

Unstable

Version 4.2.5.1

March 18, 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 drscheme 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 DrScheme 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 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.

5 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.

6 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 $(- (\text{length } l1) (\text{length } l2))$ 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 c> #t) (#<syntax:10:0 b> #t) (#<syntax:10:0 a> #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)

```

7 Net

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

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

7.1 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.

8 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`.

9 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.

9.1 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: #(struct: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)
#(struct: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)
#(struct: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))
#(struct: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))
#(struct: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))
#(struct: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)))
```

```
(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
```

```
(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))
""

```

9.2 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 expr)
```

This form quotes the source location of *expr* as a *srcloc* structure, using the location of the whole *(quote-srcloc)* expression if no *expr* is given.

Examples:

```

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

```

```
> (here)
#(struct:srcloc eval 8 0 8 1)
```

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

These forms quote various fields of the source location of `expr`, or of the whole macro application if no `expr` 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-path)`

This form quotes a module path suitable for use with `require` which refers to the module in which the macro application occurs. If executed at the top level, it may return `'top-level`, or it may return a valid module path if the current namespace was constructed by `module->namespace` (such as at the DrScheme interactions window).

This macro operates by creating a variable reference (see `#!/variable-reference`) at the point of its application. It thus automatically describes its final expanded position, rather than the module of any macro definition that happens to use it.

Examples:

```
> (quote-module-path)
top-level
> (module A scheme
  (require unstable/location)
  (define-syntax-rule (here) (quote-module-path))
  (define a (here))
  (provide a here))
> (require 'A)
> a
'A
> (module B scheme
  (require unstable/location)
  (require 'A)
  (define b (here))
  (provide b))
> (require 'B)
> b
'B
> [current-namespace (module->namespace ''A)]
> (quote-module-path)
'A
```

`(quote-module-name)`

This form quotes the name (path or symbol) of the module in which the macro application occurs, or `#f` if it occurs at the top level. As with `quote-module-path`, `quote-module-name` uses a variable reference, so a top level namespace created by `module->namespace` will be treated as a module, and the macro will always produce the module name of its final expanded position.

Examples:

```
> (quote-module-name)
#f
```

```
> (module A scheme
  (require unstable/location)
  (define-syntax-rule (here) (quote-module-name))
  (define a (here))
  (provide a here))
> (require 'A)
> a
A
> (module B scheme
  (require unstable/location)
  (require 'A)
  (define b (here))
  (provide b))
> (require 'B)
> b
B
> [current-namespace (module->namespace ''A)]
> (quote-module-name)
A
```

10 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.

11 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 *exprs* 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) 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 #(struct: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)
()
```

12 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 DrScheme.) 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 DrScheme 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 produces ~s\n"
                expr))]))
> (show-me (+ 2 5))
at compile time produces 7
at run time produces 7
> (define-for-syntax fruit 'apple)
> (define fruit 'pear)
> (show-me fruit)
at compile time produces apple
at run time produces 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-exprs* as well as the *bodys*, and the *patterns* 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 stx1 ...) → (listof A)  
  f : (-> syntax? A)  
  stx1 : syntax?
```

Performs `(map f (syntax->list stx1) ...)`.

Example:

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

13 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-

ropriate 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.

14 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
```

15 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
```

16 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-ids* and containing the methods specified by the *method-ids*.

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-ids*. 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: #(struct: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.

17 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 (lift car seq) (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`.

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

19 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
```

20 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 `scheme/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.

21 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 (`scheme/dict`) interface to a limited extent. Only `dict-ref` and the iteration-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.

22 GUI libraries

22.1 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.

22.2 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*.