Lazy Racket

Version 9.4.0.3

Eli Barzilay

October 29, 2026

`#lang lazy` package: `lazy`

Lazy Racket is available as both a language level and a module that can be used to write lazy code. To write lazy code, simply use `lazy` as your module's language:

```
#lang lazy
... lazy code here...
```

Function applications are delayed, and promises are automatically forced. The language provides bindings that are equivalent to most of the `racket/base` and `racket/list` libraries. Primitives are strict in the expected places; struct constructors are lazy; `if`, `and`, `or` etc. are plain (lazy) functions. Strict functionality is provided as-is: `begin`, I/O, mutation, parameterization, etc. To have your code make sense, you should chain side effects in `begin`s, which will sequence things properly. (Note: This is similar to threading monads through your code—only use `begin` where order matters.)

Mixing lazy and strict code is simple: you just write the lazy code in the lazy language, and strict code as usual. The lazy language treats imported functions (those that were not defined in the lazy language) as strict, and on the strict side you only need to force (possibly recursively) through promises.

A few side-effect bindings are provided as-is. For example, `read` and `printf` do the obvious thing—but note that the language is a call-by-need, and you need to be aware when promises are forced. There are also bindings for `begin` (delays a computation that forces all subexpressions), `when`, `unless`, etc. There are, however, less reliable and might change (or be dropped) in the future.

There are a few additional bindings, the important ones are special forms that force strict behaviour—there are several of these that are useful in forcing different parts of a value in different ways, as described in §2 "Forcing Values".

1 Lazy Forms and Functions

`lambda`

Lazy variant of `lambda`.

`define`

Lazy variant of `define`.

`let`

Lazy variant of `let`.

`let*`

Lazy variant of `let*`.

`letrec`

Lazy variant of `letrec`.

`parameterize`

Lazy variant of `parameterize`.

`define-values`

Lazy variant of `define-values`.

`let-values`

Lazy variant of `let-values`.

`let*-values`

Lazy variant of `let*-values`.

`letrec-values`

Lazy variant of `letrec-values`.

`if`

Lazy variant of `if`.

`set!`

Lazy variant of `set!`.

`begin`

Lazy variant of `begin`.

`begin0`

Lazy variant of `begin0`.

`when`

Lazy variant of `when`.

`unless`

Lazy variant of `unless`.

`cond`

Lazy variant of `cond`.

`case`

Lazy variant of `case`.

`values` : `procedure?`

Lazy variant of `values`.

`make-struct-type` : `procedure?`

Lazy variant of `make-struct-type`.

`cons` : `procedure?`

Lazy variant of `cons`.

`list` : `procedure?`

Lazy variant of `list`.

`list*` : `procedure?`

Lazy variant of `list*`.

`vector` : `procedure?`

Lazy variant of `vector`.

`box` : `procedure?`

Lazy variant of `box`.

`and` : `procedure?`

Lazy variant of `and`.

`or` : `procedure?`

Lazy variant of `or`.

`set-mcar!` : `procedure?`

Lazy variant of `set-mcar!`.

`set-mcdr!` : `procedure?`

Lazy variant of `set-mcdr!`.

`vector-set!` : `procedure?`

Lazy variant of `vector-set!`.

`set-box!` : `procedure?`

Lazy variant of `set-box!`.

`error` : `procedure?`

Lazy variant of `error`.

`printf` : `procedure?`

Lazy variant of `printf`.

`fprintf` : `procedure?`

Lazy variant of `fprintf`.

`display` : `procedure?`

Lazy variant of `display`.

`write` : `procedure?`

Lazy variant of `write`.

`print` : `procedure?`

Lazy variant of `print`.

`eq?` : `procedure?`

Lazy variant of `eq?`.

`eqv?` : `procedure?`

Lazy variant of `eqv?`.

`equal?` : `procedure?`

Lazy variant of `equal?`.

`list?` : `procedure?`

Lazy variant of `list?`.

`length` : `procedure?`

Lazy variant of `length`.

`list-ref` : `procedure?`

Lazy variant of `list-ref`.

`list-tail` : `procedure?`

Lazy variant of `list-tail`.

`append` : `procedure?`

Lazy variant of `append`.

`map` : `procedure?`

Lazy variant of `map`.

`for-each` : `procedure?`

Lazy variant of `for-each`.

`andmap` : `procedure?`

Lazy variant of `andmap`.

`ormap` : `procedure?`

Lazy variant of `ormap`.

`member` : `procedure?`

Lazy variant of `member`.

`memq` : `procedure?`

Lazy variant of `memq`.

`memv` : `procedure?`

Lazy variant of `memv`.

`assoc` : `procedure?`

Lazy variant of `assoc`.

`assq` : `procedure?`

Lazy variant of `assq`.

```
assv : procedure?
```

Lazy variant of assv.

```
reverse : procedure?
```

Lazy variant of reverse.

```
caar : procedure?
```

Lazy variant of caar.

```
cadr : procedure?
```

Lazy variant of cadr.

```
cdar : procedure?
```

Lazy variant of cdar.

```
cddr : procedure?
```

Lazy variant of cddr.

```
caaar : procedure?
```

Lazy variant of caaar.

```
caadr : procedure?
```

Lazy variant of caadr.

```
cadar : procedure?
```

Lazy variant of cadar.

```
caddr : procedure?
```

Lazy variant of caddr.

```
cdaar : procedure?
```

Lazy variant of cdaar.

```
cdadr : procedure?
```

Lazy variant of cdadr.

```
cddar : procedure?
```

Lazy variant of cddar.

```
cdddr : procedure?
```

Lazy variant of cdddr.

```
caaaar : procedure?
```

Lazy variant of caaaar.

```
caaadr : procedure?
```

Lazy variant of caaadr.

```
caadar : procedure?
```

Lazy variant of caadar.

```
caaddr : procedure?
```

Lazy variant of caaddr.

```
cadaar : procedure?
```

Lazy variant of cadaar.

```
cadadr : procedure?
```

Lazy variant of cadadr.

```
caddar : procedure?
```

Lazy variant of caddar.

```
cadddr : procedure?
```

Lazy variant of cadddr.

```
cdaaar : procedure?
```

Lazy variant of cdaaar.

```
cdaadr : procedure?
```

Lazy variant of cdaadr.

```
cdadar : procedure?
```

Lazy variant of cdadar.

```
cdaddr : procedure?
```

Lazy variant of cdaddr.

```
cddaar : procedure?
```

Lazy variant of cddaar.

`cddadr` : `procedure?`

Lazy variant of `cddadr`.

`cdddar` : `procedure?`

Lazy variant of `cdddar`.

`cddddr` : `procedure?`

Lazy variant of `cddddr`.

`first` : `procedure?`

Lazy variant of `first`.

`second` : `procedure?`

Lazy variant of `second`.

`third` : `procedure?`

Lazy variant of `third`.

`fourth` : `procedure?`

Lazy variant of `fourth`.

`fifth` : `procedure?`

Lazy variant of `fifth`.

`sixth` : `procedure?`

Lazy variant of `sixth`.

`seventh` : `procedure?`

Lazy variant of `seventh`.

`eighth` : `procedure?`

Lazy variant of `eighth`.

`rest` : `procedure?`

Lazy variant of `rest`.

`cons?` : `procedure?`

Lazy variant of `cons?`.

`empty` : `procedure?`

Lazy variant of `empty`.

`empty?` : `procedure?`

Lazy variant of `empty?`.

`foldl` : `procedure?`

Lazy variant of `foldl`.

`foldr` : `procedure?`

Lazy variant of `foldr`.

`last-pair` : `procedure?`

Lazy variant of `last-pair`.

`remove : procedure?`

Lazy variant of `remove`.

`remq : procedure?`

Lazy variant of `remq`.

`remv : procedure?`

Lazy variant of `remv`.

`remove* : procedure?`

Lazy variant of `remove*`.

`remq* : procedure?`

Lazy variant of `remq*`.

`remv* : procedure?`

Lazy variant of `remv*`.

`memf : procedure?`

Lazy variant of `memf`.

`assf : procedure?`

Lazy variant of `assf`.

`filter : procedure?`

Lazy variant of `filter`.

`sort : procedure?`

Lazy variant of `sort`.

`true : procedure?`

Lazy variant of `true`.

`false : procedure?`

Lazy variant of `false`.

`boolean=? : procedure?`

Lazy variant of `boolean=?`.

`symbol=? : procedure?`

Lazy variant of `symbol=?`.

`compose : procedure?`

Lazy variant of `compose`.

`build-list : procedure?`

Lazy variant of `build-list`.

`take : procedure?`

Lazy variant of `take`.

`identity : procedure?`

Lazy identity function.

`cycle : procedure?`

Creates a lazy infinite list that repeats its input arguments in order.

2 Forcing Values

```
(require lazy/force)
```
package: lazy

The bindings of `lazy/force` are re-provided by `lazy`.

```
(! expr) → any/c
  expr : any/c
```

Evaluates *expr* strictly. The result is always forced, over and over until it gets a non-promise value.

```
(!! expr) → any/c
  expr : any/c
```

Similar to `!`, but recursively forces a structure (e.g: lists).

```
(!list expr) → list?
  expr : (or/c promise? list?)
```

Forces the *expr* which is expected to be a list, and forces the `cdr`s recursively to expose a proper list structure.

```
(!!list expr) → list?
  expr : (or/c promise? list?)
```

Similar to `!list` but also forces (using `!`) the elements of the list.