Compatibility: Features from Racket Relatives

Version 8.8

February 5, 2023

The `compatibility` collection includes features borrowed from other languages closely related to Racket. We provide these features to ease porting code from these languages to Racket.

We do *not* recommend using any of these bindings in new code. Racket provides better alternatives, which we point to in this manual. We *strongly* recommend using these alternatives.

1 Legacy macro support

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

This `compatibility/defmacro` library provides support for writing legacy macros. Support for `defmacro` is provided primarily for porting code from other languages (e.g., some implementations of Scheme or Common Lisp) that use symbol-based macro systems.

Use of `defmacro` for modern Racket code is ***strongly*** discouraged. Instead, consider using `syntax-parse` or `define-simple-macro`.

```
(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 for-syntax) and properly distinguishing syntactic information from run-time information.

2 Limiting Scope: `define-package`, `open-package`, ...

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

This `compatibility/package` library provides support for the Chez Scheme module system. Support for packages is provided primarily to help porting code.

Use of packages for modern Racket code is discouraged. Instead, consider using submodules.

```
(define-package package-id exports form ...)
(open-package package-id)

exports = (id ...)
        | #:only (id ...)
        | #:all-defined
        | #:all-defined-except (id ...)
```

The `define-package` form is based on the `module` form of Chez Scheme [Waddell99].

The `define-package` form is similar to `module`, except that it can appear in any definition context. The *`form`*s within a `define-package` form can be definitions or expressions; definitions are not visible outside the `define-package` form, but *`exports`* determines a subset of the bindings that can be made visible outside the package using the definition form `(open-package package-id)`.

The `(id ...)` and `#:only (id ...)` *`exports`* forms are equivalent: exactly the listed *`id`*s are exported. The `#:all-defined` form exports all definitions from the package body, and `#:all-defined-except (id ...)` exports all definitions except the listed *`id`*s.

All of the usual definition forms work within a `define-package` body, and such definitions are visible to all expressions within the body (and, in particular, the definitions can refer to each other). However, `define-package` handles `define*`, `define*-syntax`, `define*-values`, `define*-syntaxes`, and `open*-package` specially: the bindings introduced by those forms within a `define-package` body are visible only to *`form`*s that appear later in the body, and they can shadow any binding from preceding *`form`*s (even if the preceding binding did not use one of the special `*` definition forms). If an exported identifier is defined multiple times, the last definition is the exported one.

Examples:

```
> (define-package presents (doll)
    (define doll "Molly Coddle")
    (define robot "Destructo"))
> doll
doll: undefined;
 cannot reference an identifier before its definition
  in module: top-level
```

```
> robot
robot: undefined;
 cannot reference an identifier before its definition
  in module: top-level
> (open-package presents)
> doll
"Molly Coddle"
> robot
robot: undefined;
 cannot reference an identifier before its definition
  in module: top-level
> (define-package big-russian-doll (middle-russian-doll)
    (define-package middle-russian-doll (little-russian-doll)
      (define little-russian-doll "Anastasia")))
> (open-package big-russian-doll)
> (open-package middle-russian-doll)
> little-russian-doll
"Anastasia"
```

```
(package-begin form ...)
```

Similar to `define-package`, but it only limits the visible of definitions without binding a package name. If the last *`form`* is an expression, then the expression is in tail position for the `package-begin` form, so that its result is the `package-begin` result.

A `package-begin` form can be used as an expression, but if it is used in a context where definitions are allowed, then the definitions are essentially spliced into the enclosing context (though the defined bindings remain hidden outside the `package-begin`).

Examples:

```
> (package-begin
    (define secret "mimi")
    (list secret))
'("mimi")
> secret
secret: undefined;
 cannot reference an identifier before its definition
  in module: top-level
```

```
define*
define*-values
define*-syntax
define*-syntaxes
open*-package
```

Equivalent to define, define-values, define-syntax, define-syntaxes, and open-package, except within a define-package or package-begin form, where they create bindings that are visible only to later body forms.

Examples:

```
> (define-package mail (cookies)
    (define* cookies (list 'sugar))
    (define* cookies (cons 'chocolate-chip cookies)))
> (open-package mail)
> cookies
'(chocolate-chip sugar)
> (define-syntax-rule (define-seven id) (define id 7))
> (define-syntax-rule (define*-seven id)
    (begin
      (define-package p (id) (define-seven id))
      (open*-package p)))
> (package-begin
    (define vii 8)
    (define*-seven vii)
    vii)
8
```

```
(package? v) → boolean?
  v : any/c
(package-exported-identifiers id) → (listof identifier?)
  id : identifier?
(package-original-identifiers id) → (listof identifier?)
  id : identifier?
```

The package?, package-exported-identifiers, and package-original-identifiers functions are exported for-syntax by compatibility/package.

The package? predicate returns #t if *v* is a package value as obtained by syntax-local-value on an identifier that is bound to a package.

Given such an identifier, the package-exported-identifiers function returns a list of identifiers that correspond to the bindings that would be introduced by opening the package in the lexical context being expanded. The package-original-identifiers function returns a parallel list of identifiers for existing bindings of package's exports.

2.1 Legacy Racket Package Library

```
(require racket/package)        package: compatibility-lib
```

The `racket/package` library re-exports `compatibility/package` for backward compatibility.

3 Mutable List Functions

```
(require compatibility/mlist)
```
package: compatibility-lib

This compatibility/mlist library provides support for mutable lists. Support is provided primarily to help porting Lisp/Scheme code to Racket.

Use of mutable lists for modern Racket code is ***strongly*** discouraged. Instead, consider using lists.

For functions described in this section, contracts are not directly enforced. In particular, when a mutable list is expected, supplying any other kind of value (or mutating a value that starts as a mutable list) tends to produce an exception from mcar or mcdr.

```
(mlist? v) → boolean?
  v : any/c
```

Returns #t if *v* is a mutable list: either the empty list, or a mutable pair whose second element is a mutable list.

```
(mlist v ...) → mlist?
  v : any/c
```

Returns a newly allocated mutable list containing the *v*s as its elements.

```
(list->mlist lst) → mlist?
  lst : list?
```

Returns a newly allocated mutable list with the same elements as *lst*.

```
(mlist->list mlst) → list?
  mlst : mlist?
```

Returns a newly allocated list with the same elements as *mlst*.

```
(mlength mlst) → exact-nonnegative-integer?
  mlst : mlist?
```

Returns the number of elements in *mlst*.

```
(mlist-ref mlst pos) → any/c
  mlst : mlist?
  pos : exact-nonnegative-integer?
```

Like list-ref, but for mutable lists.

```
(mlist-tail mlst pos) → any/c
  mlst : mlist?
  pos : exact-nonnegative-integer?
```

Like `list-tail`, but for mutable lists.

```
(mappend mlst ...) → mlist?
  mlst : mlist?
(mappend mlst ... v) → any/c
  mlst : mlist?
  v : any/c
```

Like `append`, but for mutable lists.

```
(mappend! mlst ...) → mlist?
  mlst : mlist?
(mappend! mlst ... v) → any/c
  mlst : mlist?
  v : any/c
```

The `mappend!` procedure appends the given mutable lists by mutating the tail of each to refer to the next, using `set-mcdr!`. Empty lists are dropped; in particular, the result of calling `mappend!` with one or more empty lists is the same as the result of the call with the empty lists removed from the set of arguments.

```
(mreverse mlst) → mlist?
  mlst : mlist?
```

Like `reverse`, but for mutable lists.

```
(mreverse! mlst) → mlist?
  mlst : mlist?
```

Like `mreverse`, but destructively reverses the mutable list by using all of the mutable pairs in *mlst* and changing them with `set-mcdr!`.

```
(mmap proc mlst ...+) → mlist?
  proc : procedure?
  mlst : mlist?
```

Like `map`, but for mutable lists.

```
(mfor-each proc mlst ...+) → void?
  proc : procedure?
  mlst : mlist?
```

Like for-each, but for mutable lists.

```
(mmember v mlst) → (or/c mlist? #f)
  v : any/c
  mlst : mlist?
```

Like member, but for mutable lists.

```
(mmemv v mlst) → (or/c mlist? #f)
  v : any/c
  mlst : mlist?
```

Like memv, but for mutable lists.

```
(mmemq v mlst) → (or/c list? #f)
  v : any/c
  mlst : mlist?
```

Like memq, but for mutable lists.

```
(massoc v mlst) → (or/c mpair? #f)
  v : any/c
  mlst : (mlistof mpair?)
```

Like assoc, but for mutable lists of mutable pairs.

```
(massv v mlst) → (or/c mpair? #f)
  v : any/c
  mlst : (mlistof mpair?)
```

Like assv, but for mutable lists of mutable pairs.

```
(massq v mlst) → (or/c mpair? #f)
  v : any/c
  mlst : (mlistof mpair?)
```

Like assq, but for mutable lists of mutable pairs.

```
(mlistof pred) → (any/c . -> . boolean?)
  pred : (any/c . -> . any/c)
```

Returns a procedure that returns #t when given a mutable list for which *pred* returns a true value for all elements.

3.1 Legacy Racket Mutable List Library

```
(require racket/mpair)
```
package: `compatibility-lib`

The `racket/mpair` library re-exports `compatibility/mlist` for backward compatibility.

Bibliography

[Waddell99] Oscar Waddell and R. Kent Dybvig, “Extending the Scope of Syntactic Abstraction,” Principles of Programming Languages, 1999.