Essentials of Programming Languages Language

Version 9.4.0.5

November 14, 2026

The *Essentials of Programming Languages* language in DrRacket provides a subset of functions and syntactic forms of `racket`—mostly the ones that correspond to `r5rs` forms. See below for a complete list. The language is intended for use with the textbook [EoPL].

`#lang eopl` package: `eopl`

The following bindings are re-provided from `racket`:

`make-parameter`	`*`	`inexact?`
`parameterize`	`/`	`zero?`
`print-struct`	`abs`	`positive?`
`unquote`	`gcd`	`negative?`
`unquote-splicing`	`lcm`	`odd?`
`quote`	`exp`	`even?`
`quasiquote`	`log`	`quotient`
`if`	`sin`	`remainder`
`lambda`	`cos`	`modulo`
`letrec`	`tan`	`floor`
`define-syntax`	`not`	`ceiling`
`delay`	`eq?`	`truncate`
`let`	`make-string`	`round`
`let*`	`symbol->string`	`numerator`
`let-syntax`	`string->symbol`	`denominator`
`letrec-syntax`	`make-rectangular`	`asin`
`and`	`exact->inexact`	`acos`
`or`	`inexact->exact`	`atan`
`cond`	`number->string`	`sqrt`
`case`	`string->number`	`expt`
`do`	`rationalize`	`make-polar`
`begin`	`output-port?`	`real-part`
`set!`	`current-input-port`	`imag-part`

```
#%module-begin      current-output-port     angle
#%app               current-error-port      magnitude
#%datum             open-input-file         input-port?
#%top               open-output-file        read
#%top-interaction   close-input-port        read-char
#%require           close-output-port       peek-char
#%provide           with-output-to-file     eof-object?
#%expression        flush-output            char-ready?
syntax-rules        string-length           write
...                 string-ci<=?            display
cons                string-ci>=?            newline
car                 string-append           write-char
cdr                 string-fill!            load
pair?               string->list            string?
map                 list->string            string
for-each            vector-length           string-ref
caar                vector-fill!            string-set!
cadr                vector->list            string=?
cdar                list->vector            substring
cddr                char-alphabetic?        string-copy
caaar               char-numeric?           string-ci=?
caadr               char-whitespace?        string<?
cadar               char-upper-case?        string>?
caddr               char-lower-case?        string<=?
cdaar               char->integer           string>=?
cdadr               integer->char           string-ci<?
cddar               char-downcase           string-ci>?
cdddr               call-with-output-file   vector?
caaaar              call-with-input-file    make-vector
caaadr              with-input-from-file    vector
caadar              apply                   vector-ref
caaddr              symbol?                 vector-set!
cadaar              null?                   char?
cadadr              list?                   char=?
caddar              list                    char<?
cadddr              length                  char>?
cdaaar              append                  char<=?
cdaadr              reverse                 char>=?
cdadar              list-tail               char-ci=?
cdaddr              list-ref                char-ci<?
cddaar              memq                    char-ci>?
cddadr              memv                    char-ci<=?
cdddar              member                  char-ci>=?
cddddr              assq                    char-upcase
=                   assv                    boolean?
<                   assoc                   eqv?
```

```
>                 procedure?          equal?
<=                number?             force
>=                complex?            call-with-values
max               real?               values
min               rational?           dynamic-wind
+                 integer?            eval
-                 exact?
```

```
(define-datatype id predicate-id
  (variant-id (field-id predicate-expr) ...)
  ...)
```

Defines the datatype *id* and a function *predicate-id* that returns `#t` for instances of the datatype, and `#f` for any other value.

Each *variant-id* is defined as a constructor function that creates an instance of the datatype; the constructor takes as many arguments as the variant's *field-id*s, and each argument is checked by applying the function produced by the variant's *predicate-expr*.

In DrScheme v209 and older, when constructor-based printing was used, variant instances were printed with a `make-` prefix before the variant name. Thus, for compatibility, in addition to *variant-id*, `make-variant-id` is also defined for each *variant-id* (to the same constructor as *variant-id*).

```
(cases datatype-id expr
  (variant-id (field-id ...) result-expr ...)
  ...)
(cases datatype-id expr
  (variant-id (field-id ...) result-expr ...)
  ...
  (else result-expr ...))
```

Branches on the datatype instance produced by *expr*, which must be an instance of the specified *datatype-id* that is defined with `define-datatype`.

```
sllgen:make-string-scanner
sllgen:make-string-parser
sllgen:make-stream-parser
sllgen:make-define-datatypes
sllgen:show-define-datatypes
sllgen:list-define-datatypes
```

Defined in the textbook's Appendix B [EoPL]. However, the DrRacket versions are syntactic forms, instead of procedures, and the arguments must be either quoted literal tables or identifiers that are defined (at the top level) to quoted literal tables.

```
sllgen:make-rep-loop : procedure?
```

Defined in the *EoPL* textbook's Appendix B [EoPL] (and still a function).

```
eopl:error : procedure?
```

As in the book.

```
(eopl:printf form v ...) → void?
  form : string?
  v : any/c
(eopl:pretty-print v [port]) → void?
  v : any/c
  port : output-port? = (current-output-port)
```

Same as `scheme/base`'s `printf` and `pretty-print`.

```
((list-of pred ...+) x) → boolean?
  pred : (any/c . -> . any)
  x : any/c
(always? x) → boolean?
  x : any/c
(maybe pred) → boolean?
  pred : (any/c . -> . boolean?)
```

As in the book [EoPL].

```
empty : empty?
```

The empty list.

```
(time expr)
```

Evaluates *expr*, and prints timing information before returning the result.

```
(collect-garbage) → void?
```

Performs a garbage collection (useful for repeatable timings).

```
(trace id ...)
(untrace id ...)
```

For debugging: `trace` redefines each *id* at the top level (bound to a procedure) so that it prints arguments on entry and results on exit. The `untrace` form reverses the action of `trace` for the given *id*s.

Tracing a function causes tail-calls in the original function to become non-tail calls.

```
(provide provide-spec ...)
```

Useful only with a module that uses `eopl` as a language: exports identifiers from the module. See `provide` from `racket` for more information.

```
eopl:error-stop : (-> any/c)
```

Defined only in the top-level namespace (i.e., not in a module); mutate this variable to install an exception-handling thunk. Typically, the handler thunk escapes through a continuation.

The `eopl` library sets this variable to `#f` in the current namespace when it executes.

```
(install-eopl-exception-handler) → void?
```

Sets an exception handler to one that checks `eopl:error-stop`.

The `eopl` library calls this function when it executes.

Bibliography

[EoPL] "*Essentials of Programming Languages*, Third Edition," MIT Press, 2008.
`http://www.eopl3.com/`