Essentials of Programming Languages Language

Version 5.3.2

January 31, 2013

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
```

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

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

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

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