

# Unstable

Version 4.2.3

December 1, 2009

`(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)

---

```
(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)`

---

`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))`.

## 4 Exceptions

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

---

```
(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 Lists

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

---

```
(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*.

Examples:

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

## 6 Net

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

### 6.1 URLs

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

---

```
(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.

## 7 Path

(require unstable/path)

---

(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*.

## 8 Strings

(require unstable/string)

---

(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.

## 9 Structs

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

---

```
(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)
()
```

## 10 Syntax

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

---

```
(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-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:6:0 (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:7:0 (lambda (x2) x2)>
```

---

```
(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.)

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:14:0 ((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)
```

## 11 Anaphoric Contracts

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

---

```
(poly/c ([id+ id-] ...) cnt)
```

Creates an “anaphoric” contract, using the `id+ ...` as the positive positions, and the `id- ...` as the negative positions.

Anaphoric contracts verify that only values provided to a given positive position flow out of the corresponding negative position.

Examples:

```
> (define/contract (f x) (poly/c ([in out]) (in . -> . out))
  (if (equal? x 17) 18 x))
> (f 1)
1
> (f #f)
#f
> (f 17)
(function f) broke the contract
  (poly/c ((in out)) (->
in out))
  on f; expected <out>, given: 18
```

---

```
(apply/c cnt [#:name name]) → contract?
cnt : any/c
name : any/c = (build-compound-type-name 'apply/c c)
```

Produces a procedure contract that is like `cnt`, but any delayed evaluation in `cnt` is re-done on every application of the contracted function.

---

```
(memory/c [#:name name
#:from from
#:to to
#:weak weak?
#:equal equal
#:table make-table]) → flat-contract? flat-contract?
name : any/c = "memory/c"
from : any/c = (format "~a:from" name)
to : any/c = (format "~a:to" name)
weak? : any/c = #t
equal : (or/c 'eq 'eqv 'equal) = 'eq
```

```
make-table : (-> hash?)
= (case equal
  [(eq) (if weak? make-weak-hasheq make-hasheq)]
  [(eqv) (if weak? make-weak-hasheqv make-hasheqv)]
  [(equal) (if weak? make-weak-hash make-hash)])
```

Produces a pair of contracts. The first contract remembers all values that flow into it, and rejects nothing. The second accepts only values that have previously been passed to the first contract.

If *weak?* is not *#f*, the first contract holds onto the values only weakly. *from* and *to* are the names of the of the two contracts.

## 12 Finding Mutated Variables

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

---

```
(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
```

## 13 Find

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

---

```
(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)) x)
a

```

## 14 Interface-Oriented Programming for Classes

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

---

```
(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.

## 15 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`.

## 16 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))
```

## 17 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
```

## 18 GUI libraries

### 18.1 Notify-boxes

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

---

```
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.

---

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

Creates a notify-box with the initial value *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) → (is-a?/c notify-box%)  
  proc : (case-> (-> any/c) (-> any/c void?))
```

Creates a notify-box with an initial value of (*proc*) that invokes *proc* on the new value when the notify-box is updated.

Useful for making a notify-box tied to a preference or parameter.

---

```
(notify-box/pref/readonly proc) → (is-a?/c notify-box%)  
  proc : (-> any/c)
```

Creates a notify-box with an initial value of (*proc*).

Useful for making a notify-box that takes its initial value from a preference or parameter but does not update the preference or parameter.

---

```
(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.

---

```
(checkbox/notify-box parent  
  label  
  notify-box) → (is-a?/c checkbox%)  
  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 *checkbox%* 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.