Unstable

Version 5.0.2

November 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 Classes and Objects

```
(require unstable/class)
```

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

This module provides tools for classes, objects, and mixins.

3.1 Predicates and Contracts

`class-or-interface/c : flat-contract?`

Recognizes classes and interfaces.

`(object-provides/c spec ...) → flat-contract?`
`  spec : class-or-interface/c`

Recognizes objects which are instances of all the given classes and interfaces.

`(class-provides/c spec ...) → flat-contract?`
`  spec : class-or-interface/c`

Recognizes classes which are subclasses (not strictly) and implementations, respectively, of all the given classes and interfaces.

`(mixin-provides/c [super-expr ...] [sub-expr ...])`

Function contract for a mixin whose argument is the parent class *c%* matching `(class-provides/c super-expr ...)` and whose result matches `(class-provides/c c% sub-expr ...)`.

3.2 Mixins

`(ensure-interface i<%> mx c%) → (class-provides/c c% i<%>)`
`  i<%> : interface?`
`  mx : (mixin-provides/c [] [i<%>])`
`  c% : class?`

Returns *c%* if it implements *i<%>*; otherwise, returns *(mx c%)*.

3.3 Methods

(send+ obj [message arg ...] ...)

Sends each message (with arguments) to *obj*, then returns *obj*.

Examples:

```
(define c%
  (class object%
    (super-new)
    (define/public (say msg) (printf "~a!\n" msg))))
> (send+ (new c%) [say 'Hello] [say 'Good-bye])
Hello!
Good-bye!
(object:c% ...)
```

(send-each objs message arg ...)

Sends the message to each object in the list *objs*, returning *(void)*.

Examples:

```
(define c%
  (class object%
    (super-new)
    (init-field msg)
    (define/public (say to) (printf "~a, ~a!\n" msg to))))
> (send-each
   (list (new c% [msg 'Hello])
         (new c% [msg 'Good-bye]))
   say 'World)
Hello, World!
Good-bye, World!
```

4 Contracts

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

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

```
(non-empty-string? x) → boolean?
  x : any/c
(non-empty-list? x) → boolean?
  x : any/c
(non-empty-bytes? x) → boolean?
  x : any/c
(non-empty-vector? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is of the appropriate data type (string, list, bytes, or vector, respectively) and is not empty; returns `#f` otherwise.

```
(singleton-list? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a list of one element; returns `#f` otherwise.

```
port-number? : contract?
```

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

```
tcp-listen-port? : contract?
```

Equivalent to `(between/c 0 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.

`failure-result/c` : `contract?`

A contract that describes the failure result arguments of procedures such as `hash-ref`.

Equivalent to `(if/c procedure? (-> any) any/c)`.

`(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.

The subsequent bindings were added by Carl Eastlund <cce@racket-lang.org>.

4.1 Flat Contracts

`nat/c` : `flat-contract?`

This contract recognizes natural numbers that satisfy `exact-nonnegative-integer?`.

`pos/c` : `flat-contract?`

This contract recognizes positive integers that satisfy `exact-positive-integer?`.

`truth/c` : `flat-contract?`

This contract recognizes Scheme truth values, i.e., any value, but with a more informative

name and description. Use it in negative positions for arguments that accept arbitrary truth values that may not be booleans.

4.2 Syntax Object Contracts

`(syntax-datum/c datum/c)` → `flat-contract?`
  datum/c : `any/c`

Recognizes syntax objects `stx` such that `(syntax->datum stx)` satisfies *datum/c*.

`(syntax-listof/c elem/c)` → `flat-contract?`
  elem/c : `any/c`

Recognizes syntax objects `stx` such that `(syntax->list stx)` satisfies `(listof elem/c)`.

`(syntax-list/c elem/c ...)` → `flat-contract?`
  elem/c : `any/c`

Recognizes syntax objects `stx` such that `(syntax->list stx)` satisfies `(list/c elem/c ...)`.

4.3 Higher-Order Contracts

`thunk/c` : `contract?`
`unary/c` : `contract?`
`binary/c` : `contract?`

These contracts recognize functions that accept 0, 1, or 2 arguments, respectively, and produce a single result.

`predicate/c` : `contract?`
`predicate-like/c` : `contract?`

These contracts recognize predicates: functions of a single argument that produce a boolean result.

The first constrains its output to satisfy `boolean?`. Use `predicate/c` in positive position for predicates that guarantee a result of `#t` or `#f`.

The second constrains its output to satisfy `truth/c`. Use `predicate-like/c` in negative position for predicates passed as arguments that may return arbitrary values as truth values.

```
comparison/c : contract?
comparison-like/c : contract?
```

These contracts recognize comparisons: functions of two arguments that produce a boolean result.

The first constrains its output to satisfy `boolean?`. Use `comparison/c` in positive position for comparisons that guarantee a result of `#t` or `#f`.

The second constrains its output to satisfy `truth/c`. Use `comparison-like/c` in negative position for comparisons passed as arguments that may return arbitrary values as truth values.

```
(sequence/c elem/c ...) → contract?
  elem/c : contract?
```

Wraps a sequence, obligating it to produce as many values as there are *elem/c* contracts, and obligating each value to satisfy the corresponding *elem/c*. The result is not guaranteed to be the same kind of sequence as the original value; for instance, a wrapped list is not guaranteed to satisfy `list?`.

Examples:

```
> (define/contract predicates
    (sequence/c (-> any/c boolean?))
    (list integer? string->symbol))
> (for ([P predicates])
    (printf "~s\n" (P "cat")))
#f
eval:3.0: (definition predicates) broke the contract
  (sequence/c (-> any/c boolean?))
 on predicates; expected <boolean?>, given: 'cat
```

```
(dict/c key/c value/c) → contract?
  key/c : contract?
  value/c : contract?
```

Wraps a dictionary, obligating its keys to satisfy *key/c* and their corresponding values to satisfy *value/c*. The result is not guaranteed to be the same kind of dictionary as the original value; for instance, a wrapped hash table is not guaranteed to satisfy `hash?`.

Examples:

```
> (define/contract table
    (dict/c symbol? string?)
```

```
    (make-immutable-hash (list (cons 'A "A") (cons 'B 2) (cons 3 "C"))))
> (dict-ref table 'A)
"A"
> (dict-ref table 'B)
eval:4.0: (definition table) broke the contract (dict/c
symbol? string?) on table; expected <string?>, given: 2
> (dict-ref table 3)
eval:4.0: (definition table) broke the contract (dict/c
symbol? string?) on table; expected <symbol?>, given: 3
```

Warning: Bear in mind that key and value contracts are re-wrapped on every dictionary operation, and dictionaries wrapped in `dict/c` multiple times will perform the checks as many times for each operation. Especially for immutable dictionaries (which may be passed through a constructor that involves `dict/c` on each update), contract-wrapped dictionaries may be much less efficient than the original dictionaries.

5 Definitions

```
(require unstable/define)
```

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

This module provides macros for creating and manipulating definitions.

5.1 Deferred Evaluation in Modules

```
(at-end expr)
```

When used at the top level of a module, evaluates *expr* at the end of the module. This can be useful for calling functions before their definitions.

Examples:

```
> (module Failure scheme
    (f 5)
    (define (f x) x))
> (require 'Failure)
reference to an identifier before its definition: f in
module: 'Failure
> (module Success scheme
    (require unstable/define)
    (at-end (f 5))
    (define (f x) x))
> (require 'Success)
```

5.2 Conditional Binding

```
(define-if-unbound x e)
(define-if-unbound (f . args) body ...)
(define-values-if-unbound [x ...] e)
(define-syntax-if-unbound x e)
(define-syntax-if-unbound (f . args) body ...)
(define-syntaxes-if-unbound [x ...] e)
```

These forms define each `x` (or `f`) if no such binding exists, or do nothing if the name(s) is(are) already bound. The `define-values-if-unbound` and `define-syntaxes-if-unbound` forms raise a syntax error if some of the given names are bound and some are

not.

These are useful for writing programs that are portable across versions of Racket with different bindings, to provide an implementation of a binding for versions that do not have it but use the built-in one in versions that do.

Examples:
```
> (define-if-unbound x 1)
> x
1
(define y 2)
> (define-if-unbound y 3)
> y
3
```

5.3 Renaming Definitions

```
(define-renamings [new old] ...)
```

This form establishes a rename transformer for each *new* identifier, redirecting it to the corresponding *old* identifier.

Examples:
```
> (define-renamings [def define] [lam lambda])
> (def plus (lam (x y) (+ x y)))
> (plus 1 2)
3
```

5.4 Forward Declarations

```
(declare-names x ...)
```

This form provides forward declarations of identifiers to be defined later. It is useful for macros which expand to mutually recursive definitions, including forward references, that may be used at the Racket top level.

5.5 Definition Shorthands

```
(define-with-parameter name parameter)
```

Defines the form *name* as a shorthand for setting the parameter *parameter*. Specifically, (*name* value body ...) is equivalent to (parameterize ([*parameter* value]) body ...).

Examples:

```
> (define-with-parameter with-input current-input-port)
> (with-input (open-input-string "Tom Dick Harry") (read))
'Tom
```

(define-single-definition *define-one-name define-many-name*)

This form defines a marco *define-one-name* as a single identifier definition form with function shorthand like define and define-syntax, based on an existing macro *define-many-name* which works like define-values or define-syntaxes.

Examples:

```
> (define-single-definition define-like define-values)
> (define-like x 0)
> x
0
> (define-like (f a b c) (printf "~s, ~s\n" a b) c)
> (f 1 2 3)
1, 2
3
```

5.6 Effectful Transformation

(in-phase1 *e*)

This form executes *e* during phase 1 (the syntax transformation phase) relative to its context, during pass 1 if it occurs in a head expansion position.

(in-phase1/pass2 *e*)

This form executes *e* during phase 1 (the syntax transformation phase) relative to its context, during pass 2 (after head expansion).

6 Dictionaries

```
(require unstable/dict)
```

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

This module provides tools for manipulating dictionary values.

6.1 Dictionary Constructors

```
(empty-dict [#:mutable? mutable?
             #:weak? weak?
             #:compare compare]) → hash?
  mutable? : boolean? = weak?
  weak? : boolean? = #f
  compare : (or/c 'eq 'eqv 'equal) = equal
```

Constructs an empty hash table based on the behavior specified by *mutable?*, *weak?*, and *compare*.

Examples:

```
> (empty-dict)
'#hash()
> (empty-dict #:mutable? #t)
'#hash()
> (empty-dict #:weak? #t)
'#hash()
> (empty-dict #:compare 'eqv)
'#hasheqv()
```

```
(make-dict  d
           [#:mutable? mutable?
            #:weak? weak?
            #:compare compare]) → hash?
  d : dict?
  mutable? : boolean? = weak?
  weak? : boolean? = #f
  compare : (or/c 'eq 'eqv 'equal) = equal
```

Converts a given dictionary *d* to a hash table based on the behavior specified by *mutable?*, *weak?*, and *compare*.

Examples:

```
> (make-dict '([1 . one] [2 . two]))
'#hash((1 . one) (2 . two))
> (make-dict '([1 . one] [2 . two]) #:mutable? #t)
'#hash((1 . one) (2 . two))
> (make-dict '([1 . one] [2 . two]) #:weak? #t)
'#hash((1 . one) (2 . two))
> (make-dict '([1 . one] [2 . two]) #:compare 'eqv)
'#hasheqv((1 . one) (2 . two))
```

```
(custom-dict  equiv?
              [hash-primary
               hash-secondary
               #:mutable? mutable?
               #:weak? weak?])      → dict?
  equiv? : (-> any/c any/c any/c)
  hash-primary : (-> any/c exact-integer?) = (lambda (x) 0)
  hash-secondary : (-> any/c exact-integer?) = (lambda (x) 0)
  mutable? : boolean? = weak?
  weak? : boolean? = #f
```

Constructs a dictionary based on custom comparison and optional hash functions. Given no hash functions, the dictionary defaults to a degenerate hash function and is thus essentially equivalent to a list-based dictionary.

Examples:

```
(define table (custom-dict = add1 sub1 #:mutable? #t))
> (dict-set! table 1 'one)
> (dict-set! table 2 'two)
> (for/list ([(key val) (in-dict table)])
    (cons key val))
'((2 . two) (1 . one))
```

6.2 Dictionary Lookup

```
(dict-ref/check d k) → any/c
  d : dict?
  k : (lambda (k) (dict-has-key? d k))
```

Looks up key `k` in dictionary `d`. Raises a contract error if `d` has no entry for `k`. Equivalent to `(dict-ref d k)`, except for the specific exception value raised.

Example:

```
> (dict-ref/check '([1 . one] [2 . two] [3 . three]) 2)
'two
```

```
(dict-ref/identity d k) → any/c
  d : dict?
  k : any/c
```

Looks up key `k` in dictionary `d`. Returns `k` if `d` has no entry for `k`. Equivalent to `(dict-ref d k (lambda () k))`.

Examples:

```
> (dict-ref/identity '([1 . one] [2 . two] [3 . three]) 2)
'two
> (dict-ref/identity '([1 . one] [2 . two] [3 . three]) 4)
4
```

```
(dict-ref/default d k v) → any/c
  d : dict?
  k : any/c
  v : any/c
```

Looks up key `k` in dictionary `d`. Returns `v` if `d` has no entry for `k`. Equivalent to `(dict-ref d k (lambda () v))`.

Examples:

```
> (dict-ref/default '([1 . one] [2 . two] [3 . three]) 2 'other)
'two
> (dict-ref/default '([1 . one] [2 . two] [3 . three]) 4 'other)
'other
```

```
(dict-ref/failure d k f) → any/c
  d : dict?
  k : any/c
  f : (-> any/c)
```

Looks up key `k` in dictionary `d`. Returns the result of applying `f` (in tail position) if `d` has no entry for `k`. Equivalent to `(dict-ref d k f)`.

Examples:

```
> (dict-ref/failure '([1 . one] [2 . two] [3 . three]) 2 gensym)
'two
> (dict-ref/failure '([1 . one] [2 . two] [3 . three]) 4 gensym)
'g3454
```

6.3 Dictionary Accessors

```
(dict-empty? d) → boolean?
  d : dict?
```

Reports whether *d* is empty (has no keys).

Examples:

```
> (dict-empty? '())
#t
> (dict-empty? '([1 . one] [2 . two]))
#f
```

6.4 Dictionary Combinations

```
(dict-union  d0
             d ...
            [#:combine combine
             #:combine/key combine/key])
 → (and/c dict? dict-can-functional-set?)
  d0 : (and/c dict? dict-can-functional-set?)
  d : dict?
  combine : (-> any/c any/c any/c)
          = (lambda _ (error 'dict-union ...))
  combine/key : (-> any/c any/c any/c any/c)
              = (lambda (k a b) (combine a b))
```

Computes the union of *d0* with each dictionary *d* by functional update, adding each element of each *d* to *d0* in turn. For each key `k` and value `v`, if a mapping from `k` to some value `v0` already exists, it is replaced with a mapping from `k` to `(combine/key k v0 v)`.

Examples:

```
> (dict-union '([1 . one]) '([2 . two]) '([3 . three]))
'((1 . one) (2 . two) (3 . three))
> (dict-union '([1    one uno]  [2    two dos])
              '([1    ein une]  [2    zwei deux])
              #:combine/key (lambda (k v1 v2) (append v1 v2)))
'((1 one uno ein une) (2 two dos zwei deux))
```

```
(dict-union!  d0
              d ...
             [#:combine combine
              #:combine/key combine/key]) → void?
  d0 : (and/c dict? dict-mutable?)
  d : dict?
  combine : (-> any/c any/c any/c)
          = (lambda _ (error 'dict-union! ...))
  combine/key : (-> any/c any/c any/c any/c)
              = (lambda (k a b) (combine a b))
```

Computes the union of *d0* with each dictionary *d* by mutable update, adding each element of each *d* to *d0* in turn. For each key k and value v, if a mapping from k to some value v0 already exists, it is replaced with a mapping from k to (*combine/key* k v0 v).

Examples:

```
(define d (make-hash))
> d
'#hash()
> (dict-union! d '([1    one uno]  [2    two dos]))
> d
'#hash((1 . (one uno)) (2 . (two dos)))
> (dict-union! d
               '([1    ein une]  [2    zwei deux])
               #:combine/key (lambda (k v1 v2) (append v1 v2)))
> d
'#hash((1 . (one uno ein une)) (2 . (two dos zwei deux)))
```

6.5 Dictionary Structure Properties

```
(wrapped-dict-property  #:unwrap unwrap
                       [#:wrap wrap
                        #:predicate pred
                        #:mutable? mutable?
                        #:weak? mutable?
                        #:functional? functional?]) → vector?
  unwrap : (-> (and/c dict? pred) dict?)
  wrap : (-> dict? (and/c dict? pred)) = (lambda (x) x)
  pred : (-> any/c boolean?) = (lambda (x) #t)
  mutable? : boolean? = weak?
  mutable? : boolean? = #f
  functional? : boolean? = #t
```

Produces a value appropriate for `prop:dict` for a derived dictionary type recognized by *pred*. Dictionaries constructed from this property will extract a nested dictionary using *unwrap* and will produce a wrapped dictionary during functional update using *wrap*.

Examples:

```
(define-struct table [dict]
  #:transparent
  #:property prop:dict
  (wrapped-dict-property
   #:unwrap (lambda (d) (table-dict d))
   #:wrap (lambda (d) (make-table d))
   #:predicate (lambda (d) (table? d))))
> (dict? (make-table '([1 . one] [2 . two])))
#t
> (dict-ref (make-table '([1 . one] [2 . two])) 1)
'one
> (dict-set (make-table '([1 . one] [2 . two])) 3 'three)
(table '((1 . one) (2 . two) (3 . three)))
```

6.6 Contracted Dictionaries

This library re-provides `dict/c` from `unstable/contract`.

7 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`.

8 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.

```
(try expr ...+)
```

The subsequent bindings were added by Carl Eastlund <cce@racket-lang.org>.

Executes the first expression *expr* in the sequence, producing its result value(s) if it returns any. If it raises an exception instead, `try` continues with the next *expr*. Exceptions raised by intermediate expressions are reported to the current logger at the `'debug` level before continuing. Exceptions raised by the final expression are not caught by `try`.

Examples:

```
> (try (+ 1 2) (+ 3 4))
3
> (try (+ 'one 'two) (+ 3 4))
7
> (try (+ 'one 'two) (+ 'three 'four))
+: expects type <number> as 1st argument, given: 'three;
other arguments were: 'four
```

9 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.

10 Functions

```
(require unstable/function)
```

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

This module provides tools for higher-order programming and creating functions.

10.1 Simple Functions

```
(identity x) → (one-of/c x)
  x : any/c
```

Returns `x`.

```
(thunk body ...)
```

Creates a function that ignores its inputs and evaluates the given body. Useful for creating event handlers with no (or irrelevant) arguments.

Examples:

```
(define f (thunk (define x 1) (printf "~a\n" x)))
> (f)
1
> (f 'x)
1
> (f #:y 'z)
1
```

10.2 Higher Order Predicates

```
((negate f) x ...) → boolean?
  f : (-> A ... boolean?)
  x : A
```

Negates the results of `f`; equivalent to `(not (f x ...))`.

This function is reprovided from `scheme/function`.

Examples:

```
(define f (negate exact-integer?))
> (f 1)
#f
> (f 'one)
#t
```

```
((conjoin f ...) x ...) → boolean?
  f : (-> A ... boolean?)
  x : A
```

Combines calls to each function with `and`. Equivalent to `(and (f x ...) ...)`

Examples:

```
(define f (conjoin exact? integer?))
> (f 1)
#t
> (f 1.0)
#f
> (f 1/2)
#f
> (f 0.5)
#f
```

```
((disjoin f ...) x ...) → boolean?
  f : (-> A ... boolean?)
  x : A
```

Combines calls to each function with `or`. Equivalent to `(or (f x ...) ...)`

Examples:

```
(define f (disjoin exact? integer?))
> (f 1)
#t
> (f 1.0)
#t
> (f 1/2)
#t
> (f 0.5)
#f
```

10.3 Currying and (Partial) Application

```
(call f x ...) → B
```

```
f : (-> A ... B)
x : A
```

Passes `x ...` to `f`. Keyword arguments are allowed. Equivalent to `(f x ...)`. Useful for application in higher-order contexts.

Examples:

```
> (map call
       (list + - * /)
       (list 1 2 3 4)
       (list 5 6 7 8))
'(6 -4 21 1/2)
(define count 0)
(define (inc)
  (set! count (+ count 1)))
(define (reset)
  (set! count 0))
(define (show)
  (printf "~a\n" count))
> (for-each call (list inc inc show reset show))
2
0
```

```
(papply f x ...) → (B ... -> C)
  f : (A ... B ... -> C)
  x : A
(papplyr f x ...) → (A ... -> C)
  f : (A ... B ... -> C)
  x : B
```

The `papply` and `papplyr` functions partially apply `f` to `x ...`, which may include keyword arguments. They obey the following equations:

```
((papply f x ...) y ...) = (f x ... y ...)
((papplyr f x ...) y ...) = (f y ... x ...)
```

Examples:

```
(define reciprocal (papply / 1))
> (reciprocal 3)
1/3
> (reciprocal 4)
1/4
(define halve (papplyr / 2))
> (halve 3)
3/2
> (halve 4)
```

```
  2
```

```
(curryn n f x ...) → (A1 ... -> ooo -> An ... -> B)
  n : exact-nonnegative-integer?
  f : (A0 ... A1 ... ooo An ... -> B)
  x : A0
(currynr n f x ...) → (An ... -> ooo -> A1 ... -> B)
  n : exact-nonnegative-integer?
  f : (A1 ... ooo An ... An+1 ... -> B)
  x : An+1
```

Note: The `ooo` above denotes a loosely associating ellipsis.

The `curryn` and `currynr` functions construct a curried version of `f`, specialized at `x ...`, that produces a result after `n` further applications. Arguments at any stage of application may include keyword arguments, so long as no keyword is duplicated. These curried functions obey the following equations:

```
(curryn 0 f x ...) = (f x ...)
((curryn (+ n 1) f x ...) y ...) = (curryn n f x ... y ...)

(currynr 0 f x ...) = (f x ...)
((currynr (+ n 1) f x ...) y ...) = (currynr n f y ... x ...)
```

The `call`, `papply`, and `papplyr` utilities are related to `curryn` and `currynr` in the following manner:

```
(call f x ...) = (curryn 0 f x ...) = (currynr 0 f x ...)
(papply f x ...) = (curryn 1 f x ...)
(papplyr f x ...) = (currynr 1 f x ...)
```

Examples:

```
(define reciprocal (curryn 1 / 1))
> (reciprocal 3)
1/3
> (reciprocal 4)
1/4
(define subtract-from (curryn 2 -))
(define from-10 (subtract-from 10))
> (from-10 5)
5
> (from-10 10)
0
(define from-0 (subtract-from 0))
> (from-0 5)
-5
```

```
> (from-0 10)
-10
(define halve (currynr 1 / 2))
> (halve 3)
3/2
> (halve 4)
2
(define subtract (currynr 2 -))
(define minus-10 (subtract 10))
> (minus-10 5)
-5
> (minus-10 10)
0
(define minus-0 (subtract 0))
> (minus-0 5)
5
> (minus-0 10)
10
```

10.4 Eta Expansion

`(eta f)`

Produces a function equivalent to *f*, except that *f* is evaluated every time it is called.

This is useful for function expressions that may be run, but not called, before *f* is defined. The `eta` expression will produce a function without evaluating *f*.

Examples:

```
(define f (eta g))
> f
#<procedure:eta>
(define g (lambda (x) (+ x 1)))
> (f 1)
2
```

`(eta* f x ...)`

Produces a function equivalent to *f*, with argument list *x* `...`. In simple cases, this is equivalent to `(lambda (x ...) (f x ...))`. Optional (positional or keyword) arguments are not allowed.

This macro behaves similarly to `eta`, but produces a function with statically known arity

which may improve efficiency and error reporting.

Examples:

```
(define f (eta* g x))
> f
#<procedure:f>
> (procedure-arity f)
1
(define g (lambda (x) (+ x 1)))
> (f 1)
2
```

10.5 Parameter Arguments

```
(lambda/parameter (param-arg ...) body ...)

     param-arg = param-arg-spec
               | keyword param-spec

param-arg-spec = id
               | [id default-expr]
               | [id #:param param-expr]
```

Constructs a function much like `lambda`, except that some optional arguments correspond to the value of a parameter. For each clause of the form `[id #:param param-expr]`, *param-expr* must evaluate to a value `param` satisfying `parameter?`. The default value of the argument *id* is `(param)`; `param` is bound to *id* via `parameterize` during the function call.

Examples:

```
(define p (open-output-string))
(define hello-world
  (lambda/parameter ([port #:param current-output-port])
    (display "Hello, World!")
    (newline port)))
> (hello-world p)
> (get-output-string p)
"Hello, World!\n"
```

11 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
```

```
(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)
```

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

The subsequent bindings were added by Sam Tobin-Hochstadt.

The subsequent bindings were added by Ryan Culpepper.

```
  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 d> #t)
(#<syntax:10:0 c> #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)
```

The subsequent bindings were added by David Van Horn.

```
(remf pred lst) → list?
```

```
pred : procedure?
lst : list?
```

Returns a list that is like *lst*, omitting the first element of *lst* for which *pred* produces a true value.

Example:

```
> (remf negative? '(1 -2 3 4 -5))
'(1 3 4 -5)
```

12 Net

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

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

12.1 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.

13 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*.

14 PLaneT Packages

```
(require unstable/planet)
```

This module provides tools relating to PLaneT packages. In addition to the binding described below, it provides `define-planet-package` and `this-package-in` from `unstable/require`, and `make-planet-path`, `syntax-source-planet-package`, `syntax-source-planet-package-owner`, `syntax-source-planet-package-name`, `syntax-source-planet-package-major`, `syntax-source-planet-package-minor`, and `syntax-source-planet-package-symbol` from `unstable/planet-syntax`.

```
(this-package-version-symbol)
(this-package-version-symbol path)
```

Produces a symbol corresponding to a `planet` module path for the current planet package, possibly with a ⟨*path*⟩ (from the grammar of `planet` module specs) into the package. This is similar to `this-package-version` and similar tools from `planet/util`.

15 Ports

```
(require unstable/port)
```

This module provides tools for port I/O.

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

```
(read-all [reader port]) → list?
  reader : (-> any/c) = read
  port : input-port? = (current-input-port)
```

This function produces a list of all the values produced by calling `(reader)` while `current-input-port` is set to `port`, up until it produces `eof`.

Examples:

```
> (read-all read (open-input-string "1 2 3"))
'(1 2 3)
> (parameterize ([current-input-port (open-input-string "a b c")])
    (read-all))
'(a b c)
```

```
(read-all-syntax [reader port]) → (syntax/c list?)
  reader : (-> (or/c syntax? eof-object?)) = read
  port : input-port? = (current-input-port)
```

This function produces a syntax object containing a list of all the syntax objects produced by calling `(reader)` while `current-input-port` is set to `port`, up until it produces `eof`. The source location of the result spans the entire portion of the port that was read.

Examples:

```
(define port1 (open-input-string "1 2 3"))
> (port-count-lines! port1)
> (read-all-syntax read-syntax port1)
#<syntax:1:0 (1 2 3)>
(define port2 (open-input-string "a b c"))
> (port-count-lines! port2)
> (parameterize ([current-input-port port2])
    (read-all-syntax))
#<syntax:1:0 (a b c)>
```

```
(port->srcloc port [source span]) → srcloc?
  port : port?
```

```
source : any/c = (object-name port)
span : exact-nonnegative-integer? = 0
```

Produces a srcloc structure representing the current position of a port, using the provided *source* and *span* values to fill in missing fields. This function relies on port-next-location, so line counting must be enabled for *port* to get meaningful results.

Examples:

```
(define port (open-input-string "1 2 3"))
> (port-count-lines! port)
> (read port)
1
> (port->srcloc port)
(srcloc 'string 1 1 2 0)
> (port->srcloc port "1 2 3" 1)
(srcloc "1 2 3" 1 1 2 1)
```

```
(read-available-bytes [port]) → (or/c bytes? eof-object?)
  port : input-port? = (current-input-port)
```

This function reads all immediately available bytes from a port and produces a byte string containing them. If there are no bytes available and the port is known to have no more input, it produces eof; if there are none available but the port may have more input, it produces an empty byte string. This procedure never blocks to wait for input from the port.

Examples:

```
(define-values [in out] (make-pipe))
> (parameterize ([current-input-port in]) (read-available-bytes))
#""
> (write-byte (char->integer #\c) out)
> (read-available-bytes in)
#"c"
> (read-available-bytes in)
#""
> (close-output-port out)
> (read-available-bytes in)
#<eof>
```

16 Pretty-Printing

```
(require unstable/pretty)
```

This module provides tools for pretty-printing.

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

```
(pretty-format/write x [columns]) → string?
  x : any/c
  columns : (or/c exact-nonnegative-integer? 'infinity)
          = (pretty-print-columns)
```

This procedure behaves like `pretty-format`, but it formats values consistently with `write` instead of `print`.

Examples:

```
> (struct both [a b] #:transparent)
> (pretty-format/write (list (both (list 'a 'b) (list "a" "b"))))
"(#(struct:both (a b) (\"a\" \"b\")))\n"
```

```
(pretty-format/display x [columns]) → string?
  x : any/c
  columns : (or/c exact-nonnegative-integer? 'infinity)
          = (pretty-print-columns)
```

This procedure behaves like `pretty-format`, but it formats values consistently with `display` instead of `print`.

Examples:

```
> (struct both [a b] #:transparent)
> (pretty-format/display (list (both (list 'a 'b) (list "a" "b"))))
"(#(struct:both (a b) (a b)))\n"
```

```
(pretty-format/print x [columns]) → string?
  x : any/c
  columns : (or/c exact-nonnegative-integer? 'infinity)
          = (pretty-print-columns)
```

This procedure behaves the same as `pretty-format`, but is named more explicitly to describe how it formats values. It is included for symmetry with `pretty-format/write` and `pretty-format/display`.

Examples:

```
> (struct both [a b] #:transparent)
> (pretty-format/print (list (both (list 'a 'b) (list "a" "b"))))
"(list (both '(a b) '(\"a\" \"b\")))\n"
```

17 Regular Expressions

```
(require unstable/regexp)
```

This module provides tools for building strings which can be compiled to regular expressions. In particular, the constructors wrap their arguments in appropriate delimeters to prevent misparsing after concatenation.

```
(regexp-sequence [#:start start
                  #:between between
                  #:end end]
                  re ...)           → string?
  start : string? = ""
  between : string? = ""
  end : string? = ""
  re : string?
```

Produces a regular expression string that matches *start*, followed by each *re* interleaved with *between*, followed by *end*.

Examples:

```
  (define re
    (pregexp
     (regexp-sequence "[0-9]+" "[0-9]+" "[0-9]+"
                      #:start (regexp-quote "(")
                      #:between (regexp-quote ",")
                      #:end (regexp-quote ")"))))
  > (regexp-match-exact? re "(1,10,100)")
  #t
  > (regexp-match-exact? re "(1,10)")
  #f
  > (regexp-match-exact? re " ( 1 , 10 , 100 ) ")
  #f
```

```
(regexp-or re ...+) → string?
  re : string?
```

Produces a regular expression string that matches any of the given *re*s.

Examples:

```
  (define re (pregexp (regexp-or "[0-9]+" "[a-z]")))
  > (regexp-match-exact? re "123")
  #t
  > (regexp-match-exact? re "c")
```

```
#t
> (regexp-match-exact? re "12c")
#f
```

```
(regexp-maybe re ...+) → string?
  re : string?
```

Produces a regular expression string that matches either the empty string, or the concatenation of all the given *re*s.

Examples:

```
(define re (pregexp (regexp-maybe "[0-9]+" "[.]" "[0-9]+")))
> (regexp-match-exact? re "123.456")
#t
> (regexp-match-exact? re "")
#t
> (regexp-match-exact? re "123")
#f
```

```
(regexp-star re ...+) → string?
  re : string?
```

Produces a regular expression string that matches zero or more consecutive occurrences of the concatenation of the given *re*s.

Examples:

```
(define re (pregexp (regexp-star "a" "b" "c")))
> (regexp-match-exact? re "")
#t
> (regexp-match-exact? re "abc")
#t
> (regexp-match-exact? re "abcabcabc")
#t
> (regexp-match-exact? re "a")
#f
```

```
(regexp-plus re ...+) → string?
  re : string?
```

Produces a regular expression string that matches one or more consecutive occurrences of the concatenation of the given *re*s.

Examples:

```
(define re (pregexp (regexp-plus "a" "b" "c")))
> (regexp-match-exact? re "")
```

```
#f
> (regexp-match-exact? re "abc")
#t
> (regexp-match-exact? re "abcabcabc")
#t
> (regexp-match-exact? re "a")
#f
```

`(regexp-save` *re* `...+)` → `string?`
re : `string?`

Produces a regular expression string that matches the concatenation of the given *re*s and saves the result.

Examples:

```
(define re
  (pregexp (regexp-sequence (regexp-save "[0-9]+") "\\1")))
> (regexp-match-exact? re "11")
#t
> (regexp-match-exact? re "123123")
#t
> (regexp-match-exact? re "123456")
#f
```

`(regexp-multi` *re* `...+)` → `string?`
re : `string?`

Produces a regular expression string that matches the concatenation of the given *re*s in multiple-line mode.

Examples:

```
(define re (pregexp (regexp-multi "^abc$")))
> (regexp-match? re "abc")
#t
> (regexp-match? re "xyz\nabc\ndef")
#t
```

18 Requiring Modules

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

This module provides tools for importing from modules.

```
(require/provide module-path ...)
```

Re-exports all bindings provided by each *module-path*. Equivalent to:

```
(require module-path ...)
(provide (all-from-out module-path ...))
```

```
(quote-require require-spec ...)
```

Produces the names exported by the *require-spec*s as a list of symbols.

Example:

```
> (quote-require racket/bool racket/function)
'(false true symbol=? false? boolean=? negate curryr curry const)
```

```
(define-planet-package name package)
```

Defines a shortcut *name* for importing modules from planet package *package*. Subsequently, (*name* module) is equivalent to (planet package/module) as a require path. For instance, to import the text and web modules from this package:

```
(define-planet-package my-package cce/scheme)
(require (my-package web) (my-package text))
```

The above require is equivalent to:

```
(require (planet cce/scheme/web) (planet cce/scheme/text))
```

```
(define-collection name collect)
```

Defines a shortcut *name* for importing modules from *collect* and its subcollections. Subsequently, (*name*) is equivalent to *collect* as a require path, and (*name* path) is equivalent to collect/path.

```
(define-collection macro syntax)
(require (macro parse))
```

The above require is equivalent to the below:

```
(require syntax/parse)
```

`(this-package-in` *path*`)`

This require transformer imports the file at *path* in the current planet package. For instance, in the package `(planet cce/scheme:7)`, writing:

```
(require (this-package-in function))
```

... is equivalent to writing:

```
(require (planet cce/scheme:7/function))
```

19 Sandboxed Evaluation

```
(require unstable/sandbox)
```

This module provides tools for sandboxed evaluation.

```
(make-trusted-evaluator language
                        input-program ...
                        #:requires requires
                        #:allow-read allow)
 → (any/c . -> . any)
  language : (or/c module-path?
                   (list/c 'special symbol?)
                   (cons/c 'begin list?))
  input-program : any/c
  requires : (listof (or/c module-path? path?))
  allow : (listof or/c module-path? path?)
(make-trusted-module-evaluator module-decl
                               #:language lang
                               #:allow-read allow)
 → (any/c . -> . any)
  module-decl : (or/c syntax? pair?)
  lang : (or/c #f module-path?)
  allow : (listof (or/c module-path? path?))
```

These procedures wrap calls to `make-evaluator` and `make-module-evaluator`, respectively, with `call-with-trusted-sandbox-configuration`.

```
(make-scribble-evaluator language
                         input-program ...
                         #:requires requires
                         #:allow-read allow)
 → (any/c . -> . any)
  language : (or/c module-path?
                   (list/c 'special symbol?)
                   (cons/c 'begin list?))
  input-program : any/c
  requires : (listof (or/c module-path? path?))
  allow : (listof or/c module-path? path?)
```

```
(make-scribble-module-evaluator module-decl
                                #:language lang
                                #:allow-read allow)
 → (any/c . -> . any)
  module-decl : (or/c syntax? pair?)
  lang : (or/c #f module-path?)
  allow : (listof (or/c module-path? path?))
```

These procedures wrap calls to `make-trusted-evaluator` and `make-trusted-module-evaluator`, respectively, with parameterizations setting `sandbox-output` and `sandbox-error-output` to `'string`.

```
(make-sandbox-namespace-specs make-ns
                              path ...)
 → (cons/c (-> namespace?) (listof module-path?))
  make-ns : (-> namespace?)
  path : module-path?
```

This function produces a value for the parameter `sandbox-namespace-specs` such that new sandbox evaluators start with a namespace constructed by *make-ns* and share a set of instances of the modules referred to by the given *path*s.

20 Scribble Documentation

```
(require unstable/scribble)
```

This module provides tools for Scribble documentation; specifically, of PLaneT packages. In addition to the bindings described below, this module provides `this-package-version-symbol` from `unstable/planet`, `this-package-in` from `unstable/require`, and `make-scribble-evaluator` and `make-scribble-module-evaluator` from `unstable/sandbox`.

```
(defmodule/this-package)
(defmodule/this-package #:use-sources [src-path ...] [src ...])
(defmodule/this-package path)
(defmodule/this-package path #:use-sources [src-path ...] [src ...])
```

This Scribble form corresponds to `defmodule` within a planet package. The displayed module path is a `planet` module path to the current planet package, possibly with a ⟨*path*⟩ (from the grammar of `planet` module specs) into the package. If the `#:use-sources` option is present, each *src-path* is similarly treated as a path into the current planet package, while each *src* is treated normally. Both sets of paths are concatenated and passed to the normal `defmodule`.

```
(defmodule*/no-declare/this-package [src-path ...] [src ...])
```

This Scribble form corresponds to `defmodule*/no-declare` within a planet package. The displayed module paths are `planet` module paths to the current planet package, possibly with ⟨*path*⟩s (from the grammar of `planet` module specs) into the package. Each *src-path* is similarly treated as a path into the current planet package, while each *src* is treated normally. Both sets of paths are concatenated and passed to the normal `defmodule*/no-declare`.

```
(schememodname/this-package)
(schememodname/this-package path)
```

This Scribble form corresponds to `schememodname` much like `defmodule/this-package` above corresponds to `defmodule`. The *path*, if present, is treated as a ⟨*path*⟩ (from the grammar of `planet` module specs) into the current planet package, and converted into a `planet` module spec.

```
(declare-exporting/this-package [mod-path ...] [mod ...])
(declare-exporting/this-package [mod-path ...] [mod ...]
                                #:use-sources [src-path ...] [src ...])
```

This Scribble form corresponds to `declare-exporting` much like `defmodule/this-package` above corresponds to `defmodule`. Each *mod-path* and *src-path* is treated as a ⟨*path*⟩ (from the grammar of `planet` module specs) into the current package. They are concatenated with the lists of *mod*s and *src*s, respectively, and passed to the normal `declare-exporting`.

21 Sets

```
(require unstable/set)
```

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

This module provides tools for representing finite sets.

```
(list->set lst) → set?
  lst : list?
(list->seteq lst) → set?
  lst : list?
(list->seteqv lst) → set?
  lst : list?
```

Produces the appropriate type of set containing the elements of the given list.

Examples:

```
  (define lst
    (list 'atom (expt 2 100) (list 'compound)
          'atom (expt 2 100) (list 'compound)))
  > (list->set lst)
  #<set: atom (compound) 126765060022822940149670320537 6>
  > (list->seteqv lst)
  #<set: atom (compound) (compound) 126765060022822940149670320537 6>
  > (list->seteq lst)
  #<set: atom (compound) (compound) 126765060022822940149670320537 6
126765060022822940149670320537 6>
```

```
(set=? a b) → boolean?
  a : set?
  b : set?
```

Reports whether two sets contain the same elements.

Examples:

```
  > (set=? (set 1) (set 1 2 3))
  #f
  > (set=? (set 1 2 3) (set 1))
  #f
  > (set=? (set 1 2 3) (set 1 2 3))
  #t
```

```
(proper-subset? a b) → boolean?
  a : set?
  b : set?
```

Reports whether `b` contains all of the elements of `a`, and at least one element not in `a`.

Examples:

```
> (proper-subset? (set 1) (set 1 2 3))
#t
> (proper-subset? (set 1 2 3) (set 1))
#f
> (proper-subset? (set 1 2 3) (set 1 2 3))
#f
```

```
(set->list s) → list?
  s : set?
```

Produces a list containing the elements of `s`.

Example:

```
> (set->list (set 1 2 3))
'(1 2 3)
```

```
(set-exclusive-or s ...+) → set?
  s : set?
```

Produces a set containing only those elements found in each `s` an odd number of times.

Example:

```
> (set-exclusive-or (set 1) (set 1 2) (set 1 2 3))
#<set: 1 3>
```

22 S-Expression Diff

```
(require unstable/sexp-diff)
```

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

```
(sexp-diff old-tree new-tree) → any/c
  old-tree : any/c
  new-tree : any/c
```

Takes two S-Expressions and returns their diff. Based on the Levenshtein distance for trees.

Example:

```
> (sexp-diff '(0 (1 2 3)) '(0 (4 2 3)))
'((0 (#:new 4 #:old 1 2 3)))
```

23 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.

23.1 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))
""
```

23.2 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
```

24 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.

25 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)
'()
```

26 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 ((a1 a) (b2 b) (c3 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 (x4) x4)>
```

(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-recorded-disappeared-uses)

```
 → (or/c (listof identifier?) false/c)
(current-recorded-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.

```
(syntax-local-value/record 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 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.

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

Add *ids* to `(current-recorded-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.)

This binding was added by Vincent St-Amour.

```
(format-unique-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-id`, but returned identifiers are guaranteed to be unique.

```
(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
```

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

The subsequent bindings were added by Sam Tobin-Hochstadt.

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)
```

The subsequent bindings were added by Carl Eastlund <cce@racket-lang.org>.

```
(syntax-list template ...)
```

This form constructs a list of syntax objects based on the given templates. It is equivalent to `(syntax->list #'(template ...))`.

Example:

```
  > (with-syntax ([(x ...) #'(1 2 3)])  (syntax-list x ...))
  '(#<syntax:3:0 1> #<syntax:3:0 2> #<syntax:3:0 3>)
```

```
(to-syntax  datum
            [#:stx stx
             #:src src
             #:ctxt ctxt
             #:prop prop
             #:cert cert]) → syntax?
  datum : any/c
  stx : (or/c false/c syntax?) = #f
  src : src/c = stx
```

```
ctxt : (or/c false/c syntax?) = stx
prop : (or/c false/c syntax?) = stx
cert : (or/c false/c syntax?) = stx
```

A wrapper for `datum->syntax` with keyword arguments.

The "master" keyword `#:stx` sets all attributes from a single syntax object, defaulting to `#f` for unadorned syntax objects.

The individual keywords `#:src`, `#:ctxt`, `#:prop`, and `#:cert` override `#:stx` for individual syntax object attributes. They control source src information, lexical context information, syntax object properties, and syntax certificates, respectively.

Examples:

```
(define blank-stx (to-syntax 'car))
> blank-stx
#<syntax car>
> (syntax-e blank-stx)
'car
> (free-identifier=? blank-stx #'car)
#f
(define full-stx (to-syntax 'car #:stx #'here))
> full-stx
#<syntax:7:0 car>
> (syntax-e full-stx)
'car
> (free-identifier=? full-stx #'car)
#t
(define partial-stx (to-syntax 'car #:ctxt #'here))
> partial-stx
#<syntax car>
> (syntax-e partial-stx)
'car
> (free-identifier=? partial-stx #'car)
#t
```

26.1 Syntax Object Source Locations

```
(syntax-source-directory stx) → (or/c path? #f)
  stx : syntax?
(syntax-source-file-name stx) → (or/c path? #f)
  stx : syntax?
```

These produce the directory and file name, respectively, of the path with which *stx* is asso-

ciated, or #f if *stx* is not associated with a path.

Examples:
```
(define loc
  (list (build-path "/tmp" "dir" "somewhere.ss")
        #f #f #f #f))
(define stx1 (datum->syntax #f 'somewhere loc))
> (syntax-source-directory stx1)
#<path:/tmp/dir/>
> (syntax-source-file-name stx1)
#<path:somewhere.ss>
(define stx2 (datum->syntax #f 'nowhere #f))
> (syntax-source-directory stx2)
#f
> (syntax-source-directory stx2)
#f
```

26.2 Macro Transformers

```
(redirect-transformer id) → (-> syntax? syntax?)
  id : identifier?
```

Constructs a function that behaves like a rename transformer; it does not cooperate with syntax-local-value like a rename transformer does, but unlike a rename transformer it may be used as a function to transform a syntax object referring to one identifier into a syntax object referring to another.

Examples:
```
> ((redirect-transformer #'x) #'a)
#<syntax:3:0 x>
> ((redirect-transformer #'y) #'(a b c))
#<syntax:4:0 (y b c)>
```

```
(head-expand stx [stop-list] intdef-ctx) → syntax?
  stx : syntax?
  stop-list : (listof identifier?) = null
  intdef-ctx : (or/c internal-definitions-context?
                     (non-empty-listof internal-definitions-context?)
                     #f)
```

This function performs head expansion on *stx*. In other words, it uses local-expand to expand *stx* until its head identifier is a core form (a member of (kernel-form-identifier-list)) or a member of *stop-list*, or until it can not be expanded further (e.g. due to

error).

It is equivalent to `(local-expand stx (syntax-local-context) (append stop-ids (kernel-form-identifier-list) intdef-ctx))`.

`(trampoline-transformer f) → (-> syntax? syntax?)`
  `f : (-> (-> syntax? void?) (-> syntax? syntax?) syntax? syntax?)`

Produces a transformer that can emit multiple results during macro expansion, to be spliced together via `begin`. This can be useful for compound expansion that relies on transformer definitions, as well as on expansion state that is difficult to marshall.

Specifically, `f` is invoked with three arguments. The first is the function used to emit intermediate results (other than the last one). The second applies the syntax mark used for the entire expansion; `syntax-local-introduce` will not be reliable during this process. The third is the syntax object to expand.

Examples:

```
> (define-syntax magic-begin
    (trampoline-transformer
     (lambda (emit intro stx)
       (syntax-case stx ()
         [(_ term ...)
          (let loop ([terms (syntax->list #'(term ...))])
            (cond
             [(null? terms) #'(begin)]
             [(null? (cdr terms)) (car terms)]
             [else
              (printf "Presto: ~s!\n"
                      (syntax->datum (car terms)))
              (emit (car terms))
              (loop (cdr terms))]))])))))
(magic-begin
 (define x 1)
 (define y 2)
 (+ x y))
Presto: (define x 1)!
Presto: (define y 2)!
3
```

`(quote-transformer x) → syntax?`
  `x : any/c`

Produces a syntax object representing an expression that reconstructs `x` when executed, including faithfully reconstructing any syntax objects contained in `x`. Note that `quote` nor-

mally converts syntax objects to non-syntax data, and `quote-syntax` does the opposite.

Examples:

```
> (define-for-syntax x (list 1 #'(2 3) 4))
> (define-syntax (the-many-faces-of-x stx)
    (with-syntax ([x x] [qx (quote-transformer x)])
     #'(list 'x
             (quote-syntax x)
             qx)))
> (the-many-faces-of-x)
'((1 (2 3) 4) #<syntax (1 (2 3) 4)> (1 #<syntax:3:0 (2 3)> 4))
```

27 Planet Package Macros

```
(require unstable/planet-syntax)
```

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

```
(syntax-source-planet-package stx)
 → (or/c (list/c string?
                 string?
                 exact-nonnegative-integer?
                 exact-nonnegative-integer?)
         #f)
  stx : syntax?
(syntax-source-planet-package-owner stx) → (or/c string? #f)
  stx : syntax?
(syntax-source-planet-package-name stx) → (or/c string? #f)
  stx : syntax?
(syntax-source-planet-package-major stx)
 → (or/c exact-nonnegative-integer? #f)
  stx : syntax?
(syntax-source-planet-package-minor stx)
 → (or/c exact-nonnegative-integer? #f)
  stx : syntax?
(syntax-source-planet-package-symbol  stx
                                      [text]) → (or/c symbol? #f)
  stx : syntax?
  text : (or/c text? #f) = #f
```

These functions extract the planet package with which *stx* is associated, if any, based on its source location information and the currently installed set of planet packages. They produce, respectively, the planet package s-expression, its owner, name, major version number, minor version number, or a symbol corresponding to a `planet` module path. They each produce `#f` if *stx* is not associated with a planet package.

Examples:

```
(define loc
  (list (build-path (current-directory) "file.ss")
        #f #f #f #f))
(define stx (datum->syntax #f 'stx loc))
> (syntax-source-planet-package stx)
#f
> (syntax-source-planet-package-owner stx)
#f
> (syntax-source-planet-package-name stx)
```

```
#f
> (syntax-source-planet-package-major stx)
#f
> (syntax-source-planet-package-minor stx)
#f
> (syntax-source-planet-package-symbol stx)
#f
> (syntax-source-planet-package-symbol stx "there")
#f
```

```
(make-planet-path stx id) → syntax?
  stx : syntax?
  id : (or/c identifier? #f)
```

Constructs a syntax object representing a require spec for the planet package from which *stx* arises, with suffix *id* (if any).

Examples:

```
(define loc
  (list (build-path (current-directory) "file.ss")
        #f #f #f #f))
(define stx (datum->syntax #f 'stx loc))
> (make-planet-path stx #f)
#<syntax (planet #f)>
> (make-planet-path stx #'there)
#<syntax:5:0 (planet #f)>
```

28 Text Representations

```
(require unstable/text)
```

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

This module provides tools for manipulating and converting textual data.

28.1 Contracts and Predicates

```
text/c : flat-contract?
(text? v) → boolean?
  v : any/c
```

This contract and predicate recognize text values: strings, byte strings, symbols, and keywords, as well as syntax objects containing them.

Examples:

```
> (text? "text")
#t
> (text? #"text")
#t
> (text? 'text)
#t
> (text? '#:text)
#t
> (text? #'"text")
#t
> (text? #'#"text")
#t
> (text? #'text)
#t
> (text? #'#:text)
#t
> (text? '(not text))
#f
```

```
(string-literal? v) → boolean?
  v : any/c
(bytes-literal? v) → boolean?
  v : any/c
```

```
(keyword-literal? v) → boolean?
  v : any/c
```

These predicates recognize specific text types stored in syntax objects.

Examples:

```
> (string-literal? #'"literal")
#t
> (string-literal? "not literal")
#f
> (bytes-literal? #'#"literal")
#t
> (bytes-literal? #"not literal")
#f
> (keyword-literal? #'#:literal)
#t
> (keyword-literal? '#:not-literal)
#f
```

28.2 Text Conversions and Concatenation

```
(text->string [#:before before
               #:between between
               #:after after]
               text ...)          → string?
  before : text/c = ""
  between : text/c = ""
  after : text/c = ""
  text : text/c
(text->bytes [#:before before
              #:between between
              #:after after]
              text ...)          → bytes?
  before : text/c = ""
  between : text/c = ""
  after : text/c = ""
  text : text/c
```

```
(text->symbol [#:before before
               #:between between
               #:after after]
               text ...)          → symbol?
  before : text/c = ""
  between : text/c = ""
  after : text/c = ""
  text : text/c
(text->keyword [#:before before
                #:between between
                #:after after]
                text ...)          → keyword?
  before : text/c = ""
  between : text/c = ""
  after : text/c = ""
  text : text/c
```

These functions convert text values to specific types. They concatenate each *text* argument, adding *before* and *after* to the front and back of the result and *between* between each argument.

Examples:

```
  > (text->string #"concat" #'enate)
  "concatenate"
  > (text->bytes #:between "-" 'concat #'#:enate)
  #"concat-enate"
  > (text->symbol #:before "(" #:after ")" '#:concat #'"enate")
  '|(concatenate)|
  > (text->keyword #:before #'< #:between #'- #:after #'> "concat" #'#"enate")
  '#:<concat-enate>
```

```
(text->string-literal [#:before before
                       #:between between
                       #:after after
                       #:stx stx]
                       text ...)          → string-literal?
  before : text/c = ""
  between : text/c = ""
  after : text/c = ""
  stx : (or/c syntax? false/c) = #f
  text : text/c
```

```
(text->bytes-literal [#:before before
                     #:between between
                     #:after after
                     #:stx stx]
                     text ...)         → bytes-literal?
  before : text/c = ""
  between : text/c = ""
  after : text/c = ""
  stx : (or/c syntax? false/c) = #f
  text : text/c
(text->identifier [#:before before
                   #:between between
                   #:after after
                   #:stx stx]
                   text ...)         → identifier?
  before : text/c = ""
  between : text/c = ""
  after : text/c = ""
  stx : (or/c syntax? false/c) = #f
  text : text/c
(text->keyword-literal [#:before before
                        #:between between
                        #:after after
                        #:stx stx]
                        text ...)         → keyword-literal?
  before : text/c = ""
  between : text/c = ""
  after : text/c = ""
  stx : (or/c syntax? false/c) = #f
  text : text/c
```

These functions convert text values to specific syntax object types, deriving syntax object properties from the *stx* argument. They concatenate each *text* argument, adding *before* and *after* to the front and back of the result and *between* between each argument.

Examples:

```
> (text->string-literal #"concat" #'enate)
#<syntax "concatenate">
> (text->bytes-literal #:between "-" 'concat #'#:enate)
#<syntax #"concat-enate">
> (text->identifier #:before "(" #:after ")"
                     #:stx #'props
                    '#:concat #'"enate")
#<syntax:4:0 |(concatenate)|>
> (text->keyword-literal #:before #'< #:between #'- #:after #'>
                         #:stx #'props
```

```
                          "concat" #'#"enate")
  #<syntax:5:0 #:<concat-enate>>
```

28.3 Text Comparisons

```
(text=? one two) → boolean?
  one : text/c
  two : text/c
(text<? one two) → boolean?
  one : text/c
  two : text/c
(text<=? one two) → boolean?
  one : text/c
  two : text/c
(text>? one two) → boolean?
  one : text/c
  two : text/c
(text>=? one two) → boolean?
  one : text/c
  two : text/c
```

These predicates compare the character content of two text values. They are equivalent to:

```
  (text=? one two) = (string=? (text->string one) (text-
>string two))
  (text<? one two) = (string<? (text->string one) (text-
>string two))
  (text<=? one two) = (string<=? (text->string one) (text-
>string two))
  (text>? one two) = (string>? (text->string one) (text-
>string two))
  (text>=? one two) = (string>=? (text->string one) (text-
>string two))
```

Examples:

```
  > (text=? #"x" #'y)
  #f
  > (text<? #"x" #'y)
  #t
  > (text<=? #"x" #'y)
  #t
  > (text>? #"x" #'y)
  #f
  > (text>=? #"x" #'y)
```

```
#f
```

29 Multiple Values

```
(require unstable/values)
```

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

This module provides tools for manipulating functions and expressions that produce multiple values.

```
(values->list expr)
```

Produces a list of the values returned by *expr*.

Example:

```
> (values->list (values 1 2 3))
'(1 2 3)
```

```
(map2 f lst ...) → (listof B) (listof C)
  f : (-> A ... (values B C))
  lst : (listof A)
```

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

Example:

```
> (map2 (lambda (x) (values (+ x 1) (- x 1))) (list 1 2 3))
'(2 3 4)
'(0 1 2)
```

```
(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)
```

```
(foldr/values f vs lst ...) → B ...
  f : (-> A ... B ... (values B ...))
  vs : (list/c B ...)
  lst : (listof A)
(foldl/values f vs lst ...) → B ...
  f : (-> A ... B ... (values B ...))
  vs : (list/c B ...)
  lst : (listof A)
```

These functions combine the values in the lists *lst* ... using the multiple-valued function *f*; `foldr/values` traverses the lists right to left and `foldl/values` traverses left to right.

Examples:

```
(define (add/cons a b c d)
  (values (+ a c) (cons b d)))
> (foldr/values add/cons (list 0 null)
                (list 1 2 3 4) (list 5 6 7 8))
10
'(5 6 7 8)
> (foldl/values add/cons (list 0 null)
                (list 1 2 3 4) (list 5 6 7 8))
10
'(8 7 6 5)
```

30 XML and CSS

```
(require unstable/web)
```

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

This module provides tools for programmatic creation of static web pages. It is based on the XML collection; see documentation for xexpr?.

```
css/c : flat-contract?
(css? v) → boolean?
  v : any/c
```

This contract and predicate pair recognizes CSS-expressions, which are described by the following grammar:

```
       css = (list style ...)

  style-def = (cons selector (list property ...))

   property = (list name value)

   selector = text

       name = text

      value = text
```

Here, text is any of the datatypes described in §28 “Text Representations”.

```
xexpr/c : flat-contract?
```

This flat contract corresponds to xexpr?. It is reprovided from xml. In versions of PLT Scheme before the implementation of xexpr/c, this module provides its own definition.

```
(write-css css [out]) → void?
  css : css/c
  out : output-port? = (current-output-port)
(write-xexpr css [out]) → void?
  css : css/c
  out : output-port? = (current-output-port)
```

These functions write CSS-expressions and X-expressions, respectively, to output ports, by

their canonical text representations.

```
(create-stylesheet file css) → void?
  file : path-string?
  css : css/c
(create-webpage file xexpr) → void?
  file : path-string?
  xexpr : xexpr/c
```

These functions write style sheets (represented as CSS-expressions) or webpages (represented as X-expressions) to files.

31 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.

32 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 [dict]) → dict?
  stx : syntax?
  dict : dict? = (make-immutable-free-id-table)
```

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. Each mutated variable is added to `dict`, mapped to `#t`. If `dict` is mutable, as determined by `dict-mutable?`, then the table is updated destructively. Otherwise, the table is updated functionally.

Examples:

```
> (define t (find-mutated-vars #'(begin (set! var 'foo) 'bar)))
> (dict-ref t #'var #f)
#t
> (dict-ref t #'other-var #f)
#f
> (define tbl (make-free-id-table))
> (find-mutated-vars #'(begin (set! var 'foo) 'bar) tbl)
#<mutable-free-id-table>
> (dict-ref tbl #'var #f)
#t
```

}

33 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
```

34 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.

35 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)
```

36 Hash Tables

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

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

This module provides tools for manipulating hash tables.

36.1 Hash Table Lookup

```
(hash-ref/check h k) → any/c
  h : hash?
  k : (lambda (k) (hash-has-key? h k))
```

Looks up key `k` in hash table `h`. Raises a contract error if `h` has no entry for `k`. Equivalent to `(hash-ref h k)`, except for the specific exception value raised.

Example:

```
  > (hash-ref/check (make-immutable-hash '([1 . one] [2 . two] [3 .
three])) 2)
  'two
```

```
(hash-ref/identity h k) → any/c
  h : hash?
  k : any/c
```

Looks up key `k` in hash table `h`. Returns `k` if `h` has no entry for `k`. Equivalent to `(hash-ref h k (lambda () k))`.

Examples:

```
  > (hash-ref/identity (make-immutable-hash '([1 . one] [2 .
two] [3 . three])) 2)
  'two
  > (hash-ref/identity (make-immutable-hash '([1 . one] [2 .
two] [3 . three])) 4)
  4
```

```
(hash-ref/default h k v) → any/c
  h : hash?
  k : any/c
  v : any/c
```

Looks up key *k* in hash table *h*. Returns *v* if *h* has no entry for *k*. Equivalent to (hash-ref *h k* (lambda () *v*)).

Examples:

```
  > (hash-ref/default (make-immutable-hash '([1 . one] [2 . two] [3 .
three])) 2 'other)
  'two
  > (hash-ref/default (make-immutable-hash '([1 . one] [2 . two] [3 .
three])) 4 'other)
  'other
```

```
(hash-ref/failure h k f) → any/c
  h : hash?
  k : any/c
  f : (-> any/c)
```

Looks up key *k* in hash table *h*. Returns the result of applying *f* (in tail position) if *h* has no entry for *k*. Equivalent to (hash-ref *h k f*).

Examples:

```
  > (hash-ref/failure (make-immutable-hash '([1 . one] [2 . two] [3 .
three])) 2 gensym)
  'two
  > (hash-ref/failure (make-immutable-hash '([1 . one] [2 . two] [3 .
three])) 4 gensym)
  'g1946
```

36.2 Hash Table Combinations

```
(hash-union  h0
             h ...
            [#:combine combine
             #:combine/key combine/key])
 → (and/c hash? hash-can-functional-set?)
  h0 : (and/c hash? hash-can-functional-set?)
  h : hash?
  combine : (-> any/c any/c any/c)
          = (lambda _ (error 'hash-union ...))
  combine/key : (-> any/c any/c any/c any/c)
              = (lambda (k a b) (combine a b))
```

Computes the union of *h0* with each hash table *h* by functional update, adding each element of each *h* to *h0* in turn. For each key k and value v, if a mapping from k to some value v0

already exists, it is replaced with a mapping from k to (*combine/key* k v0 v).

Examples:

```
  > (hash-union (make-immutable-hash '([1 . one])) (make-immutable-
hash '([2 . two])) (make-immutable-hash '([3 . three])))
  '#hash((1 . one) (2 . two) (3 . three))
  > (hash-union (make-immutable-hash '([1    one uno]  [2    two dos]))
                (make-immutable-hash '([1    ein une]  [2    zwei deux]))
                #:combine/key (lambda (k v1 v2) (append v1 v2)))
  '#hash((1 . (one uno ein une)) (2 . (two dos zwei deux)))
```

```
(hash-union! h0
             h ...
             [#:combine combine
              #:combine/key combine/key]) → void?
  h0 : (and/c hash? hash-mutable?)
  h : hash?
  combine : (-> any/c any/c any/c)
          = (lambda _ (error 'hash-union ...))
  combine/key : (-> any/c any/c any/c any/c)
              = (lambda (k a b) (combine a b))
```

Computes the union of *h0* with each hash table *h* by mutable update, adding each element of each *h* to *h0* in turn. For each key k and value v, if a mapping from k to some value v0 already exists, it is replaced with a mapping from k to (*combine/key* k v0 v).

Examples:

```
  (define h (make-hash))
  > h
  '#hash()
  > (hash-union! h (make-immutable-hash '([1    one uno]  [2    two dos])))
  > h
  '#hash((1 . (one uno)) (2 . (two dos)))
  > (hash-union! h
                 (make-immutable-hash '([1    ein une]  [2    zwei deux]))
                 #:combine/key (lambda (k v1 v2) (append v1 v2)))
  > h
  '#hash((1 . (one uno ein une)) (2 . (two dos zwei deux)))
```

37 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
```

The subsequent bindings were added by Carl Eastlund <cce@racket-lang.org>.

```
(match? val-expr pat ...)
```

Returns `#t` if the result of *val-expr* matches any of *pat*, and returns `#f` otherwise.

Examples:

```
> (match? (list 1 2 3)
    (list a b c)
    (vector x y z))
#t
> (match? (vector 1 2 3)
    (list a b c)
    (vector x y z))
#t
> (match? (+ 1 2 3)
    (list a b c)
    (vector x y z))
#f
```

(as ([*lhs-id rhs-expr*] ...) *pat* ...)

As a match expander, binds each *lhs-id* as a pattern variable with the result value of *rhs-expr*, and continues matching each subsequent *pat*.

Example:

```
> (match (list 1 2 3)
    [(as ([a 0]) (list b c d)) (list a b c d)])
'(0 1 2 3)
```

38 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
```

39 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.

40 Debugging

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

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

This module provides macros and functions for printing out debugging information.

```
(debug options ... expr)

options = #:name name-expr
        | #:source srcloc-expr
```

Writes debugging information about the evaluation of *expr* to the current error port. The name and source location of the expression may be overridden by keyword options; their defaults are the syntactic form of the expression and its syntactic source location, respectively.

Examples:

```
> (debug 0)
>> eval:2.0: 0
   result: 0
<< eval:2.0: 0
0
> (debug #:name "one, two, three" (values 1 2 3))
>> eval:3.0: "one, two, three"
   results: (values 1 2 3)
<< eval:3.0: "one, two, three"
1
2
3
> (debug #:source (make-srcloc 'here 1 2 3 4)
    (error 'function "something went wrong"))
>> here:1.2: (error 'function "something went wrong")
   raised exception: function: something went wrong
<< here:1.2: (error 'function "something went wrong")
function: something went wrong
```

```
(dprintf fmt arg ...) → void?
  fmt : string?
  arg : any/c
```

Constructs a message in the same manner as `format` and writes it to `(current-error-port)`, with indentation reflecting the number of nested `debug` forms.

Examples:

```
> (dprintf "level: ~a" 0)
level: 0
> (debug (dprintf "level: ~a" 1))
>> eval:3.0: (dprintf "level: ~a" 1)
  level: 1
  result: #<void>
<< eval:3.0: (dprintf "level: ~a" 1)
> (debug (debug (dprintf "level: ~a" 2)))
>> eval:4.0: (debug (dprintf "level: ~a" 2))
  >> eval:4.0: (dprintf "level: ~a" 2)
    level: 2
    result: #<void>
  << eval:4.0: (dprintf "level: ~a" 2)
  result: #<void>
<< eval:4.0: (debug (dprintf "level: ~a" 2))
```

```
(debugf function-expr argument ...)

argument = argument-expr
         | argument-keyword argument-expr
```

Logs debugging information for `(#%app function-expr argument ...)`, including the evaluation and results of the function and each argument.

Example:

```
> (debugf + 1 2 3)
>> eval:2.0: debugf
  >> eval:2.0: +
    result: #<procedure:+>
  << eval:2.0: +
  >> eval:2.0: 1
    result: 1
  << eval:2.0: 1
  >> eval:2.0: 2
    result: 2
  << eval:2.0: 2
  >> eval:2.0: 3
    result: 3
  << eval:2.0: 3
  result: 6
<< eval:2.0: debugf
6
```

```
(begin/debug expr ...)
```

```
(define/debug id expr)
(define/debug (head args) body ...+)
(define/private/debug id expr)
(define/private/debug (head args) body ...+)
(define/public/debug id expr)
(define/public/debug (head args) body ...+)
(define/override/debug id expr)
(define/override/debug (head args) body ...+)
(define/augment/debug id expr)
(define/augment/debug (head args) body ...+)
(let/debug ([lhs-id rhs-expr] ...) body ...+)
(let/debug loop-id ([lhs-id rhs-expr] ...) body ...+)
(let*/debug ([lhs-id rhs-expr] ...) body ...+)
(letrec/debug ([lhs-id rhs-expr] ...) body ...+)
(let-values/debug ([(lhs-id ...) rhs-expr] ...) body ...+)
(let*-values/debug ([(lhs-id ...) rhs-expr] ...) body ...+)
(letrec-values/debug ([(lhs-id ...) rhs-expr] ...) body ...+)
(with-syntax/debug ([pattern stx-expr] ...) body ...+)
(with-syntax*/debug ([pattern stx-expr] ...) body ...+)
(parameterize/debug ([param-expr value-expr] ...) body ...+)
```

These macros add logging based on `debug` to the evaluation of expressions in `begin`, `define`, `define/private`, `define/public`, `define/override`, `define/augment`, `let`, `let*`, `letrec`, `let-values`, `let*-values`, `letrec-values`, `with-syntax`, `with-syntax*`, and `parameterize`.

41 Contracts for macro subexpressions

This library provides a procedure `wrap-expr/c` for applying contracts to macro subexpressions.

```
(require unstable/wrapc)
```

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

```
(wrap-expr/c  contract-expr
              expr
             [#:positive pos-blame
              #:negative neg-blame
              #:name expr-name
              #:macro macro-name
              #:context context])   → syntax?
  contract-expr : syntax?
  expr : syntax?
  pos-blame : (or/c syntax? string? module-path-index?
                    'from-macro 'use-site 'unknown)
            = 'use-site
  neg-blame : (or/c syntax? string? module-path-index?
                    'from-macro 'use-site 'unknown)
            = 'from-macro
  expr-name : (or/c identifier? symbol? string? #f) = #f
  macro-name : (or/c identifier? symbol? string? #f) = #f
  context : (or/c syntax? #f) = (current-syntax-context)
```

Returns a syntax object representing an expression that applies the contract represented by *contract-expr* to the value produced by *expr*.

The contract's positive blame represents the obligations of the expression being wrapped. The negative blame represents the obligations of the macro imposing the contract—the ultimate caller of `wrap-expr/c`. By default, the positive blame is taken as the module currently being expanded, and the negative blame is inferred from the definition site of the macro (itself inferred from the *context* argument). But both blame locations can be overridden.

Positive and negative blame locations are determined from *pos-blame* and *neg-blame*, respectively, as follows:

- If the argument is a string, it is used directly as the blame label.
- If the argument is syntax, its source location is used to produce the blame label.
- If the argument is a module path index, its resolved module path is used.

- If the argument is `'from-macro`, the macro is inferred from either the *macro-name* argument (if *macro-name* is an identifier) or the *context* argument, and the module where it is *defined* is used as the negative blame location. If neither an identifier *macro-name* nor a *context* argument is given, the location is `"unknown"`.
- If the argument is `'same-as-use-site`, the module being expanded is used.
- If the argument is `'unknown`, the blame label is `"unknown"`.

The *macro-name* argument is used to determine the macro's binding, if it is an identifier. If *expr-name* is given, *macro-name* is also included in the contract error message. If *macro-name* is omitted or `#f`, but *context* is a syntax object, then *macro-name* is determined from *context*.

If *expr-name* is not `#f`, it is used in the contract's error message to describe the expression the contract is applied to.

The *context* argument is used, when necessary, to infer the macro name for the negative blame party and the contract error message. The *context* should be either an identifier or a syntax pair with an identifer in operator position; in either case, that identifier is taken as the macro ultimately requesting the contract wrapping.

Examples:

```
> (define-syntax (myparameterize1 stx)
    (syntax-case stx ()
      [(_ ((p v)) body)
       (with-syntax ([cp (wrap-expr/c
                          #'parameter? #'p
                          #:name "the parameter argument"
                          #:context stx)])
         #'(parameterize ((cp v)) body))]))
> (myparameterize1 ((current-input-port
                     (open-input-string "(1 2 3)")))
    (read))
'(1 2 3)
> (myparameterize1 (('whoops 'something))
    'whatever)
eval:4.0: top-level broke the contract parameter? on
  the parameter argument of myparameterize1
; expected <parameter?>, given: 'whoops
> (module mod racket
    (require (for-syntax unstable/wrapc))
    (define-syntax (app stx)
      (syntax-case stx ()
        [(app f arg)
         (with-syntax ([cf (wrap-expr/c
                            #'(-> number? number?)
```

```
                              #'f
                              #:name "the function argument"
                              #:context stx)])
           #'(cf arg))]))
    (provide app))
> (require 'mod)
> (app add1 5)
6
> (app add1 'apple)
```
eval:8.0: (quote mod) broke the contract (-> number? number?) on the function argument of app given to top-level; expected <number?>, given: 'apple
```
> (app (lambda (x) 'pear) 5)
```
eval:9.0: top-level broke the contract (-> number? number?) on the function argument of app; expected <number?>, given: 'pear

42 Contracts for struct type properties

```
(require unstable/prop-contract)
```

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

(struct-type-property/c *value-contract*) → contract?
  value-contract : contract?

Produces a contract for struct type properties. When the contract is applied to a struct type property, it produces a wrapped struct type property. When the wrapped struct type property is used to create a new struct type (via `struct`, `make-struct-type`, etc), it applies *value-contract* to the value associated with the property.

The contract has no effect on the struct type property accessor.

Examples:

```
> (define-values (prop prop? prop-ref)
    (make-struct-type-property 'prop))
> (define/contract wrapped
      (struct-type-property/c (-> any/c (-> number? number?)))
    prop)
> (struct s (f)
    #:property wrapped (lambda (s) (s-f s)))
> (define (get-f s) ((prop-ref s) s))
> (define s1 (s add1))
> ((get-f s1) 5)
6
> ((get-f s1) 'apple)
eval:3.0: (definition wrapped) broke the contract
  (struct-type-property/c
   (-> any/c (-> number? number?)))
 on wrapped; expected <number?>, given: 'apple
> (define s2 (s (lambda (n) (if (zero? n) 'zero 'nonzero))))
> ((get-f s2) 5)
eval:3.0: top-level broke the contract
  (struct-type-property/c
   (-> any/c (-> number? number?)))
 on wrapped; expected <number?>, given: 'nonzero
> ((get-f s2) 'apple)
eval:3.0: (definition wrapped) broke the contract
  (struct-type-property/c
   (-> any/c (-> number? number?)))
 on wrapped; expected <number?>, given: 'apple
```

43 GUI libraries

43.1 DrRacket Language Levels

```
(require unstable/gui/language-level)
```

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

`language-level@ : unit?`

This unit imports `drracket:tool^` and exports `language-level^`.

`language-level^ : signature`

```
(make-language-level  name
                      path
                      mixin ...
                     [#:number number
                      #:hierarchy hierarchy
                      #:summary summary
                      #:url url
                      #:reader reader])
 → (object-provides/c drracket:language:language<%>)
  name : string?
  path : module-path?
  mixin : (-> class? class?)
  number : integer? = ...
  hierarchy : (listof (cons/c string? integer?)) = ...
  summary : string? = name
  url : (or/c string? #f) = #f
  reader : (->* [] [any/c input-port?] (or/c syntax? eof-object?))
         = read-syntax
```

Constructs a language level as an instance of `drracket:language:language<%>` with the given *name* based on the language defined by the module at *path*. Applies `(drracket:language:get-default-mixin)` and the given *mixin*s to `simple-language-level%` to construct the class, and uses the optional keyword arguments to fill in the language's description and reader.

```
simple-language-level% : (class-provides/c drracket:language:language<%>
                                           drracket:language:module-based-language<%>
                                           drracket:language:simple-module-based-language<%>)
```

Equal to (drracket:language:module-based-language->language-mixin (drracket:language:simple-module-based-language->module-based-language-mixin drracket:language:simple-module-based-language%)).

```
(language-level-render-mixin to-sexp
                             show-void?)
 → (mixin-provides/c [drracket:language:language<%>] [])
  to-sexp : (-> any/c any/c)
  show-void? : boolean?
```

Produces a mixin that overrides render-value/format to apply *to-sexp* to each value before printing it, and to skip void? values (pre-transformation) if *show-void?* is #f.

```
(language-level-capability-mixin dict)
 → (mixin-provides/c [drracket:language:language<%>] [])
  dict : dict?
```

Produces a mixin that augments capability-value to look up each key in *dict*, producing the corresponding value if the key is found and deferring to inner otherwise.

```
language-level-no-executable-mixin : (mixin-provides/c [drracket:language:language<%>] [])
```

Overrides create-executable to print an error message in a dialog box.

```
language-level-eval-as-module-mixin : (mixin-provides/c [drracket:language:language<%>
                                                         drracket:language:module-based-langua
                                                        [])
```

Overrides front-end/complete-program to wrap terms from the definition in a module based on the language level's definition module. This duplicates the behavior of the HtDP teaching languages, for instance.

```
language-level-macro-stepper-mixin : (mixin-provides/c [drracket:language:language<%>]
                                                       [])
```

This mixin enables the macro stepper for its language level.

```
language-level-check-expect-mixin : (mixin-provides/c [drracket:language:language<%>] [])
```

This mixin overrides on-execute to set up the check-expect test engine to a language level similarly to the HtDP teaching languages.

```
(language-level-metadata-mixin reader-module
                               meta-lines
                               meta->settings
                               settings->meta)
 → (mixin-provides/c [drracket:language:language<%>] [])
  reader-module : module-path?
  meta-lines : exact-nonnegative-integer?
  meta->settings : (-> string? any/c any/c)
  settings->meta : (-> symbol? any/c string?)
```

This mixin constructs a language level that stores metadata in saved files allowing Drracket to automatically switch back to this language level upon opening them. It overrides `get-reader-module`, `get-metadata`, `metadata->settings`, and `get-metadata-lines`.

The resulting language level uses the reader from *reader-module*, and is recognized in files that start with a reader directive for that module path within the first *meta-lines* lines. Metadata about the language's settings is marshalled between a string and a usable value (based on a default value) by *meta->settings*, and between a usable value for a current module (with a symbolic name) by *settings->meta*.

43.2 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.

43.3 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*.

43.4 Slideshow Presentations

```
(require unstable/gui/slideshow)
```

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

43.4.1 Text Formatting

```
(with-size size expr)
```

Sets `current-font-size` to *size* while running *expr*.

```
(with-scale scale expr)
```

Multiplies `current-font-size` by *scale* while running *expr*.

```
(big text)
(small text)
```

Scale `current-font-size` by `3/2` or `2/3`, respectively, while running *text*.

```
(with-font font expr)
```

Sets `current-main-font` to *font* while running *expr*.

(with-style *style* *expr*)

Adds *style* to current-main-font (via cons) while running *expr*.

(bold *text*)
(italic *text*)
(subscript *text*)
(superscript *text*)
(caps *text*)

Adds the attributes for bold, italic, superscript, subscript, or small caps text, respectively, to current-main-font while running *text*.

43.4.2 Pict Colors

(color *c* *p*) → pict?
  c : color/c
  p : pict?

Applies color *c* to picture *p*. Equivalent to (colorize *p* *c*).

(red *pict*) → pict?
  pict : pict?
(orange *pict*) → pict?
  pict : pict?
(yellow *pict*) → pict?
  pict : pict?
(green *pict*) → pict?
  pict : pict?
(blue *pict*) → pict?
  pict : pict?
(purple *pict*) → pict?
  pict : pict?
(black *pict*) → pict?
  pict : pict?
(brown *pict*) → pict?
  pict : pict?
(gray *pict*) → pict?
  pict : pict?
(white *pict*) → pict?
  pict : pict?

```
(cyan pict) → pict?
  pict : pict?
(magenta pict) → pict?
  pict : pict?
```

These functions apply appropriate colors to picture `p`.

```
(light color) → color/c
  color : color/c
(dark color) → color/c
  color : color/c
```

These functions produce ligher or darker versions of a color.

```
color/c : flat-contract?
```

This contract recognizes color strings, `color%` instances, and RGB color lists.

43.4.3 Pict Manipulation

```
(fill pict width height) → pict?
  pict : pict?
  width : (or/c real? #f)
  height : (or/c real? #f)
```

Extends *pict*'s bounding box to a minimum *width* and/or *height*, placing the original picture in the middle of the space.

Conditional Manipulations

These pict transformers all take boolean arguments that determine whether to transform the pict or leave it unchanged. These transformations can be useful for staged slides, as the resulting pict always has the same size and shape, and its contents always appear at the same position, but changing the boolean argument between slides can control when the transformation occurs.

```
(show pict [show?]) → pict?
  pict : pict?
  show? : truth/c = #t
```

```
(hide pict [hide?]) → pict?
  pict : pict?
  hide? : truth/c = #t
```

These functions conditionally show or hide an image, essentially choosing between *pict* and (ghost *pict*). The only difference between the two is the default behavior and the opposite meaning of the *show?* and *hide?* booleans. Both functions are provided for mnemonic purposes.

```
(strike pict [strike?]) → pict?
  pict : pict?
  strike? : truth/c = #t
```

Displays a strikethrough image by putting a line through the middle of *pict* if *strike?* is true; produces *pict* unchanged otherwise.

```
(shade pict [shade? #:ratio ratio]) → pict?
  pict : pict?
  shade? : truth/c = #t
  ratio : (real-in 0 1) = 1/2
```

Shades *pict* to show with *ratio* of its normal opacity; if *ratio* is 1 or *shade?* is #f, shows *pict* unchanged.

Conditional Combinations

These pict control flow operators decide which pict of several to use. All branches are evaluated; the resulting pict is a combination of the pict chosen by normal conditional flow with ghost applied to all the other picts. The result is a picture large enough to accomodate each alternative, but showing only the chosen one. This is useful for staged slides, as the pict chosen may change with each slide but its size and position will not.

```
(pict-if maybe-combine test-expr then-expr else-expr)

maybe-combine =
              | #:combine combine-expr
```

Chooses either *then-expr* or *else-expr* based on *test-expr*, similarly to if. Combines the chosen, visible image with the other, invisible image using *combine-expr*, defaulting to pict-combine.

```
(pict-cond maybe-combine [test-expr pict-expr] ...)
```

```
maybe-combine =
              | #:combine combine-expr
```

Chooses a *pict-expr* based on the first successful *test-expr*, similarly to `cond`. Combines the chosen, visible image with the other, invisible images using *combine-expr*, defaulting to `pict-combine`.

```
(pict-case test-expr maybe-combine [literals pict-expr] ...)

maybe-combine =
              | #:combine combine-expr
```

Chooses a *pict-expr* based on *test-expr* and each list of *literals*, similarly to `case`. Combines the chosen, visible image with the other, invisible images using *combine-expr*, defaulting to `pict-combine`.

```
(pict-match test-expr maybe-combine [pattern pict-expr] ...)

maybe-combine =
              | #:combine combine-expr
```

Chooses a *pict-expr* based on *test-expr* and each *pattern*, similarly to `match`. Combines the chosen, visible image with the other, invisible images using *combine-expr*, defaulting to `pict-combine`.

`pict-combine`

This syntax parameter determines the default pict combining form used by the above macros. It defaults to `lbl-superimpose`.

```
(with-pict-combine combine-id body ...)
```

Sets `pict-combine` to refer to *combine-id* within each of the *body* terms, which are spliced into the containing context.

43.4.4 Staged Slides

```
(staged [name ...] body ...)
```

Executes the *body* terms once for each stage *name*. The terms may include expressions and mutually recursive definitions. Within the body, each *name* is bound to a number from `1` to

the number of stages in order. Furthermore, during execution `stage` is bound to the number of the current stage and `stage-name` is bound to a symbol representing the *name* of the current stage. By comparing `stage` to the numeric value of each *name*, or `stage-name` to quoted symbols of the form `'name`, the user may compute based on the progression of the stages.

```
stage
stage-name
```

These keywords are bound during the execution of `staged` and should not be used otherwise.

```
(slide/staged [name ...] arg ...)
```

Creates a staged slide. Equivalent to `(staged [name ...] (slide arg ...))`.

Within a staged slide, the boolean arguments to `hide`, `show`, `strike`, and `shade` can be used to determine in which stages to perform a transformation. The macros `pict-if`, `pict-cond`, `pict-case`, and `pict-match` may also be used to create images which change naturally between stages.

43.4.5 Tables

```
(tabular  row
          ...
         [#:gap gap
          #:hgap hgap
          #:vgap vgap
          #:align align
          #:halign halign
          #:valign valign]) → pict?
  row : (listof (or/c string? pict?))
  gap : natural-number/c = gap-size
  hgap : natural-number/c = gap
  vgap : natural-number/c = gap
  align : (->* [] [] #:rest (listof pict?) pict?)
        = lbl-superimpose
  halign : (->* [] [] #:rest (listof pict?) pict?) = align
  valign : (->* [] [] #:rest (listof pict?) pict?) = align
```

Constructs a table containing the given *row*s, all of which must be of the same length. Applies `t` to each string in a *row* to construct a pict. The *hgap*, *vgap*, *halign*, and *valign* are used to determine the horizontal and vertical gaps and alignments as in `table` (except that every row and column is uniform).

43.4.6 Multiple Columns

```
(two-columns one two)
```

Constructs a two-column pict using *one* and *two* as the two columns. Sets current-para-width appropriately in each column.

```
(mini-slide pict ...) → pict?
  pict : pict?
```

Appends each *pict* vertically with space between them, similarly to the slide function.

```
(columns pict ...) → pict?
  pict : pict?
```

Combines each *pict* horizontally, aligned at the top, with space in between.

```
(column width body ...)
```

Sets current-para-width to *width* during execution of the *body* expressions.

```
(column-size n [r]) → real?
  n : exact-positive-integer?
  r : real? = (/ n)
```

Computes the width of one column out of *n* that takes up a ratio of *r* of the available space (according to current-para-width).

The subsequent bindings were added by Vincent St-Amour.

```
(ellipse/border w
                h
                #:color color
                #:border-color border-color
                #:border-width border-width) → pict?
  w : real?
  h : real?
  color : color/c
  border-color : color/c
  border-width : real?
```

```
(circle/border diameter
               #:color color
               #:border-color border-color
               #:border-width border-width) → pict?
  diameter : real?
  color : color/c
  border-color : color/c
  border-width : real?
(rectangle/border w
                  h
                  #:color color
                  #:border-color border-color
                  #:border-width border-width) → pict?
  w : real?
  h : real?
  color : color/c
  border-color : color/c
  border-width : real?
(rounded-rectangle/border w
                          h
                          #:color color
                          #:border-color border-color
                          #:border-width border-width) → pict?
  w : real?
  h : real?
  color : color/c
  border-color : color/c
  border-width : real?
```

These functions create shapes with border of the given color and width.

The subsequent bindings were added by Scott Owens.

```
(blank-line) → pict?
```

Adds a blank line of the current font size's height.

```
(pin-label-line label
                pict
                src-pict
                src-coord-fn
                dest-pict
                dest-coord-fn
                #:start-angle start-angle
                #:end-angle end-angle
                #:start-pull start-pull
                #:end-pull end-pull
                #:line-width line-width
                #:color color
                #:under? under?
                #:x-adjust x-adjust
                #:y-adjust y-adjust)      → pict?
  label : pict?
  pict : pict?
  src-pict : pict-path?
  src-coord-fn : (-> pict-path? (values real? real?))
  dest-pict : pict-path?
  dest-coord-fn : (-> pict-path? (values real? real?))
  start-angle : (or/c real? #f)
  end-angle : (or/c real? #f)
  start-pull : real?
  end-pull : real?
  line-width : (or/c real? #f)
  color : (or/c #f string? (is-a?/c color%))
  under? : any/c
  x-adjust : real?
  y-adjust : real?
```

```
(pin-arrow-label-line label
                      arrow-size
                      pict
                      src-pict
                      src-coord-fn
                      dest-pict
                      dest-coord-fn
                      #:start-angle start-angle
                      #:end-angle end-angle
                      #:start-pull start-pull
                      #:end-pull end-pull
                      #:line-width line-width
                      #:color color
                      #:under? under?
                      #:hide-arrowhead? hide-arrowhead?
                      #:x-adjust x-adjust
                      #:y-adjust y-adjust)
 → pict?
  label : pict?
  arrow-size : real?
  pict : pict?
  src-pict : pict-path?
  src-coord-fn : (-> pict-path? (values real? real?))
  dest-pict : pict-path?
  dest-coord-fn : (-> pict-path? (values real? real?))
  start-angle : (or/c real? #f)
  end-angle : (or/c real? #f)
  start-pull : real?
  end-pull : real?
  line-width : (or/c real? #f)
  color : (or/c #f string? (is-a?/c color%))
  under? : any/c
  hide-arrowhead? : any/c
  x-adjust : real?
  y-adjust : real?
```

```
(pin-arrows-label-line label
                       arrow-size
                       pict
                       src-pict
                       src-coord-fn
                       dest-pict
                       dest-coord-fn
                       #:start-angle start-angle
                       #:end-angle end-angle
                       #:start-pull start-pull
                       #:end-pull end-pull
                       #:line-width line-width
                       #:color color
                       #:under? under?
                       #:hide-arrowhead? hide-arrowhead?
                       #:x-adjust x-adjust
                       #:y-adjust y-adjust)
 → pict?
  label : pict?
  arrow-size : real?
  pict : pict?
  src-pict : pict-path?
  src-coord-fn : (-> pict-path? (values real? real?))
  dest-pict : pict-path?
  dest-coord-fn : (-> pict-path? (values real? real?))
  start-angle : (or/c real? #f)
  end-angle : (or/c real? #f)
  start-pull : real?
  end-pull : real?
  line-width : (or/c real? #f)
  color : (or/c #f string? (is-a?/c color%))
  under? : any/c
  hide-arrowhead? : any/c
  x-adjust : real?
  y-adjust : real?
```

These functions behave like `pin-line`, `pin-arrow-line` and `pin-arrows-line` with the addition of a label attached to the line.

43.5 GUI Widgets

```
(require unstable/gui/window)
```

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

mation.

43.5.1 Locked Text Fields

These classes and mixins provide text and combo field controls that cannot be directly edited by the user, but may be updated by other controls.

```
locked-text-field-mixin : (class? . -> . class?)
```
  argument extends/implements: `text-field%`

This mixin updates text field classes to prevent user edits, but allow programmatic update of the text value. It also sets the undo history length to a default of 0, as user undo commands are disabled and the history takes up space.

```
(new locked-text-field-mixin [[undo-history undo-history]])
 → (is-a?/c locked-text-field-mixin)
  undo-history : exact-nonnegative-integer? = 0
```

The mixin adds the *undo-history* initialization argument to control the length of the undo history. It defaults to 0 to save space, but may be set higher.

The mixin inherits all the initialization arguments of its parent class; it does not override any of them.

```
(send a-locked-text-field set-value str) → void?
  str : string?
```

Overrides `set-value` in `text-field%`.

Unlocks the text field's nested editor, calls the parent class's `set-value`, and then re-locks the editor.

```
locked-text-field% : class?
```
  superclass: `text-field%`

Equal to `(locked-text-field-mixin text-field%)`.

```
locked-combo-field% : class?
```
  superclass: `combo-field%`

Equal to `(locked-text-field-mixin combo-field%)`.

43.5.2 Union GUIs

`union-container-mixin` : `(class? . -> . class?)`
  argument extends/implements: `area-container<%>`

This mixin modifies a container class to display only one of its child areas at a time, but to leave room to switch to any of them.

`(send` *an-union-container* `choose` *child*`)` → `void?`
  child : `(is-a?/c subwindow<%>)`

  This method changes which of the container's children is displayed. The chosen child is shown and the previous choice is hidden.

`union-pane%` : `class?`
  superclass: `pane%`

Equal to `(union-container-mixin pane%)`.

`union-panel%` : `class?`
  superclass: `panel%`

Equal to `(union-container-mixin panel%)`.