Legacy Print Convert

Version 9.2.0.2

April 22, 2026

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

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

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

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

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

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

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

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

```
(booleans-as-true/false) → boolean?
```

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

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

```
(use-named/undefined-handler) → (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`.

```
(hash-table-constructor-with-lists) → boolean?
(hash-table-constructor-with-lists use-list?) → void?
  use-list? : any/c
```

A parameter that controls whether the key/value pairs in a hash are printed with `cons` or `list`. The initial value of the parameter is `#f`, meaning that a hash table like `(hash 'x 1 'y 2)` converts to `'(make-immutable-hash (list (cons 'y 2) (cons 'x 1)))`.

Added in version 1.2 of package `pconvert-lib`.

```
(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.

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

1 Print Convert Properties

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

```
prop:print-converter : property?
(print-converter? v) → any
  v : any/c
(print-converter-proc v)
 → (any/c (any/c . -> . any/c) . -> . any/c)
  v : print-converter?
```

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

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

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

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

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

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