger logger-name make-log-receiver log-receiver?
```