Unstable

Version 5.0

June 6, 2010

```
(require unstable)
```

This manual documents some of the libraries available in the `unstable` collection.

The name `unstable` is intended as a warning that the **interfaces** in particular are unstable. Developers of planet packages and external projects should avoid using modules in the unstable collection. Contracts may change, names may change or disappear, even entire modules may move or disappear without warning to the outside world.

Developers of unstable libraries must follow the guidelines in §1 “Guidelines for developing `unstable` libraries”.

1 Guidelines for developing `unstable` libraries

Any collection developer may add modules to the `unstable` collection.

Every module needs an owner to be responsible for it.

- If you add a module, you are its owner. Add a comment with your name at the top of the module.
- If you add code to someone else's module, tag your additions with your name. The module's owner may ask you to move your code to a separate module if they don't wish to accept responsibility for it.

When changing a library, check all uses of the library in the collections tree and update them if necessary. Notify users of major changes.

Place new modules according to the following rules. (These rules are necessary for maintaining PLT's separate text, gui, and drracket distributions.)

- Non-GUI modules go under `unstable` (or subcollections thereof). Put the documentation in `unstable/scribblings` and include with `include-section` from `unstable/scribblings/unstable.scrbl`.
- GUI modules go under `unstable/gui`. Put the documentation in `unstable/scribblings/gui` and include them with `include-section` from `unstable/scribblings/gui.scrbl`.
- Do not add modules depending on DrRacket to the `unstable` collection.
- Put tests in `tests/unstable`.

Keep documentation and tests up to date.

2 Bytes

```
(require unstable/bytes)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(bytes-ci=? b1 b2) → boolean?
  b1 : bytes?
  b2 : bytes?
```

Compares two bytes case insensitively.

```
(read/bytes b) → serializable?
  b : bytes?
```

reads a value from `b` and returns it.

```
(write/bytes v) → bytes?
  v : serializable?
```

writes `v` to a bytes and returns it.

3 Contracts

```
(require unstable/contract)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

`non-empty-string/c : contract?`

Contract for non-empty strings.

`port-number? : contract?`

Equivalent to `(between/c 1 65535)`.

`path-element? : contract?`

Equivalent to `(or/c path-string? (symbols 'up 'same))`.

The subsequent bindings were added by Ryan Culpepper.

```
(if/c predicate then-contract else-contract) → contract?
  predicate : (-> any/c any/c)
  then-contract : contract?
  else-contract : contract?
```

Produces a contract that, when applied to a value, first tests the value with *predicate*; if *predicate* returns true, the *then-contract* is applied; otherwise, the *else-contract* is applied. The resulting contract is a flat contract if both *then-contract* and *else-contract* are flat contracts.

For example, the following contract enforces that if a value is a procedure, it is a thunk; otherwise it can be any (non-procedure) value:

```
(if/c procedure? (-> any) any/c)
```

Note that the following contract is **not** equivalent:

```
(or/c (-> any) any/c) ; wrong!
```

The last contract is the same as `any/c` because `or/c` tries flat contracts before higher-order contracts.

```
(rename-contract contract name) → contract?
  contract : contract?
```

name : any/c

Produces a contract that acts like *contract* but with the name *name*.

The resulting contract is a flat contract if *contract* is a flat contract.

4 Directories

```
(require unstable/dirs)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

This library defines utilities dealing with the directory paths used by the Racket distribution.

```
(path->directory-relative-string  path
                                  [#:default default
                                   #:dirs dirs])
 → (or/c string? (one-of/c default))
  path : path-string?
  default : any/c = (if (path? path) (path->string path) path)
  dirs :  (listof (cons/c (-> path?) any/c))
       = library-relative-directories
```

Produces a string rendering of *path*, replacing distribution-specific paths (normally: collections, user-installed collections, or PLanet cache) with short abbreviations.

The set of paths and their abbreviations may be overridden by the `#:dirs` option, which accepts an association list. Its keys must be thunks which produce a path. Its values may be either `#f` for no abbreviation (the directory prefix is simply omitted) or any other value to be displayed in the output. For instance, `"document.txt"` relative to a path abbreviated `"path"` would be rendered as `"<path>/document.txt"`.

If the path is not relative to one of the given directories, the default return value is a string rendering of the unmodified path. This default may be overridden by providing *default*.

Examples:

```
> (path->directory-relative-string
   (build-path "source" "project.rkt"))
"source/project.rkt"
> (path->directory-relative-string
   (build-path (current-directory) "source" "project.rkt"))
"<collects>/unstable/source/project.rkt"
> (path->directory-relative-string
   (build-path "/" "source" "project.rkt"))
"/source/project.rkt"
> (path->directory-relative-string
   (build-path "/" "source" "project.rkt")
   #:default #f)
#f
> (path->directory-relative-string
```

```
    (build-path "/" "source" "project.rkt")
    #:dirs (list
            (cons (lambda () (build-path "/" "source"))
                  'src)))
  "<src>/project.rkt"
```

`library-relative-directories` : `(listof (cons (-> path?) any/c))`

Represents the default directory substitutions for `path->directory-relative-string`. By default, the collections directory is replaced by `collects`, the user-installed collections directory is replaced by `user`, and the PLaneT cache is replaced by `planet`.

`setup-relative-directories` : `(listof (cons (-> path?) any/c))`

Represents the directory substitutions used by `setup-plt`. The collections directory is omitted, the user-installed collections directory is replaced by `user`, and the PLaneT cache is replaced by `planet`.

5 Exceptions

```
(require unstable/exn)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(network-error s fmt v ...) → void
  s : symbol?
  fmt : string?
  v : any/c
```

Like `error`, but throws a `exn:fail:network`.

```
(exn->string exn) → string?
  exn : (or/c exn? any/c)
```

Formats *exn* with `(error-display-handler)` as a string.

6 Filesystem

```
(require unstable/file)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(make-directory*/ignore-exists-exn pth) → void
  pth : path-string?
```

Like `make-directory*`, except it ignores errors when the path already exists. Useful to deal with race conditions on processes that create directories.

7 Lists

```
(require unstable/list)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(list-prefix? l r) → boolean?
  l : list?
  r : list?
```

True if `l` is a prefix of `r`.

Example:

```
> (list-prefix? '(1 2) '(1 2 3 4 5))
#t
```

The subsequent bindings were added by Sam Tobin-Hochstadt.

```
(filter-multiple l f ...) → list? ...
  l : list?
  f : procedure?
```

Produces `(values (filter f l) ...)`.

Example:

```
> (filter-multiple (list 1 2 3 4 5) even? odd?)
'(2 4)
'(1 3 5)
```

```
(extend l1 l2 v) → list?
  l1 : list?
  l2 : list?
  v : any/c
```

Extends `l2` to be as long as `l1` by adding `(- (length l1) (length l2))` copies of `v` to the end of `l2`.

Example:

```
> (extend '(1 2 3) '(a) 'b)
'(a b b)
```

The subsequent bindings were added by Ryan Culpepper.

```
(check-duplicate  lst
                  [#:key extract-key
                   #:same? same?])    → (or/c any/c #f)
  lst : list?
```

```
  extract-key : (-> any/c any/c) = (lambda (x) x)
  same? : (or/c (any/c any/c . -> . any/c) = equal?
                dict?)
```

Returns the first duplicate item in *lst*. More precisely, it returns the first *x* such that there was a previous *y* where (*same?* (*extract-key x*) (*extract-key y*)).

The *same?* argument can either be an equivalence predicate such as equal? or eqv? or a dictionary. In the latter case, the elements of the list are mapped to #t in the dictionary until an element is discovered that is already mapped to a true value. The procedures equal?, eqv?, and eq? automatically use a dictionary for speed.

Examples:

```
  > (check-duplicate '(1 2 3 4))
  #f
  > (check-duplicate '(1 2 3 2 1))
  2
  > (check-duplicate '((a 1) (b 2) (a 3)) #:key car)
  '(a 3)
  > (define id-t (make-free-id-table))
  > (check-duplicate (syntax->list #'(a b c d a b))
                     #:same? id-t)
  #<syntax:10:0 a>
  > (dict-map id-t list)
  '((#<syntax:10:0 a> #t) (#<syntax:10:0 b> #t) (#<syntax:10:0 c> #t)
(#<syntax:10:0 d> #t))
```

The subsequent bindings were added by Carl Eastlund.

```
(map/values n f lst ...) → (listof B_1) ... (listof B_n)
  n : natural-number/c
  f : (-> A ... (values B_1 ... B_n))
  lst : (listof A)
```

Produces lists of the respective values of *f* applied to the elements in *lst* ... sequentially.

Example:

```
  > (map/values
     3
     (lambda (x)
       (values (+ x 1) x (- x 1)))
     (list 1 2 3))
  '(2 3 4)
  '(1 2 3)
  '(0 1 2)
```

8 Net

```
(require unstable/net)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

8.1 URLs

```
(require unstable/net/url)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(url-replace-path proc u) → url?
  proc : ((listof path/param?) . -> . (listof path/param?))
  u : url?
```

Replaces the URL path of *u* with *proc* of the former path.

```
(url-path->string url-path) → string?
  url-path : (listof path/param?)
```

Formats *url-path* as a string with "/" as a delimiter and no params.

9 Path

```
(require unstable/path)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(explode-path* p) → (listof path-element?)
  p : path-string?
```

Like `normalize-path`, but does not resolve symlinks.

```
(path-without-base base p) → (listof path-element?)
  base : path-string?
  p : path-string?
```

Returns, as a list, the portion of *p* after *base*, assuming *base* is a prefix of *p*.

```
(directory-part p) → path?
  p : path-string?
```

Returns the directory part of *p*, returning `(current-directory)` if it is relative.

```
(build-path-unless-absolute base p) → path?
  base : path-string?
  p : path-string?
```

Prepends *base* to *p*, unless *p* is absolute.

```
(strip-prefix-ups p) → (listof path-element?)
  p : (listof path-element?)
```

Removes all the prefix `".."`s from *p*.

10 Source Locations

There are two libraries in this collection for dealing with source locations; one for manipulating representations of them, and the other for quoting the location of a particular piece of source code.

10.1 Representations

```
(require unstable/srcloc)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

This module defines utilities for manipulating representations of source locations, including both `srcloc` structures and all the values accepted by `datum->syntax`'s third argument: syntax objects, lists, vectors, and `#f`.

```
(source-location? x) → boolean?
  x : any/c
(source-location-list? x) → boolean?
  x : any/c
(source-location-vector? x) → boolean?
  x : any/c
```

These functions recognize valid source location representations. The first, `source-location?`, recognizes `srcloc` structures, syntax objects, lists, and vectors with appropriate structure, as well as `#f`. The latter predicates recognize only valid lists and vectors, respectively.

Examples:

```
> (source-location? #f)
#t
> (source-location? #'here)
#t
> (source-location? (make-srcloc 'here 1 0 1 0))
#t
> (source-location? (make-srcloc 'bad 1 #f 1 0))
#f
> (source-location? (list 'here 1 0 1 0))
#t
> (source-location? (list* 'bad 1 0 1 0 'tail))
#f
> (source-location? (vector 'here 1 0 1 0))
```

```
#t
> (source-location? (vector 'bad 0 0 0 0))
#f
```

```
(check-source-location! name x) → void?
  name : symbol?
  x : any/c
```

This procedure checks that its input is a valid source location. If it is, the procedure returns `(void)`. If it is not, `check-source-location!` raises a detailed error message in terms of *name* and the problem with *x*.

Examples:

```
  > (check-source-location! 'this-example #f)
  > (check-source-location! 'this-example #'here)
  > (check-source-location! 'this-example (make-
srcloc 'here 1 0 1 0))
  > (check-source-location! 'this-example (make-
srcloc 'bad 1 #f 1 0))
  this-example: expected a source location with line number
  and column number both numeric or both #f; got 1 and #f
  respectively: (srcloc 'bad 1 #f 1 0)
  > (check-source-location! 'this-example (list 'here 1 0 1 0))
  > (check-source-location! 'this-example (list* 'bad 1 0 1 0 'tail))
  this-example: expected a source location (a list of 5
  elements); got an improper list: '(bad 1 0 1 0 . tail)
  > (check-source-location! 'this-example (vector 'here 1 0 1 0))
  > (check-source-location! 'this-example (vector 'bad 0 0 0 0))
  this-example: expected a source location with a positive
  line number or #f (second element); got line number 0:
  '#(bad 0 0 0 0)
```

```
(build-source-location loc ...) → srcloc?
  loc : source-location?
(build-source-location-list loc ...) → source-location-list?
  loc : source-location?
(build-source-location-vector loc ...) → source-location-vector?
  loc : source-location?
(build-source-location-syntax loc ...) → syntax?
  loc : source-location?
```

These procedures combine multiple (zero or more) source locations, merging locations within the same source and reporting `#f` for locations that span sources. They also convert the result to the desired representation: `srcloc`, list, vector, or syntax object, respectively.

Examples:

```
> (build-source-location)
(srcloc #f #f #f #f #f)
> (build-source-location-list)
'(#f #f #f #f #f)
> (build-source-location-vector)
'#(#f #f #f #f #f)
> (build-source-location-syntax)
#<syntax ()>
> (build-source-location #f)
(srcloc #f #f #f #f #f)
> (build-source-location-list #f)
'(#f #f #f #f #f)
> (build-source-location-vector #f)
'#(#f #f #f #f #f)
> (build-source-location-syntax #f)
#<syntax ()>
> (build-source-location (list 'here 1 2 3 4))
(srcloc here 1 2 3 4)
> (build-source-location-list (make-srcloc 'here 1 2 3 4))
'(here 1 2 3 4)
> (build-source-location-vector (make-srcloc 'here 1 2 3 4))
'#(here 1 2 3 4)
> (build-source-location-syntax (make-srcloc 'here 1 2 3 4))
#<syntax:1:2 ()>
> (build-source-location (list 'here 1 2 3 4) (vector 'here 5 6 7 8))
(srcloc here 1 2 3 12)
> (build-source-location-list (make-srcloc 'here 1 2 3 4) (vector 'here 5 6 7 8))
'(here 1 2 3 12)
> (build-source-location-vector (make-srcloc 'here 1 2 3 4) (vector 'here 5 6 7 8))
'#(here 1 2 3 12)
> (build-source-location-syntax (make-srcloc 'here 1 2 3 4) (vector 'here 5 6 7 8))
#<syntax:1:2 ()>
> (build-source-location (list 'here 1 2 3 4) (vector 'there 5 6 7 8))
(srcloc #f #f #f #f #f)
> (build-source-location-list (make-srcloc 'here 1 2 3 4) (vector 'there 5 6 7 8))
'(#f #f #f #f #f)
> (build-source-location-vector (make-srcloc 'here 1 2 3 4) (vector 'there 5 6 7 8))
'#(#f #f #f #f #f)
> (build-source-location-syntax (make-srcloc 'here 1 2 3 4) (vector 'there 5 6 7 8))
#<syntax ()>
```

(source-location-known? *loc*) → boolean?
  loc : source-location?

This predicate reports whether a given source location contains more information than simply `#f`.

Examples:

```
> (source-location-known? #f)
#f
> (source-location-known? (make-srcloc #f #f #f #f #f))
#f
> (source-location-known? (make-srcloc 'source 1 2 3 4))
#t
> (source-location-known? (list #f #f #f #f #f))
#f
> (source-location-known? (vector 'source #f #f #f #f))
#t
> (source-location-known? (datum->syntax #f null #f))
#t
> (source-location-known? (datum->syntax #f null (list 'source #f #f #f #f)))
#t
```

```
(source-location-source loc) → any/c
  loc : source-location?
(source-location-line loc)
 → (or/c orexact-positive-integer? #f)
  loc : source-location?
(source-location-column loc)
 → (or/c exact-nonnegative-integer? #f)
  loc : source-location?
(source-location-position loc)
 → (or/c exact-positive-integer? #f)
  loc : source-location?
(source-location-span loc)
 → (or/c exact-nonnegative-integer? #f)
  loc : source-location?
```

These accessors extract the fields of a source location.

Examples:

```
> (source-location-source #f)
#f
> (source-location-line (make-srcloc 'source 1 2 3 4))
1
> (source-location-column (list 'source 1 2 3 4))
2
> (source-location-position (vector 'source 1 2 3 4))
3
> (source-location-span (datum->syntax #f null (list 'source 1 2 3 4)))
```

```
4
```

```
(source-location-end loc)
 → (or/c exact-nonnegative-integer? #f)
  loc : source-location?
```

This accessor produces the end position of a source location (the sum of its position and span, if both are numbers) or `#f`.

Examples:

```
> (source-location-end #f)
#f
> (source-location-end (make-srcloc 'source 1 2 3 4))
7
> (source-location-end (list 'source 1 2 3 #f))
#f
> (source-location-end (vector 'source 1 2 #f 4))
#f
```

```
(update-source-location loc
                        #:source source
                        #:line line
                        #:column column
                        #:position position
                        #:span span)        → source-location?
  loc : source-location?
  source : any/c
  line : (or/c exact-nonnegative-integer? #f)
  column : (or/c exact-positive-integer? #f)
  position : (or/c exact-nonnegative-integer? #f)
  span : (or/c exact-positive-integer? #f)
```

Produces a modified version of *loc*, replacing its fields with *source*, *line*, *column*, *position*, and/or *span*, if given.

Examples:

```
> (update-source-location #f #:source 'here)
'(here #f #f #f #f)
> (update-source-location (list 'there 1 2 3 4) #:line 20 #:column 79)
'(there 20 79 3 4)
> (update-source-location (vector 'everywhere 1 2 3 4) #:position #f #:span #f)
'#(everywhere 1 2 #f #f)
```

```
(source-location->string loc) → string?
  loc : source-location?
(source-location->prefix loc) → string?
  loc : source-location?
```

These procedures convert source locations to strings for use in error messages. The first produces a string describing the source location; the second appends ": " to the string if it is non-empty.

Examples:

```
> (source-location->string (make-srcloc 'here 1 2 3 4))
"here:1.2"
> (source-location->string (make-srcloc 'here #f #f 3 4))
"here::3-7"
> (source-location->string (make-srcloc 'here #f #f #f #f))
"here"
> (source-location->string (make-srcloc #f 1 2 3 4))
":1.2"
> (source-location->string (make-srcloc #f #f #f 3 4))
"::3-7"
> (source-location->string (make-srcloc #f #f #f #f #f))
""
> (source-location->prefix (make-srcloc 'here 1 2 3 4))
"here:1.2: "
> (source-location->prefix (make-srcloc 'here #f #f 3 4))
"here::3-7: "
> (source-location->prefix (make-srcloc 'here #f #f #f #f))
"here: "
> (source-location->prefix (make-srcloc #f 1 2 3 4))
":1.2: "
> (source-location->prefix (make-srcloc #f #f #f 3 4))
"::3-7: "
> (source-location->prefix (make-srcloc #f #f #f #f #f))
""
```

10.2 Quoting

```
(require unstable/location)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

This module defines macros that evaluate to various aspects of their own source location.

Note: The examples below illustrate the use of these macros and the representation of their output. However, due to the mechanism by which they are generated, each example is considered a single character and thus does not have realistic line, column, and character positions.

Furthermore, the examples illustrate the use of source location quoting inside macros, and the difference between quoting the source location of the macro definition itself and quoting the source location of the macro's arguments.

```
(quote-srcloc)
(quote-srcloc form)
(quote-srcloc form #:module-source expr)
```

Quotes the source location of *form* as a `srcloc` structure, using the location of the whole `(quote-srcloc)` expression if no *expr* is given. Uses relative directories for paths found within the collections tree, the user's collections directory, or the PLaneT cache.

Examples:

```
> (quote-srcloc)
(srcloc eval 2 0 2 1)
> (define-syntax (not-here stx) #'(quote-srcloc))
> (not-here)
(srcloc eval 3 0 3 1)
> (not-here)
(srcloc eval 3 0 3 1)
> (define-syntax (here stx) #`(quote-srcloc #,stx))
> (here)
(srcloc eval 7 0 7 1)
> (here)
(srcloc eval 8 0 8 1)
```

```
(quote-source-file)
(quote-source-file form)
(quote-line-number)
(quote-line-number form)
(quote-column-number)
(quote-column-number form)
(quote-character-position)
(quote-character-position form)
(quote-character-span)
(quote-character-span form)
```

Quote various fields of the source location of *form*, or of the whole macro application if no *form* is given.

Examples:

```
> (list (quote-source-file)
        (quote-line-number)
        (quote-column-number)
        (quote-character-position)
        (quote-character-span))
'(eval 2 0 2 1)
> (define-syntax (not-here stx)
    #'(list (quote-source-file)
            (quote-line-number)
            (quote-column-number)
            (quote-character-position)
            (quote-character-span)))
> (not-here)
'(eval 3 0 3 1)
> (not-here)
'(eval 3 0 3 1)
> (define-syntax (here stx)
    #`(list (quote-source-file #,stx)
            (quote-line-number #,stx)
            (quote-column-number #,stx)
            (quote-character-position #,stx)
            (quote-character-span #,stx)))
> (here)
'(eval 7 0 7 1)
> (here)
'(eval 8 0 8 1)
```

```
(quote-module-name)
(quote-module-path)
```

Quote the name of the module in which the form is compiled. The `quote-module-name` form produces a string or a symbol, while `quote-module-path` produces a module path.

These forms use relative names for modules found in the collections or PLaneT cache; their results are suitable for printing, but not for accessing libraries programmatically, such as via `dynamic-require`.

Examples:

```
> (module A racket
    (require unstable/location)
    (define-syntax-rule (name) (quote-module-name))
    (define-syntax-rule (path) (quote-module-path))
    (define a-name (name))
    (define a-path (path))
```

```
    (provide (all-defined-out)))
> (require 'A)
> a-name
'A
> a-path
''A
> (module B racket
    (require unstable/location)
    (require 'A)
    (define b-name (name))
    (define b-path (path))
    (provide (all-defined-out)))
> (require 'B)
> b-name
'B
> b-path
''B
> (quote-module-name)
'top-level
> (quote-module-path)
'top-level
> [current-namespace (module->namespace ''A)]
> (quote-module-name)
'A
> (quote-module-path)
''A
```

11 Strings

```
(require unstable/string)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(lowercase-symbol! sb) → symbol?
  sb : (or/c string? bytes?)
```

Returns `sb` as a lowercase symbol.

```
(read/string s) → serializable?
  s : string?
```

reads a value from `s` and returns it.

```
(write/string v) → string?
  v : serializable?
```

writes `v` to a string and returns it.

12 Structs

```
(require unstable/struct)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(make struct-id expr ...)
```

Creates an instance of *struct-id*, which must be bound as a struct name. The number of *expr*s is statically checked against the number of fields associated with *struct-id*. If they are different, or if the number of fields is not known, an error is raised at compile time.

Examples:

```
> (define-struct triple (a b c))
> (make triple 3 4 5)
#<triple>
> (make triple 2 4)
eval:4:0: make: wrong number of arguments for struct triple
(expected 3, got 2) in: (make triple 2 4)
```

```
(struct->list v [#:on-opaque on-opaque]) → (or/c list? #f)
  v : any/c
  on-opaque : (or/c 'error 'return-false 'skip) = 'error
```

Returns a list containing the struct instance *v*'s fields. Unlike `struct->vector`, the struct name itself is not included.

If any fields of *v* are inaccessible via the current inspector the behavior of `struct->list` is determined by *on-opaque*. If *on-opaque* is `'error` (the default), an error is raised. If it is `'return-false`, `struct->list` returns `#f`. If it is `'skip`, the inaccessible fields are omitted from the list.

Examples:

```
> (define-struct open (u v) #:transparent)
> (struct->list (make-open 'a 'b))
'(a b)
> (struct->list #s(pre 1 2 3))
'(1 2 3)
> (define-struct (secret open) (x y))
> (struct->list (make-secret 0 1 17 22))
struct->list: expected argument of type <non-opaque
struct>; given (secret 0 1 ...)
> (struct->list (make-secret 0 1 17 22) #:on-opaque 'return-false)
#f
```

```
> (struct->list (make-secret 0 1 17 22) #:on-opaque 'skip)
'(0 1)
> (struct->list 'not-a-struct #:on-opaque 'return-false)
#f
> (struct->list 'not-a-struct #:on-opaque 'skip)
'()
```

13 Syntax

```
(require unstable/syntax)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(current-syntax-context) → (or/c syntax? false/c)
(current-syntax-context stx) → void?
  stx : (or/c syntax? false/c)
```

The current contextual syntax object, defaulting to `#f`. It determines the special form name that prefixes syntax errors created by `wrong-syntax`.

```
(wrong-syntax stx format-string v ...) → any
  stx : syntax?
  format-string : string?
  v : any/c
```

Raises a syntax error using the result of `(current-syntax-context)` as the "major" syntax object and the provided `stx` as the specific syntax object. (The latter, `stx`, is usually the one highlighted by DrRacket.) The error message is constructed using the format string and arguments, and it is prefixed with the special form name as described under `current-syntax-context`.

Examples:

```
> (wrong-syntax #'here "expected ~s" 'there)
?: expected there
> (parameterize ((current-syntax-context #'(look over here)))
    (wrong-syntax #'here "expected ~s" 'there))
eval:4:0: look: expected there at: here in: (look over here)
```

A macro using `wrong-syntax` might set the syntax context at the very beginning of its transformation as follows:

```
(define-syntax (my-macro stx)
  (parameterize ((current-syntax-context stx))
    (syntax-case stx ()
      __)))
```

Then any calls to `wrong-syntax` during the macro's transformation will refer to `my-macro` (more precisely, the name that referred to `my-macro` where the macro was used, which may be different due to renaming, prefixing, etc).

(define/with-syntax *pattern expr*)

Definition form of with-syntax. That is, it matches the syntax object result of *expr* against *pattern* and creates pattern variable definitions for the pattern variables of *pattern*.

Examples:

```
> (define/with-syntax (px ...) #'(a b c))
> (define/with-syntax (tmp ...) (generate-temporaries #'(px ...)))
> #'([tmp px] ...)
#<syntax:7:0 ((a5 a) (b6 b) (c7 c))>
```

(define-pattern-variable *id expr*)

Evaluates *expr* and binds it to *id* as a pattern variable, so *id* can be used in subsequent syntax patterns.

Examples:

```
> (define-pattern-variable name #'Alice)
> #'(hello name)
#<syntax:9:0 (hello Alice)>
```

(with-temporaries (*temp-id* ...) . *body*)

Evaluates *body* with each *temp-id* bound as a pattern variable to a freshly generated identifier.

Example:

```
> (with-temporaries (x) #'(lambda (x) x))
#<syntax:10:0 (lambda (x8) x8)>
```

(generate-temporary [*name-base*]) → identifier?
  name-base : any/c = 'g

Generates one fresh identifier. Singular form of generate-temporaries. If *name-base* is supplied, it is used as the basis for the identifier's name.

(generate-n-temporaries *n*) → (listof identifier?)
  n : exact-nonnegative-integer?

Generates a list of *n* fresh identifiers.

(current-caught-disappeared-uses)

```
 → (or/c (listof identifier?) false/c)
(current-caught-disappeared-uses ids) → void?
  ids : (or/c (listof identifier?) false/c)
```

Parameter for tracking disappeared uses. Tracking is "enabled" when the parameter has a non-false value. This is done automatically by forms like `with-disappeared-uses`.

```
(with-disappeared-uses stx-expr)

  stx-expr : syntax?
```

Evaluates the *stx-expr*, catching identifiers looked up using `syntax-local-value/catch`. Adds the caught identifiers to the `'disappeared-uses` syntax property of the resulting syntax object.

```
(with-catching-disappeared-uses body-expr)
```

Evaluates the *body-expr*, catching identifiers looked up using `syntax-local-value/catch`. Returns two values: the result of *body-expr* and the list of caught identifiers.

```
(syntax-local-value/catch id predicate) → any/c
  id : identifier?
  predicate : (-> any/c boolean?)
```

Looks up *id* in the syntactic environment (as `syntax-local-value`). If the lookup succeeds and returns a value satisfying the predicate, the value is returned and *id* is recorded ("caught") as a disappeared use. If the lookup fails or if the value does not satisfy the predicate, `#f` is returned and the identifier is not recorded as a disappeared use.

If not used within the extent of a `with-disappeared-uses` form or similar, has no effect.

```
(record-disappeared-uses ids) → void?
  ids : (listof identifier?)
```

Add *ids* to the current disappeared uses.

If not used within the extent of a `with-disappeared-uses` form or similar, has no effect.

```
(format-symbol fmt v ...) → symbol?
  fmt : string?
  v : (or/c string? symbol? identifier? keyword? char? number?)
```

Like format, but produces a symbol. The format string must use only ~a placeholders. Identifiers in the argument list are automatically converted to symbols.

Example:

```
> (format-symbol "make-~a" 'triple)
'make-triple
```

```
(format-id  lctx
            [#:source src
             #:props props
             #:cert cert]
             fmt
             v ...)        → identifier?
  lctx : (or/c syntax? #f)
  src : (or/c syntax? #f) = #f
  props : (or/c syntax? #f) = #f
  cert : (or/c syntax? #f) = #f
  fmt : string?
  v : (or/c string? symbol? identifier? keyword? char? number?)
```

Like format-symbol, but converts the symbol into an identifier using *lctx* for the lexical context, *src* for the source location, *props* for the properties, and *cert* for the inactive certificates. (See datum->syntax.)

The format string must use only ~a placeholders. Identifiers in the argument list are automatically converted to symbols.

Examples:

```
> (define-syntax (make-pred stx)
    (syntax-case stx ()
      [(make-pred name)
       (format-id #'name "~a?" (syntax-e #'name))]))
> (make-pred pair)
#<procedure:pair?>
> (make-pred none-such)
reference to undefined identifier: none-such?
> (define-syntax (better-make-pred stx)
    (syntax-case stx ()
      [(better-make-pred name)
       (format-id #'name #:source #'name
                  "~a?" (syntax-e #'name))]))
> (better-make-pred none-such)
reference to undefined identifier: none-such?
```

(Scribble doesn't show it, but the DrRacket pinpoints the location of the second error but not of the first.)

```
(internal-definition-context-apply intdef-ctx
                                   stx)       → syntax?
  intdef-ctx : internal-definition-context?
  stx : syntax?
```

Applies the renamings of *intdef-ctx* to *stx*.

```
(syntax-local-eval stx [intdef-ctx]) → any
  stx : syntax?
  intdef-ctx : (or/c internal-definition-context? #f) = #f
```

Evaluates *stx* as an expression in the current transformer environment (that is, at phase level 1), optionally extended with *intdef-ctx*.

Examples:

```
> (define-syntax (show-me stx)
    (syntax-case stx ()
      [(show-me expr)
       (begin
         (printf "at compile time produces ~s\n"
                 (syntax-local-eval #'expr))
         #'(printf "at run time produes ~s\n"
                   expr))]))
> (show-me (+ 2 5))
at compile time produces 7
at run time produes 7
> (define-for-syntax fruit 'apple)
> (define fruit 'pear)
> (show-me fruit)
at compile time produces apple
at run time produes pear
```

The subsequent bindings were added by Sam Tobin-Hochstadt.

```
(with-syntax* ([pattern stx-expr] ...)
  body ...+)
```

Similar to `with-syntax`, but the pattern variables are bound in the remaining *stx-expr*s as well as the *body*s, and the *pattern*s need not bind distinct pattern variables; later bindings shadow earlier bindings.

Example:

```
> (with-syntax* ([(x y) (list #'val1 #'val2)]
                 [nest #'((x) (y))])
    #'nest)
```

```
#<syntax:22:0 ((val1) (val2))>
```

```
(syntax-map f stxl ...) → (listof A)
  f : (-> syntax? A)
  stxl : syntax?
```

Performs `(map f (syntax->list stxl) ...)`.

Example:

```
> (syntax-map syntax-e #'(a b c))
'(a b c)
```

14 Polymorphic Contracts

```
(require unstable/poly-c)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

`(poly/c (x ...) c)`

Creates a contract for polymorphic functions that may inspect their arguments. Each function is protected by `c`, where each `x` is bound in `c` and refers to a polymorphic type that is instantiated each time the function is applied.

At each application of a function, the `poly/c` contract constructs a new weak, `eq?`-based hash table for each `x`. Values flowing into the polymorphic function (i.e. values protected by some `x` in negative position with respect to `poly/c`) are stored in the hash table. Values flowing out of the polymorphic function (i.e. protected by some `x` in positive position with respect to `poly/c`) are checked for their presence in the hash table. If they are present, they are returned; otherwise, a contract violation is signalled.

Examples:

```
> (define/contract (check x y) (poly/c [X] (boolean? X . -> . X))
    (if (or (not x) (equal? y 'surprise))
        'invalid
        y))
> (check #t 'ok)
'ok
> (check #f 'ignored)
eval:2.0: (function check) broke the contract (poly/c (X)
...) on check; expected a(n) X; got: 'invalid
> (check #t 'surprise)
eval:2.0: (function check) broke the contract (poly/c (X)
...) on check; expected a(n) X; got: 'invalid
```

`(parametric/c (x ...) c)`

Creates a contract for parametric polymorphic functions. Each function is protected by `c`, where each `x` is bound in `c` and refers to a polymorphic type that is instantiated each time the function is applied.

At each application of a function, the `parametric/c` contract constructs a new opaque wrapper for each `x`; values flowing into the polymorphic function (i.e. values protected by some `x` in negative position with respect to `parametric/c`) are wrapped in the corresponding opaque wrapper. Values flowing out of the polymorphic function (i.e. values protected by some `x` in positive position with respect to `parametric/c`) are checked for the appro-

priate wrapper. If they have it, they are unwrapped; if they do not, a contract violation is signalled.

Examples:

```
> (define/contract (check x y) (parametric/c [X] (boolean? X . ->
. X))
    (if (or (not x) (equal? y 'surprise))
        'invalid
        y))
> (check #t 'ok)
'ok
> (check #f 'ignored)
eval:2.0: (function check) broke the contract (parametric/c
(X) ...) on check; expected a(n) X; got: 'invalid
> (check #t 'surprise)
'surprise
```

(memory/c *positive?* *name*) → contract?
  positive? : boolean?
  name : any/c

This function constructs a contract that records values flowing in one direction in a fresh, weak hash table, and looks up values flowing in the other direction, signalling a contract violation if those values are not in the table.

If *positive?* is true, values in positive position get stored and values in negative position are checked. Otherwise, the reverse happens.

(opaque/c *positive?* *name*) → contract?
  positive? : boolean?
  name : any/c

This function constructs a contract that wraps values flowing in one direction in a unique, opaque wrapper, and unwraps values flowing in the other direction, signalling a contract violation if those values are not wrapped.

If *positive?* is true, values in positive position get wrapped and values in negative position get unwrapped. Otherwise, the reverse happens.

15 Finding Mutated Variables

```
(require unstable/mutated-vars)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(find-mutated-vars stx) → void?
  stx : syntax?
```

Traverses `stx`, which should be `module-level-form` in the sense of the grammar for fully-expanded forms, and records all of the variables that are mutated.

```
(is-var-mutated? id) → boolean?
  id : identifier?
```

Produces `#t` if `id` is mutated by an expression previously passed to `find-mutated-vars`, otherwise produces `#f`.

Examples:

```
> (find-mutated-vars #'(begin (set! var 'foo) 'bar))
> (is-var-mutated? #'var)
#t
> (is-var-mutated? #'other-var)
#f
```

16 Find

```
(require unstable/find)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(find  pred
       x
      [#:stop-on-found? stop-on-found?
       #:stop stop
       #:get-children get-children])    → list?
  pred : (-> any/c any/c)
  x : any/c
  stop-on-found? : any/c = #f
  stop : (or/c #f (-> any/c any/c)) = #f
  get-children : (or/c #f (-> any/c (or/c #f list?))) = #f
```

Returns a list of all values satisfying *pred* contained in *x* (possibly including *x* itself).

If *stop-on-found?* is true, the children of values satisfying *pred* are not examined. If *stop* is a procedure, then the children of values for which *stop* returns true are not examined (but the values themselves are; *stop* is applied after *pred*). Only the current branch of the search is stopped, not the whole search.

The search recurs through pairs, vectors, boxes, and the accessible fields of structures. If *get-children* is a procedure, it can override the default notion of a value's children by returning a list (if it returns false, the default notion of children is used).

No cycle detection is done, so `find` on a cyclic graph may diverge. To do cycle checking yourself, use *stop* and a mutable table.

Examples:

```
> (find symbol? '((all work) and (no play)))
'(all work and no play)
> (find list? '#((all work) and (no play)) #:stop-on-found? #t)
'((all work) (no play))
> (find negative? 100
        #:stop-on-found? #t
        #:get-children (lambda (n) (list (- n 12))))
'(-8)
> (find symbol? (shared ([x (cons 'a x)]) x)
        #:stop (let ([table (make-hasheq)])
                 (lambda (x)
                   (begin0 (hash-ref table x #f)
```

```
                              (hash-set! table x #t)))))
  '(a)
```

```
(find-first  pred
             x
            [#:stop stop
             #:get-children get-children
             #:default default])          → any/c
  pred : (-> any/c any/c)
  x : any/c
  stop : (or/c #f (-> any/c any/c)) = #f
  get-children : (or/c #f (-> any/c (or/c #f list?))) = #f
  default : any/c = (lambda () (error ....))
```

Like `find`, but only returns the first match. If no matches are found, *default* is applied as a thunk if it is a procedure or returned otherwise.

Examples:

```
  > (find-first symbol? '((all work) and (no play)))
  'all
  > (find-first list? '#((all work) and (no play)))
  '(all work)
  > (find-first negative? 100
                #:get-children (lambda (n) (list (- n 12))))
  -8
  > (find-first symbol? (shared ([x (cons 'a x)]) x))
  'a
```

17 Interface-Oriented Programming for Classes

```
(require unstable/class-iop)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(define-interface name-id (super-ifc-id ...) (method-id ...))
```

Defines *name-id* as a static interface extending the interfaces named by the *super-ifc-id*s and containing the methods specified by the *method-id*s.

A static interface name is used by the checked method call variants (`send/i`, `send*/i`, and `send/apply/i`). When used as an expression, a static interface name evaluates to an interface value.

Examples:

```
> (define-interface stack<%> () (empty? push pop))
> stack<%>
#<|interface:stack<%>|>
> (define stack%
    (class* object% (stack<%>)
      (define items null)
      (define/public (empty?) (null? items))
      (define/public (push x) (set! items (cons x items)))
      (define/public (pop) (begin (car items) (set! items (cdr items))))
      (super-new)))
```

```
(define-interface/dynamic name-id ifc-expr (method-id ...))
```

Defines *name-id* as a static interface with dynamic counterpart *ifc-expr*, which must evaluate to an interface value. The static interface contains the methods named by the *method-id*s. A run-time error is raised if any *method-id* is not a member of the dynamic interface *ifc-expr*.

Use `define-interface/dynamic` to wrap interfaces from other sources.

Examples:

```
> (define-interface/dynamic object<%> (class-
>interface object%) ())
> object<%>
#<interface:object%>
```

```
(send/i obj-exp static-ifc-id method-id arg-expr ...)
```

Checked variant of `send`.

The argument *`static-ifc-id`* must be defined as a static interface. The method *`method-id`* must be a member of the static interface *`static-ifc-id`*; otherwise a compile-time error is raised.

The value of `obj-expr` must be an instance of the interface *`static-ifc-id`*; otherwise, a run-time error is raised.

Examples:

```
> (define s (new stack%))
> (send/i s stack<%> push 1)
> (send/i s stack<%> popp)
eval:9:0: send/i: method not in static interface in: popp
> (send/i (new object%) stack<%> push 2)
send/i: interface check failed on: (object)
```

`(send*/i` *`obj-expr static-ifc-id (method-id arg-expr ...) ...`*`)`

Checked variant of `send*`.

Example:

```
> (send*/i s stack<%>
    (push 2)
    (pop))
```

`(send/apply/i` *`obj-expr static-ifc-id method-id arg-expr ... list-arg-expr`*`)`

Checked variant of `send/apply`.

Example:

```
> (send/apply/i s stack<%> push (list 5))
```

`(define/i` *`id static-ifc-id expr`*`)`

Checks that *`expr`* evaluates to an instance of *`static-ifc-id`* before binding it to *`id`*. If *`id`* is subsequently changed (with `set!`), the check is performed again.

No dynamic object check is performed when calling a method (using `send/i`, etc) on a name defined via `define/i`.

`(init/i (`*`id static-ifc-id maybe-default-expr`*`) ...)`
`(init-field/i (`*`id static-ifc-id maybe-default-expr`*`) ...)`

```
(init-private/i (id static-ifc-id maybe-default-expr) ...)

maybe-default-expr = ()
                   | default-expr
```

Checked versions of `init` and `init-field`. The value attached to each *id* is checked against the given interface.

No dynamic object check is performed when calling a method (using `send/i`, etc) on a name bound via one of these forms. Note that in the case of `init-field/i` this check omission is unsound in the presence of mutation from outside the class. This should be fixed.

18 Sequences

```
(require unstable/sequence)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(in-syntax stx) → sequence?
  stx : syntax?
```

Produces a sequence equivalent to `(syntax->list lst)`.

An `in-syntax` application can provide better performance for syntax iteration when it appears directly in a `for` clause.

Example:

```
> (for/list ([x (in-syntax #'(1 2 3))])
    x)
'(#<syntax:2:0 1> #<syntax:2:0 2> #<syntax:2:0 3>)
```

```
(in-pairs seq) → sequence?
  seq : sequence?
```

Produces a sequence equivalent to `(in-parallel (sequence-lift car seq) (sequence-lift cdr seq))`.

```
(in-sequence-forever seq val) → sequence?
  seq : sequence?
  val : any/c
```

Produces a sequence whose values are the elements of *seq*, followed by *val* repeated.

```
(sequence-lift f seq) → sequence?
  f : procedure?
  seq : sequence?
```

Produces the sequence of *f* applied to each element of *seq*.

Example:

```
> (for/list ([x (sequence-lift add1 (in-range 10))])
    x)
'(1 2 3 4 5 6 7 8 9 10)
```

19 Hash Tables

```
(require unstable/hash)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(hash-union t1 t2 combine) → hash?
  t1 : hash?
  t2 : hash?
  combine : (any/c any/c any/c . -> . any/c)
```

Produces the combination of *t1* and *t2*. If either *t1* or *t2* has a value for key `k`, then the result has the same value for `k`. If both *t1* and *t2* have a value for `k`, the result has the value `(`*combine* `k (hash-ref` *t1* `k) (hash-ref` *t2* `k))` for `k`.

Examples:

```
  > (hash-union #hash((a . 5) (b . 0)) #hash((d . 12) (c .
1)) (lambda (k v1 v2) v1))
  '#hash((c . 1) (b . 0) (d . 12) (a . 5))
  > (hash-union #hash((a . 5) (b . 0)) #hash((a . 12) (c .
1)) (lambda (k v1 v2) v1))
  '#hash((c . 1) (b . 0) (a . 5))
```

20 Match

```
(require unstable/match)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(== val comparator)
(== val)
```

A match expander which checks if the matched value is the same as `val` when compared by `comparator`. If `comparator` is not provided, it defaults to `equal?`.

Examples:

```
> (match (list 1 2 3)
    [(== (list 1 2 3)) 'yes]
    [_ 'no])
'yes
> (match (list 1 2 3)
    [(== (list 1 2 3) eq?) 'yes]
    [_ 'no])
'no
> (match (list 1 2 3)
    [(list 1 2 (== 3 =)) 'yes]
    [_ 'no])
'yes
```

21 Skip Lists

```
(require unstable/skip-list)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

Skip lists are a simple, efficient data structure for mutable dictionaries with totally ordered keys. They were described in the paper "Skip Lists: A Probabilistic Alternative to Balanced Trees" by William Pugh in Communications of the ACM, June 1990, 33(6) pp668-676.

A skip-list is a dictionary (`dict?` from `racket/dict`). It also supports extensions of the dictionary interface for iterator-based search and mutation.

```
(make-skip-list =? <?) → skip-list?
  =? : (any/c any/c . -> . any/c)
  <? : (any/c any/c . -> . any/c)
```

Makes a new empty skip-list. The skip-list uses `=?` and `<?` to order keys.

Examples:

```
> (define skip-list (make-skip-list = <))
> (skip-list-set! skip-list 3 'apple)
> (skip-list-set! skip-list 6 'cherry)
> (dict-map skip-list list)
'((3 apple) (6 cherry))
> (skip-list-ref skip-list 3)
'apple
> (skip-list-remove! skip-list 6)
> (skip-list-count skip-list)
1
```

```
(skip-list? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a skip-list, `#f` otherwise.

```
(skip-list-ref skip-list key [default]) → any/c
  skip-list : skip-list?
  key : any/c
  default : any/c = (lambda () (error ....))
```

```
(skip-list-set! skip-list key value) → void?
  skip-list : skip-list?
  key : any/c
  value : any/c
(skip-list-remove! skip-list key) → void?
  skip-list : skip-list?
  key : any/c
(skip-list-count skip-list) → exact-nonnegative-integer?
  skip-list : skip-list?
(skip-list-iterate-first skip-list) → (or/c skip-list-iter? #f)
  skip-list : skip-list?
(skip-list-iterate-next skip-list iter)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  iter : skip-list-iter?
(skip-list-iterate-key skip-list iter) → any/c
  skip-list : skip-list?
  iter : skip-list-iter?
(skip-list-iterate-value skip-list iter) → any/c
  skip-list : skip-list?
  iter : skip-list-iter?
```

Implementations of `dict-ref`, `dict-set!`, `dict-remove!`, `dict-count`, `dict-iterate-first`, `dict-iterate-next`, `dict-iterate-key`, and `dict-iterate-value`, respectively.

```
(skip-list-iterate-greatest/<? skip-list
                               key)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  key : any/c
(skip-list-iterate-greatest/<=? skip-list
                                key)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  key : any/c
(skip-list-iterate-least/>? skip-list key)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  key : any/c
(skip-list-iterate-least/>=? skip-list key)
 → (or/c skip-list-iter? #f)
  skip-list : skip-list?
  key : any/c
```

Return the position of, respectively, the greatest key less than *key*, the greatest key less than or equal to *key*, the least key greater than *key*, and the least key greater than or equal to *key*.

```
(skip-list-iterate-set-key! skip-list
                            iter
                            key)      → void?
  skip-list : skip-list?
  iter : skip-list-iter?
  key : any/c
(skip-list-iterate-set-value! skip-list
                              iter
                              value)    → void?
  skip-list : skip-list?
  iter : skip-list-iter?
  value : any/c
```

Set the key and value, respectively, at the position *iter* in *skip-list*.

Warning: Changing a position's key to be less than its predecessor's key or greater than its successor's key results in an out-of-order skip-list, which may cause comparison-based operations to behave incorrectly.

```
(skip-list-iter? v) → boolean?
  v : any/c
```

Returns `#t` if *v* represents a position in a skip-list, `#f` otherwise.

22 Interval Maps

```
(require unstable/interval-map)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

An interval-map is a mutable dictionary-like data structure where mappings are added by *half-open* intervals and queried by discrete points. Interval-maps can be used with any total order. Internally, an interval-map uses a skip-list (`unstable/skip-list`) of intervals for efficient query and update.

Interval-maps implement the dictionary (`racket/dict`) interface to a limited extent. Only `dict-ref` and the iteraction-based methods (`dict-iterate-first`, `dict-map`, etc) are supported. For the iteration-based methods, the mapping's keys are considered the pairs of the start and end positions of the mapping's intervals.

Examples:

```
> (define r (make-numeric-interval-map))
> (interval-map-set! r 1 5 'apple)
> (interval-map-set! r 6 10 'pear)
> (interval-map-set! r 3 6 'banana)
> (dict-map r list)
'(((1 . 3) apple) ((3 . 6) banana) ((6 . 10) pear))
```

```
(make-interval-map =? <? [translate]) → interval-map?
  =? : (any/c any/c . -> . any/c)
  <? : (any/c any/c . -> . any/c)
  translate : (or/c (any/c any/c . -> . (any/c . -> . any/c)) #f)
            = #f
```

Makes a new empty interval-map. The interval-map uses `=?` and `<?` to order the endpoints of intervals.

If *translate* is a procedure, the interval-map supports contraction and expansion of regions of its domain via `interval-map-contract!` and `interval-map-expand!`. See also `make-numeric-interval-map`.

```
(make-numeric-interval-map) → interval-map-with-translate?
```

Makes a new empty interval-map suitable for representing numeric ranges.

Equivalent to

```
(make-interval-map = < (lambda (x y) (lambda (z) (+ z (- y x)))))
```

```
(interval-map? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is an interval-map, `#f` otherwise.

```
(interval-map-with-translate? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is an interval-map constructed with support for translation of keys, `#f` otherwise.

```
(interval-map-ref  interval-map
                   position
                  [default])     → any/c
  interval-map : interval-map?
  position : any/c
  default : any/c = (lambda () (error ....))
```

Return the value associated with *position* in *interval-map*. If no mapping is found, *default* is applied if it is a procedure, or returned otherwise.

```
(interval-map-set! interval-map
                   start
                   end
                   value)        → void?
  interval-map : interval-map?
  start : any/c
  end : any/c
  value : any/c
```

Updates *interval-map*, associating every position in [*start*, *end*) with *value*.

Existing interval mappings contained in [*start*, *end*) are destroyed, and partly overlapping intervals are truncated. See `interval-map-update*!` for an updating procedure that preserves distinctions within [*start*, *end*).

```
(interval-map-update*!  interval-map
                        start
                        end
                        updater
                       [default])     → void?
  interval-map : interval-map?
```

```
  start : any/c
  end : any/c
  updater : (any/c . -> . any/c)
  default : any/c = (lambda () (error ....))
```

Updates *interval-map*, associating every position in [*start*, *end*) with the result of applying *updater* to the position's previously associated value, or to the default value produced by *default* if no mapping exists.

Unlike interval-map-set!, interval-map-update*! preserves existing distinctions within [*start*, *end*).

```
(interval-map-remove! interval-map
                      start
                      end)          → void?
  interval-map : interval-map?
  start : any/c
  end : any/c
```

Removes the value associated with every position in [*start*, *end*).

```
(interval-map-expand! interval-map
                      start
                      end)          → void?
  interval-map : interval-map-with-translate?
  start : any/c
  end : any/c
```

Expands *interval-map*'s domain by introducing a gap [*start*, *end*) and adjusting intervals after *start* using (*translate start end*).

If *interval-map* was not constructed with a *translate* argument, an exception is raised. If *start* is not less than *end*, an exception is raised.

```
(interval-map-contract! interval-map
                        start
                        end)          → void?
  interval-map : interval-map-with-translate?
  start : any/c
  end : any/c
```

Contracts *interval-map*'s domain by removing all mappings on the interval [*start*, *end*) and adjusting intervals after *end* using (*translate end start*).

If *interval-map* was not constructed with a *translate* argument, an exception is raised. If *start* is not less than *end*, an exception is raised.

```
(interval-map-cons*! interval-map
                     start
                     end
                     v
                    [default])     → void?
  interval-map : interval-map?
  start : any/c
  end : any/c
  v : any/c
  default : any/c = null
```

Same as the following:

```
(interval-map-update*! interval-map start end
                       (lambda (old) (cons v old))
                       default)
```

```
(interval-map-iter? v) → boolean?
  v : any/c
```

Returns `#t` if *v* represents a position in an interval-map, `#f` otherwise.

23 Generics

```
(require unstable/generics)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(define-generics (name prop:name name?)
  [method . kw-formals*]
  ...)

kw-formals* = (arg* ...)
            | (arg* ...+ . rest-id)
            | rest-id

      arg* = id
           | [id]
           | keyword id
           | keyword [id]
```

name : identifier?

prop:name : identifier?

name? : identifier?

method : identifier?

Defines *name* as a transformer binding for the static information about a new generic group.

Defines *prop:name* as a structure type property. Structure types implementing this generic group should have this property where the value is a vector with one element per *method* where each value is either `#f` or a procedure with the same arity as specified by *kw-formals**. (*kw-formals** is similar to the `kw-formals` used by `lambda`, except no expression is given for optional arguments.) The arity of each method is checked by the guard on the structure type property.

Defines *name?* as a predicate identifying instances of structure types that implement this generic group.

Defines each *method* as a generic procedure that calls the corresponding method on values where *name?* is true. Each method must have a required by-position argument that is `free-identifier=?` to *name*. This argument is used in the generic definition to locate the specialization.

```
(generics name
          [method . kw-formals*]
          ...)
```

name : identifier?

method : identifier?

Expands to

```
(define-generics (name prop:name name?)
  [method . kw-formals*]
  ...)
```

where *prop:name* and *name?* are created with the lexical context of *name*.

```
(define-methods name definition ...)
```

name : identifier?

name must be a transformer binding for the static information about a new generic group.

Expands to a value usable as the property value for the structure type property of the *name* generic group.

If the *definition*s define the methods of *name*, then they are used in the property value.

If any method of *name* is not defined, then `#f` is used to signify that the structure type does not implement the particular method.

Allows `define/generic` to appear in *definition* ....

```
(define/generic local-name method-name)
```

local-name : identifier?

method-name : identifier?

When used inside `define-methods`, binds *local-name* to the generic for *method-name*. This is useful for method specializations to use the generic methods on other values.

Syntactically an error when used outside `define-methods`.

Examples:

```
> (define-generics (printable prop:printable printable?)
    (gen-print printable [port]))
```

```
    (gen-port-print port printable)
    (gen-print* printable [port] #:width width #:height [height]))
> (define-struct num (v)
    #:property prop:printable
    (define-methods printable
      (define/generic super-print gen-print)
      (define (gen-print n [port (current-output-port)])
        (fprintf port "Num: ~a" (num-v n)))
      (define (gen-port-print port n)
        (super-print n port))
      (define (gen-print* n [port (current-output-port)]
                          #:width w #:height [h 0])
        (fprintf port "Num (~ax~a): ~a" w h (num-v n)))))
> (define-struct bool (v)
    #:property prop:printable
    (define-methods printable
      (define/generic super-print gen-print)
      (define (gen-print b [port (current-output-port)])
        (fprintf port "Bool: ~a"
                 (if (bool-v b) "Yes" "No")))
      (define (gen-port-print port b)
        (super-print b port))
      (define (gen-print* b [port (current-output-port)]
                          #:width w #:height [h 0])
        (fprintf port "Bool (~ax~a): ~a" w h
                 (if (bool-v b) "Yes" "No")))))
> (define x (make-num 10))
> (gen-print x)
Num: 10
> (gen-port-print (current-output-port) x)
Num: 10
> (gen-print* x #:width 100 #:height 90)
Num (100x90): 10
> (define y (make-bool #t))
> (gen-print y)
Bool: Yes
> (gen-port-print (current-output-port) y)
Bool: Yes
> (gen-print* y #:width 100 #:height 90)
Bool (100x90): Yes
```

24 Mark Parameters

```
(require unstable/markparam)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

This library provides a simplified version of parameters that are backed by continuation marks, rather than parameterizations. This means they are slightly slower, are not inherited by child threads, do not have initial values, and cannot be imperatively mutated.

```
(struct mark-parameter ())
```

The struct for mark parameters. It is guaranteed to be serializable and transparent. If used as a procedure, it calls `mark-parameter-first` on itself.

```
(mark-parameter-first mp [tag]) → any/c
  mp : mark-parameter?
  tag : continuation-prompt-tag?
      = default-continuation-prompt-tag
```

Returns the first value of *mp* up to *tag*.

```
(mark-parameter-all mp [tag]) → list?
  mp : mark-parameter?
  tag : continuation-prompt-tag?
      = default-continuation-prompt-tag
```

Returns the values of *mp* up to *tag*.

```
(mark-parameters-all mps none-v [tag]) → (listof vector?)
  mps : (listof mark-parameter?)
  none-v : [any/c #f]
  tag : continuation-prompt-tag?
      = default-continuation-prompt-tag
```

Returns the values of the *mps* up to *tag*. The length of each vector in the result list is the same as the length of *mps*, and a value in a particular vector position is the value for the corresponding mark parameter in *mps*. Values for multiple mark parameter appear in a single vector only when the mark parameters are for the same continuation frame in the current continuation. The *none-v* argument is used for vector elements to indicate the lack of a value.

`(mark-parameterize ([mp expr] ...) body-expr ...)`

Parameterizes `(begin body-expr ...)` by associating each *mp* with the evaluation of *expr* in the parameterization of the entire expression.

25 Debugging

```
(require unstable/debug)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(debug (f args ...))
(debug f args ...)
```

Produce debugging output for the application of `f`, including the values of `args`.

Examples:

```
> (debug (+ 3 4 (* 5 6)))
starting + (#<procedure:+>)
arguments are:
 3: 3
 4: 4
 (* 5 6): 30
+ result was 37
37
> (debug + 1 2 3)
starting + (#<procedure:+>)
arguments are:
 1: 1
 2: 2
 3: 3
+ result was 6
6
```

```
(debugm f args ...)
```

Produce debugging output for the application of `f`, but does not parse or print args. Suitable for use debugging macros.

Examples:

```
> (debugm match (list 1 2 3)
    [(list x y z) (+ x y z)])
starting match
match result was 6
6
> (debugm + 1 2 3)
starting +
+ result was 6
6
```

26 Byte Counting Ports

```
(require unstable/byte-counting-port)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

This library provides an output port constructor like `open-output-nowhere`, except it counts how many bytes have been written (available through `file-position`.)

```
(make-byte-counting-port [name]) → output-port?
  name : any/c = 'byte-counting-port
```

Creates and returns an output port that discards all output sent to it (without blocking.) The *name* argument is used as the port's name. The total number bytes written is available through `file-position`.

27 GUI libraries

27.1 Notify-boxes

```
(require unstable/gui/notify)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
notify-box% : class?
  superclass: object%
```

A notify-box contains a mutable cell. The notify-box notifies its listeners when the contents of the cell is changed.

Examples:

```
> (define nb (new notify-box% (value 'apple)))
> (send nb get)
'apple
> (send nb set 'orange)
> (send nb listen (lambda (v) (printf "New value: ~s\n" v)))
> (send nb set 'potato)
New value: potato
```

```
(new notify-box% [value value]) → (is-a?/c notify-box%)
  value : any/c
```

Creates a notify-box initially containing *value*.

```
(send a-notify-box get) → any/c
```

Gets the value currently stored in the notify-box.

```
(send a-notify-box set v) → void?
  v : any/c
```

Updates the value stored in the notify-box and notifies the listeners.

```
(send a-notify-box listen listener) → void?
  listener : (-> any/c any)
```

Adds a callback to be invoked on the new value when the notify-box's contents change.

```
(send a-notify-box remove-listener listener) → void?
```

listener : (-> any/c any)

Removes a previously-added callback.

(send *a-notify-box* remove-all-listeners) → void?

Removes all previously registered callbacks.

```
(notify-box/pref  proc
                  [#:readonly? readonly?]) → (is-a?/c notify-box%)
  proc : (case-> (-> any/c) (-> any/c void?))
  readonly? : boolean? = #f
```

Creates a notify-box with an initial value of (*proc*). Unless *readonly?* is true, *proc* is invoked on the new value when the notify-box is updated.

Useful for tying a notify-box to a preference or parameter. Of course, changes made directly to the underlying parameter or state are not reflected in the notify-box.

Examples:

```
> (define animal (make-parameter 'ant))
> (define nb (notify-box/pref animal))
> (send nb listen (lambda (v) (printf "New value: ~s\n" v)))
> (send nb set 'bee)
New value: bee
> (animal 'cow)
> (send nb get)
'bee
> (send nb set 'deer)
New value: deer
> (animal)
'deer
```

```
(define-notify name value-expr)

  value-expr : (is-a?/c notify-box%)
```

Class-body form. Declares *name* as a field and get-*name*, set-*name*, and listen-*name* as methods that delegate to the get, set, and listen methods of value.

The *value-expr* argument must evaluate to a notify-box, not just the initial contents for a notify box.

Useful for aggregating many notify-boxes together into one “configuration” object.

Examples:

```
> (define config%
    (class object%
      (define-notify food (new notify-box% (value 'apple)))
      (define-notify animal (new notify-box% (value 'ant)))
      (super-new)))
> (define c (new config%))
> (send c listen-food
          (lambda (v) (when (eq? v 'honey) (send c set-
animal 'bear))))
> (let ([food (get-field food c)])
    (send food set 'honey))
> (send c get-animal)
'bear
```

```
(menu-option/notify-box parent
                        label
                        notify-box)
 → (is-a?/c checkable-menu-item%)
  parent : (or/c (is-a?/c menu%) (is-a?/c popup-menu%))
  label : label-string?
  notify-box : (is-a?/c notify-box%)
```

Creates a `checkable-menu-item%` tied to *notify-box*. The menu item is checked whenever `(send` *notify-box* `get)` is true. Clicking the menu item toggles the value of *notify-box* and invokes its listeners.

```
(check-box/notify-box parent
                      label
                      notify-box) → (is-a?/c check-box%)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
  label : label-string?
  notify-box : (is-a?/c notify-box%)
```

Creates a `check-box%` tied to *notify-box*. The check-box is checked whenever `(send` *notify-box* `get)` is true. Clicking the check box toggles the value of *notify-box* and invokes its listeners.

```
(choice/notify-box parent
                   label
                   choices
                   notify-box) → (is-a?/c choice%)
  parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
                 (is-a?/c panel%) (is-a?/c pane%))
```

```
  label : label-string?
  choices : (listof label-string?)
  notify-box : (is-a?/c notify-box%)
```

Creates a `choice%` tied to *notify-box*. The choice control has the value `(send` *notify-box* `get)` selected, and selecting a different choice updates *notify-box* and invokes its listeners.

If the value of *notify-box* is not in *choices*, either initially or upon an update, an error is raised.

```
(menu-group/notify-box parent
                       labels
                       notify-box)
 → (listof (is-a?/c checkable-menu-item%))
  parent : (or/c (is-a?/c menu%) (is-a?/c popup-menu%))
  labels : (listof label-string?)
  notify-box : (is-a?/c notify-box%)
```

Returns a list of `checkable-menu-item%` controls tied to *notify-box*. A menu item is checked when its label is `(send` *notify-box* `get)`. Clicking a menu item updates *notify-box* to its label and invokes *notify-box*'s listeners.

27.2 Preferences

```
(require unstable/gui/prefs)
```

This library is *unstable*; compatibility will not be maintained. See *Unstable* for more information.

```
(pref:get/set pref) → (case-> (-> any/c) (-> any/c void?))
  pref : symbol?
```

Returns a procedure that when applied to zero arguments retrieves the current value of the preference (`framework/preferences`) named *pref* and when applied to one argument updates the preference named *pref*.