Unstable: May Change Without Warning

Version 5.1.3

August 15, 2011

```
(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 Automata: Compiling State Machines

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

This package provides macros and functions for writing state machines over `racket/match` patterns (as opposed to concrete characters.)

2.1 Machines

```
(require unstable/automata/machine)
```

Each of the subsequent macros compile to instances of the machines provided by this module. This is a documented feature of the modules, so these functions should be used to, for example, determine if the machine is currently accepting.

```
(struct machine (next))
  next : (any/c . -> . machine?)
```

An applicable structure for machines. When the structure is applied, the `next` field is used as the procedure.

```
(struct machine-accepting machine (next))
  next : (any/c . -> . machine?)
```

A sub-structure of `machine` that is accepting.

```
(machine-accepts? m i) → boolean?
  m : machine?
  i : (listof any/c)
```

Returns `#t` if *m* ends in an accepting state after consuming every element of *i*.

```
(machine-accepts?/prefix-closed m i) → boolean?
  m : machine?
  i : (listof any/c)
```

Returns `#t` if *m* stays in an accepting state during the consumption of every element of *i*.

```
machine-null : machine?
```

A machine that is never accepting.

```
machine-epsilon : machine?
```

A machine that is initially accepting and never accepting afterwards.

```
machine-sigma* : machine?
```

A machine that is always accepting.

```
(machine-complement m) → machine?
  m : machine?
```

A machine that inverts the acception criteria of `m`.

```
(machine-star m) → machine?
  m : machine?
```

A machine that simulates the Kleene star of `m`. `m` may be invoked many times.

```
(machine-union m0 m1) → machine?
  m0 : machine?
  m1 : machine?
```

A machine that simulates the union of `m0` and `m1`.

```
(machine-intersect m0 m1) → machine?
  m0 : machine?
  m1 : machine?
```

A machine that simulates the intersection of `m0` and `m1`.

```
(machine-seq m0 m1) → machine?
  m0 : machine?
  m1 : machine?
```

A machine that simulates the sequencing of `m0` and `m1`. `m1` may be invoked many times.

```
(machine-seq* m0 make-m1) → machine?
  m0 : machine?
  make-m1 : (-> machine?)
```

A machine that simulates the sequencing of *m0* and (*make-m1*). (*make-m1*) may be invoked many times.

2.2 Deterministic Finite Automata

```
(require unstable/automata/dfa)
```

This module provides a macro for deterministic finite automata.

```
(dfa start
     (end ...)
     [state ([evt next-state]
             ...)]
     ...)
```

start : identifier?

end : identifier?

state : identifier?

next-state : identifier?

A machine that starts in state *start* where each state behaves as specified in the rules. If a *state* is in (*end* ...), then it is constructed with machine-accepting. *next-state* need not be a state from this DFA.

Examples:

```
(define M
  (dfa s1 (s1)
       [s1 ([0 s2]
            [(? even?) s1])]
       [s2 ([0 s1]
            [(? even?) s2])]))
> (machine-accepts? M (list 2 0 4 0 2))
#t
> (machine-accepts? M (list 0 4 0 2 0))
#f
> (machine-accepts? M (list 2 0 2 2 0 8))
#t
> (machine-accepts? M (list 0 2 0 0 10 0))
#t
> (machine-accepts? M (list))
#t
> (machine-accepts? M (list 4 0))
```

```
#f
```

2.3 Non-Deterministic Finite Automata

```
(require unstable/automata/nfa)
```

This module provides a macro for non-deterministic finite automata.

```
(nfa (start:id ...)
     (end:id ...)
     [state:id ([evt:expr (next-state:id ...)]
                ...)]
     ...)

  start : identifier?
  end : identifier?
  state : identifier?
  next-state : identifier?
```

A `machine` that starts in state `(set start ...)` where each state behaves as specified in the rules. If a state is in `(end ...)`, then the machine is accepting. `next-state` must be a state from this NFA.

These machines are efficiently compiled to use the smallest possible bit-string as a set representation and unsafe numeric operations where appropriate for inspection and adjusting the sets.

Examples:

```
(define M
  (nfa (s1 s3) (s1 s3)
       [s1 ([0 (s2)]
            [1 (s1)])]
       [s2 ([0 (s1)]
            [1 (s2)])]
       [s3 ([0 (s3)]
            [1 (s4)])]
       [s4 ([0 (s4)]
            [1 (s3)])]))
> (machine-accepts? M (list 1 0 1 0 1))
#t
> (machine-accepts? M (list 0 1 0 1 0))
#t
> (machine-accepts? M (list 1 0 1 1 0 1))
```

```
#t
> (machine-accepts? M (list 0 1 0 0 1 0))
#t
> (machine-accepts? M (list))
#t
> (machine-accepts? M (list 1 0))
#f
```

2.4 Non-Deterministic Finite Automata (with epsilon transitions)

```
(require unstable/automata/nfa-ep)
```

This module provides a macro for non-deterministic finite automata with epsilon transitions.

```
epsilon
```

A binding for use in epsilon transitions.

```
(nfa/ep (start:id ...)
        (end:id ...)
        [state:id ([epsilon (epsilon-state:id ...)]
                   ...
                   [evt:expr (next-state:id ...)]
                   ...)]
        ...)
```

```
start : identifier?
end : identifier?
state : identifier?
epsilon-state : identifier?
next-state : identifier?
```

Extends `nfa` with epsilon transitions, which must be listed first for each state.

Examples:

```
(define M
  (nfa/ep (s0) (s1 s3)
          [s0 ([epsilon (s1)]
               [epsilon (s3)])]
          [s1 ([0 (s2)]
               [1 (s1)])]
          [s2 ([0 (s1)]
```

```
                 [1 (s2)])]
            [s3 ([0 (s3)]
                 [1 (s4)])]
            [s4 ([0 (s4)]
                 [1 (s3)])]))
> (machine-accepts? M (list 1 0 1 0 1))
#t
> (machine-accepts? M (list 0 1 0 1 0))
#t
> (machine-accepts? M (list 1 0 1 1 0 1))
#t
> (machine-accepts? M (list 0 1 0 0 1 0))
#t
> (machine-accepts? M (list))
#t
> (machine-accepts? M (list 1 0))
#f
```

2.5 Regular Expressions

```
(require unstable/automata/re)
```

This module provides a macro for regular expression compilation.

```
(re re-pat)

re-pat = (rec id re-pat)
       | ,expr
       | (complement re-pat)
       | (seq re-pat ...)
       | (union re-pat ...)
       | (star re-pat)
       | epsilon
       | nullset
       | re-transformer
       | (re-transformer . datum)
       | (dseq pat re-pat)
       | pat
```

Compiles a regular expression over match patterns to a `machine`.

The interpretation of the pattern language is mostly intuitive. The pattern language may be extended with `define-re-transformer`. `dseq` allows bindings of the `match` pattern to be used in the rest of the regular expression. (Thus, they are not *really* regular expressions.)

`unquote` escapes to Racket to evaluate an expression that evaluates to a regular expression (this happens once, at compile time.) `rec` binds a Racket identifier to a delayed version of the inner expression; even if the expression is initially accepting, this delayed version is never accepting.

The compiler will use an NFA, provided `complement` and `dseq` are not used. Otherwise, many NFAs connected with the machine simulation functions from `unstable/automata/machine` are used.

```
complement
seq
union
star
epsilon
nullset
dseq
rec
```

Bindings for use in `re`.

```
(define-re-transformer id expr)
```

Binds *`id`* as an regular expression transformer used by the `re` macro. The expression should evaluate to a function that accepts a syntax object and returns a syntax object that uses the regular expression pattern language.

2.5.1 Extensions

```
(require unstable/automata/re-ext)
```

This module provides a few transformers that extend the syntax of regular expression patterns.

```
(opt re-pat)
```

Optionally matches *`re-pat`*.

```
(plus re-pat)
```

Matches one or more *`re-pat`* in sequence.

```
(rep re-pat num)
```

Matches *re-pat* in sequence *num* times, where *num* must be syntactically a number.

`(difference re-pat_0 re-pat_1)`

Matches everything that *re-pat_0* does, except what *re-pat_1* matches.

`(intersection re-pat_0 re-pat_1)`

Matches the intersection of *re-pat_0* and *re-pat_1*.

`(seq/close re-pat ...)`

Matches the prefix closure of the sequence `(seq re-pat ...)`.

2.5.2 Examples

Examples:

```
> (define-syntax-rule (test-re R (succ ...) (fail ...))
    (let ([r (re R)])
      (printf "Success: ~v => ~v\n" succ (machine-accepts? r succ))
      ...
      (printf "Failure: ~v => ~v\n" fail (machine-accepts? r fail))
      ...))
> (test-re epsilon
           [(list)]
           [(list 0)])
Success: '() => #t
Failure: '(0) => #f
> (test-re nullset
          []
          [(list) (list 1)])
Failure: '() => #f
Failure: '(1) => #f
> (test-re "A"
          [(list "A")]
          [(list)
           (list "B")])
Success: '("A") => #t
Failure: '() => #f
Failure: '("B") => #f
> (test-re (complement "A")
          [(list)
```

```
           (list "B")
           (list "A" "A")]
          [(list "A")])
Success: '() => #t
Success: '("B") => #t
Success: '("A" "A") => #t
Failure: '("A") => #f
> (test-re (union 0 1)
          [(list 1)
           (list 0)]
          [(list)
           (list 0 1)
           (list 0 1 1)])
Success: '(1) => #t
Success: '(0) => #t
Failure: '() => #f
Failure: '(0 1) => #f
Failure: '(0 1 1) => #f
> (test-re (seq 0 1)
          [(list 0 1)]
          [(list)
           (list 0)
           (list 0 1 1)])
Success: '(0 1) => #t
Failure: '() => #f
Failure: '(0) => #f
Failure: '(0 1 1) => #f
> (test-re (star 0)
          [(list)
           (list 0)
           (list 0 0)]
          [(list 1)])
Success: '() => #t
Success: '(0) => #t
Success: '(0 0) => #t
Failure: '(1) => #f
> (test-re (opt "A")
          [(list)
           (list "A")]
          [(list "B")])
Success: '() => #t
Success: '("A") => #t
Failure: '("B") => #f
> (define-re-transformer my-opt
    (syntax-rules ()
      [(_ pat)
```

```
      (union epsilon pat)]))
> (test-re (my-opt "A")
           [(list)
            (list "A")]
           [(list "B")])
Success: '() => #t
Success: '("A") => #t
Failure: '("B") => #f
> (test-re (plus "A")
           [(list "A")
            (list "A" "A")]
           [(list)])
Success: '("A") => #t
Success: '("A" "A") => #t
Failure: '() => #f
> (test-re (rep "A" 3)
           [(list "A" "A" "A")]
           [(list)
            (list "A")
            (list "A" "A")])
Success: '("A" "A" "A") => #t
Failure: '() => #f
Failure: '("A") => #f
Failure: '("A" "A") => #f
> (test-re (difference (? even?) 2)
           [(list 4)
            (list 6)]
           [(list 3)
            (list 2)])
Success: '(4) => #t
Success: '(6) => #t
Failure: '(3) => #f
Failure: '(2) => #f
> (test-re (intersection (? even?) 2)
           [(list 2)]
           [(list 1)
            (list 4)])
Success: '(2) => #t
Failure: '(1) => #f
Failure: '(4) => #f
> (test-re (complement (seq "A" (opt "B")))
           [(list "A" "B" "C")]
           [(list "A")
            (list "A" "B")])
Success: '("A" "B" "C") => #t
Failure: '("A") => #f
```

```
Failure: '("A" "B") => #f
> (test-re (seq epsilon 1)
           [(list 1)]
           [(list 0)
            (list)])
Success: '(1) => #t
Failure: '(0) => #f
Failure: '() => #f
> (test-re (seq 1 epsilon)
           [(list 1)]
           [(list 0)
            (list)])
Success: '(1) => #t
Failure: '(0) => #f
Failure: '() => #f
> (test-re (seq epsilon
                (union (seq (star 1) (star (seq 0 (star 1) 0 (star 1))))
                       (seq (star 0) (star (seq 1 (star 0) 1 (star 0)))))
                epsilon)
           [(list 1 0 1 0 1)
            (list 0 1 0 1 0)
            (list 1 0 1 1 0 1)
            (list 0 1 0 0 1 0)
            (list)]
           [(list 1 0)])
Success: '(1 0 1 0 1) => #t
Success: '(0 1 0 1 0) => #t
Success: '(1 0 1 1 0 1) => #t
Success: '(0 1 0 0 1 0) => #t
Success: '() => #t
Failure: '(1 0) => #f
> (test-re (star (complement 1))
           [(list 0 2 3 4)
            (list)
            (list 2)
            (list 234 5 9 1 9 0)
            (list 1 0)
            (list 0 1)]
           [(list 1)])
Success: '(0 2 3 4) => #t
Success: '() => #t
Success: '(2) => #t
Success: '(234 5 9 1 9 0) => #t
Success: '(1 0) => #t
Success: '(0 1) => #t
Failure: '(1) => #f
```

```
> (test-re (dseq x (? (curry equal? x)))
           [(list 0 0)
            (list 1 1)]
           [(list)
            (list 1)
            (list 1 0)])
Success: '(0 0) => #t
Success: '(1 1) => #t
Failure: '() => #f
Failure: '(1) => #f
Failure: '(1 0) => #f
```

3 Bytes

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

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

Compares two bytes case insensitively.

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

`read`s a value from `b` and returns it.

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

`write`s `v` to a bytes and returns it.

4 Contracts

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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 Asumu Takikawa.

```
(option/c contract) → contract?
  contract : contract?
```

Creates a contract that acts like *contract* but will also accept `#f`. Intended to describe situations where a failure or default value may be used.

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
predicates: self-contract violation, expected <boolean?>,
given: 'cat
  contract from (definition predicates), blaming
(definition predicates)
  contract:
```

```
      (sequence/c (-> any/c boolean?))
    at: eval:2.0
```

```
(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)
table: self-contract violation, expected <string?>, given: 2
  contract from (definition table), blaming (definition
table)
  contract: (dict/c symbol? string?)
        at: eval:4.0
> (dict-ref table 3)
table: contract violation, expected <symbol?>, given: 3
  contract from top-level, blaming (definition table)
  contract: (dict/c symbol? string?)
        at: eval:4.0
```

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 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: May Change Without Warning* 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 other arguments have the same meaning as for `expr/c`.

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)
the parameter argument of myparameterize1: self-contract
violation, expected <parameter?>, given: 'whoops
  contract from top-level, blaming top-level
  contract: parameter?
        at: eval:4.0
> (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)
the function argument of app: contract violation, expected
<number?>, given: 'apple
  contract from top-level, blaming (quote mod)
  contract: (-> number? number?)
        at: eval:8.0
> (app (lambda (x) 'pear) 5)
the function argument of app: self-contract violation,
expected <number?>, given: 'pear
  contract from top-level, blaming top-level
  contract: (-> number? number?)
        at: eval:9.0
```

6 Contracts for struct type properties

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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 that applies `value-contract` to the value associated with the property when used to create a new struct type (via `struct`, `make-struct-type`, etc).

The struct type property's accessor function is not affected; it must be protected separately.

Examples:

```
> (module propmod racket
    (require racket/contract
             unstable/prop-contract)
    (define-values (prop prop? prop-ref)
      (make-struct-type-property 'prop))
    (define (prop-app x v)
      (((prop-ref x) x) v))
    (provide/contract
     [prop? (-> any/c boolean?)]
     [prop (struct-type-property/c
            (-> prop? (-> number? boolean?)))]
     [prop-app (-> prop? number? boolean?)])
    (provide prop-ref))
> (module structmod racket
    (require 'propmod)
    (struct s (f) #:property prop (lambda (s) (s-f s)))
    (provide (struct-out s)))
> (require 'propmod 'structmod)
> (define s1 (s even?))
> (prop-app s1 5)
#f
> (prop-app s1 'apple)
prop-app: contract violation, expected <number?>, given:
'apple
  contract from propmod, blaming top-level
  contract: (-> prop? number? boolean?)
```

```
            at: eval:2.0
> (define s2 (s "not a fun"))
> (prop-app s2 5)
prop: contract violation, expected a procedure that accepts
1 mandatory argument without any keywords, given: "not a
fun"
   contract from propmod, blaming structmod
   contract:
      (struct-type-property/c
       (-> prop? (-> number? boolean?)))
   at: eval:2.0
> (define s3 (s list))
> (prop-app s3 5)
prop: contract violation, expected <boolean?>, given: '(5)
   contract from propmod, blaming structmod
   contract:
      (struct-type-property/c
       (-> prop? (-> number? boolean?)))
   at: eval:2.0
> ((prop-ref s3) 'apple)
prop: self-contract violation, expected <prop?>, given:
'apple
   contract from propmod, blaming propmod
   contract:
      (struct-type-property/c
       (-> prop? (-> number? boolean?)))
   at: eval:2.0
```

The first contract error above is a simple function contract violation on `prop-app`. The second and third contract errors above blame the `structmod` module, because it accepted the struct type property contract. To avoid blame, `structmod` should have placed a contract on `s`. The final contract error, involving `s3`, blames `propmod` because the struct type property contract obliges `propmod` to make sure the property's value is not misused, but `propmod` allows direct access to the property value via `prop-ref`. To avoid blame, `propmod` should remove the export of `prop-ref` or protect it with a contract.

7 Debugging

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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:6.0: (dprintf "level: ~a" 1)
  level: 1
  result: #<void>
<< eval:6.0: (dprintf "level: ~a" 1)
> (debug (debug (dprintf "level: ~a" 2)))
>> eval:7.0: (debug (dprintf "level: ~a" 2))
  >> eval:7.0: (dprintf "level: ~a" 2)
    level: 2
    result: #<void>
  << eval:7.0: (dprintf "level: ~a" 2)
  result: #<void>
<< eval:7.0: (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:8.0: debugf
  >> eval:8.0: +
    result: #<procedure:+>
  << eval:8.0: +
  >> eval:8.0: 1
    result: 1
  << eval:8.0: 1
  >> eval:8.0: 2
    result: 2
  << eval:8.0: 2
  >> eval:8.0: 3
    result: 3
  << eval:8.0: 3
  result: 6
<< eval:8.0: 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.

8 Definitions

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

Provides macros for creating and manipulating definitions.

8.1 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)
```

8.2 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)
```

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
```

8.3 Renaming Definitions

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

Establishes a rename transformer for each `new` identifier, redirecting it to the corresponding `old` identifier.

Examples:

```
> (define-renaming use #%app)
> (define-renamings [def define] [lam lambda])
> (def plus (lam (x y) (use + x y)))
> (use plus 1 2)
3
```

8.4 Forward Declarations

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

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.

8.5 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*)

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
```

8.6 Macro Definitions

(define-syntax-block (*macro-decl* ...) *body* ...)

```
macro-decl = macro-id
           | [macro-id expander-id]
```

Defines a syntax transformer for each *macro-id* based on the local definition of each *expander-id* (defaulting to *macro-id*/proc) in *body* .... Especially useful for mutually recursive expander functions and phase 1 macro definitions. Subsumes the behavior of `define-syntax-set`.

Examples:

```
> (define-syntax-block
     ([implies expand-implies]
      nand)
```

```
    (define-syntax-rule (==> pattern template)
      (syntax-rules () [pattern template]))

    (define expand-implies (==> (_ a b) (or (not a) b)))
    (define nand/proc (==> (_ a ...) (not (and a ...)))))
> (implies #t (printf "True!\n"))
True!
> (implies #f (printf "False!\n"))
#t
> (nand #t #t (printf "All True!\n"))
All True!
#f
> (nand #t #f (printf "Some False!\n"))
#t
> (define-syntax-block (undefined-macro)
    (define irrelevant "Whoops!"))
```
reference to undefined identifier: undefined-macro/proc

8.7 Effectful Transformation

```
(in-phase1 e)
```

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)
```

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

9 Dictionaries

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

This module provides tools for manipulating dictionary values.

```
(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
```

```
(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)))
```

10 Exceptions

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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.

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

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

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
```

11 Filesystem

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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.

12 Find

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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
```

13 Finding Mutated Variables

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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
```

}

14 Functions

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

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

14.1 Simple Functions

14.2 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?))
reference to undefined identifier: negate
> (f 1)
reference to undefined identifier: f
> (f 'one)
reference to undefined identifier: f
```

```
((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
```

14.3 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
```

14.4 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
```

14.5 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"
```

15 Generics

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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
```

16 Hash Tables

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

This module provides tools for manipulating hash tables.

```
(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)))
```

17 Interface-Oriented Programming for Classes

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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.

```
(define-interface-expander id transformer-expr)
```

Defines *id* as a macro that can be used within `define-interface` forms.

Examples:

```
> (define-interface-expander stack-methods
    (lambda (stx) #'[empty? push pop]))
> (define-interface stack<%> ()
    ((stack-methods)))
> (interface->method-names stack<%>)
'(empty? pop push)
```

18 Lists

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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
```

```
(take-common-prefix l r #:same? same?) → list?
  l : list?
  r : list?
  same? : equal?
```

Returns the longest common prefix of *l* and *r*.

Example:

```
> (take-common-prefix '(a b c d) '(a b x y z))
'(a b)
```

```
(drop-common-prefix l r #:same same?) → list? list?
  l : list?
  r : list?
  same? : equal?
```

Returns the tails of *l* and *r* with the common prefix removed.

Example:

```
> (drop-common-prefix '(a b c d) '(a b x y z))
'(c d)
'(x y z)
```

```
(split-common-prefix l r #:same? same?) → list? list? list?
  l : list?
```

```
r : list?
same? : equal?
```

Returns the longest common prefix together with the tails of `l` and `r` with the common prefix removed.

Example:

```
> (split-common-prefix '(a b c d) '(a b x y z))
'(a b)
'(c d)
'(x y z)
```

The subsequent bindings were added by Sam Tobin-Hochstadt.

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

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

Example:

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

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

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

Example:

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

The subsequent bindings were added by Ryan Culpepper.

```
(check-duplicate  lst
                 [#:key extract-key
                  #:same? same?])    → (or/c any/c #f)
  lst : list?
  extract-key : (-> any/c any/c) = (lambda (x) x)
  same? : (or/c (any/c any/c . -> . any/c) = equal?
                dict?)
```

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

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

Examples:

```
> (check-duplicate '(1 2 3 4))
#f
> (check-duplicate '(1 2 3 2 1))
2
> (check-duplicate '((a 1) (b 2) (a 3)) #:key car)
'(a 3)
> (define id-t (make-free-id-table))
> (check-duplicate (syntax->list #'(a b c d a b))
                   #:same? id-t)
#<syntax:13:0 a>
> (dict-map id-t list)
'((#<syntax:13:0 a> #t) (#<syntax:13:0 d> #t) (#<syntax:13:0 c> #t)
(#<syntax:13:0 b> #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)
```

```
(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)
```

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)
```

19 Logging

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

This module provides tools for logging.

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

```
(with-logging-to-port port
                      proc
                     [#:level level]) → any
  port : output-port?
  proc : (-> any)
  level : (or/c 'fatal 'error 'warning 'info 'debug) = 'info
```

Runs *proc*, outputting any logging of level *level* or higher to *port*. Returns whatever *proc* returns.

Example:

```
> (let ([my-log (open-output-string)])
    (with-logging-to-port my-log
      (lambda ()
        (log-warning "Warning World!")
        (+ 2 2))
      #:level 'warning)
    (get-output-string my-log))
"Warning World!\n"
```

```
(with-intercepted-logging interceptor
                          proc
                         [#:level level]) → any
  interceptor : (-> (vector/c
                      (or/c 'fatal 'error 'warning 'info 'debug)
                      string?
                      any/c)
                    any)
  proc : (-> any)
  level : (or/c 'fatal 'error 'warning 'info 'debug) = 'info
```

Runs *proc*, calling *interceptor* on any log message of level *level* or higher. *interceptor* receives the entire log vectors (see §14.5.3 "Receiving Logged Events") as arguments. Returns whatever *proc* returns.

Example:

```
> (let ([warning-counter 0])
    (with-intercepted-logging
      (lambda (l)
        (when (eq? (vector-ref l 0)
                   'warning)
          (set! warning-counter (add1 warning-counter))))
      (lambda ()
        (log-warning "Warning!")
        (log-warning "Warning again!")
        (+ 2 2))
      #:level 'warning)
    warning-counter)
2
```

20 Mark Parameters

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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.

21 Match

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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)
```

22 Net

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

22.1 URLs

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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.

23 Path

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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`.

24 Ports

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

This module provides tools for port I/O.

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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)
```

25 Pretty-Printing

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

This module provides tools for pretty-printing.

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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"
```

26 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)
'(true false symbol=? false? boolean=? thunk* thunk negate identity
curryr curry const)
```

27 Sequences

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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)
```

28 Strings

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

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

Returns `sb` as a lowercase symbol.

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

`read`s a value from `s` and returns it.

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

`write`s `v` to a string and returns it.

29 Structs

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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)
'()
```

30 Syntax

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

```
(explode-module-path-index mpi)
 → (listof (or/c module-path? resolved-module-path? #f))
  mpi : module-path-index?
```

Unfolds *mpi* using `module-path-index-split`, returning a list of the relative module paths together with the terminal resolved module path or `#f` for the "self" module.

Examples:

```
> (explode-module-path-index (car (identifier-binding #'lambda)))
'("kw.rkt" racket/private/pre-base #f)
> (explode-module-path-index (caddr (identifier-binding #'lambda)))
'(racket/base #f)
> (explode-module-path-index (car (identifier-binding #'define-
values)))
'('#%kernel #f)
```

```
(phase-of-enclosing-module)
```

Returns the phase level of the module in which the form occurs (and for the instantiation of the module in which the form is executed). For example, if a module is required directly by the "main" module (or the top level), its phase level is 0. If a module is required for-syntax by the "main" module (or the top level), its phase level is 1.

Examples:

```
> (module helper racket
    (require unstable/syntax)
    (displayln (phase-of-enclosing-module)))
> (require 'helper)
0
> (require (for-meta 1 'helper))
1
```

The subsequent bindings were added by Vincent St-Amour <stamourv@racket-lang.org>.

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

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

```
(syntax-within? a b) → boolean?
  a : syntax?
  b : syntax?
```

Returns true is syntax `a` is within syntax `b` in the source. Bounds are inclusive.

The subsequent bindings were added by Sam Tobin-Hochstadt <samth@racket-lang.org>.

```
(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:9:0 1> #<syntax:9:0 2> #<syntax:9:0 3>)
```

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

Examples:

```
(define loc
  (list (build-path "/tmp" "dir" "somewhere.rkt")
        #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.rkt>
(define stx2 (datum->syntax #f 'nowhere #f))
> (syntax-source-directory stx2)
#f
> (syntax-source-directory stx2)
#f
```

31 Temporal Contracts: Explicit Contract Monitors

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

The contract system implies the presence of a "monitoring system" that ensures that contracts are not violated. The `racket/contract` system compiles this monitoring system into checks on values that cross a contracted boundary. This module provides a facility to pass contract boundary crossing information to an explicit monitor for approval. This monitor may, for example, use state to enforce temporal constraints, such as a resource is locked before it is accessed.

31.1 Warning! Experimental!

This library is truly experimental and the interface is likely to drastically change as we get more experience making use of temporal contracts. In particular, the library comes with no advice about designing temporal contracts, which are much more subtle than standard contracts. This subtlety is compounded because, while temporal contract violations have accurate blame information, we cannot yet connect violations to sub-pieces of the temporal formula.

For example, applying `f` to `"three"` when it is contracted to only accept numbers will error by blaming the caller and providing the explanation "expected a <number?>, received: "three"". In contrast, applying `g` to `"even"` and then to `"odd"` when `g` is contracted to accept strings on every odd invocation, but numbers on every even invocation, will error by blaming the second (odd) call, but will not provide any explanation except "the monitor disallowed the call with arguments: "odd"". Translating non-acceptance of an event trace by an automata into a palatable user explanation is an open problem.

31.2 Monitors

```
(require unstable/temp-c/monitor)
```

```
(struct monitor (label)
        #:transparent)
  label : symbol?
(struct monitor:proj monitor (label proj-label v)
        #:transparent)
  label : symbol?
  proj-label : symbol?
  v : any/c
```

```
(struct monitor:call monitor (label
                              proj-label
                              f
                              app-label
                              kws
                              kw-args
                              args)
         #:transparent)
  label : symbol?
  proj-label : symbol?
  f : procedure?
  app-label : symbol?
  kws : (listof keyword?)
  kw-args : list?
  args : list?
(struct monitor:return monitor (label
                                proj-label
                                f
                                app-label
                                kws
                                kw-args
                                args
                                rets)
         #:transparent)
  label : symbol?
  proj-label : symbol?
  f : procedure?
  app-label : symbol?
  kws : (listof keyword?)
  kw-args : list?
  args : list?
  rets : list?
(monitor/c monitor-allows? label c) → contract?
  monitor-allows? : (-> monitor? boolean?)
  label : symbol?
  c : contract?
```

`monitor/c` creates a new contract around *`c`* that uses *`monitor-allows?`* to approve contract boundary crossings. (*`c`* approves positive crossings first.)

Whenever a value `v` is projected by the result of `monitor/c`, *`monitor-allows?`* must approve a `(monitor:proj label proj-label v)` structure, where `proj-label` is a unique symbol for this projection.

If *`monitor-allows?`* approves and the value is not a function, then the value is returned.

If the value is a function, then a projection is returned, whenever it is called, *monitor-allows?* must approve a (monitor:call *label* proj-label v app-label kws kw-args args) structure, where app-label is a unique symbol for this application and kws, kw-args, args are the arguments passed to the function.

Whenever it returns, *monitor-allows?* must approve a (monitor:return *label* proj-label v app-label kws kw-args args rets) structure, where ret are the return values of the application.

The unique projection label allows explicitly monitored contracts to be useful when used in a first-class way at different boundaries.

The unique application label allows explicitly monitored contracts to pair calls and returns when functions return multiple times or never through the use of continuations.

Here is a short example that uses an explicit monitor to ensure that *malloc* and *free* are used correctly.

```
(define allocated (make-weak-hasheq))
(define memmon
  (match-lambda
    [(monitor:return 'malloc _ _ _ _ _ _ (list addr))
     (hash-set! allocated addr #t)
     #t]
    [(monitor:call 'free _ _  _ _ _ (list addr))
     (hash-has-key? allocated addr)]
    [(monitor:return 'free _ _ _ _ _ (list addr) _)
     (hash-remove! allocated addr)
     #t]
    [_
     #t]))
(provide/contract
 [malloc (monitor/c memmon 'malloc (-> number?))]
 [free (monitor/c memmon 'free (-> number? void))])
```

31.3 Domain Specific Language

```
(require unstable/temp-c/dsl)
```

Constructing explicit monitors using only monitor/c can be a bit onerous. This module provides some helpful tools for making the definition easier. It provides everything from unstable/temp-c/monitor, as well as all bindings from unstable/automata/re and unstable/automata/re-ext. The latter provide a DSL for writing "dependent" regular expression machines over arbitrary racket/match patterns.

First, a few `match` patterns are available to avoid specify all the details of monitored events (since most of the time the detailed options are unnecessary.)

```
(call n a ...)
```

A `match` expander for call events to the labeled function *n* with arguments *a*.

```
(ret n a ...)
```

A `match` expander for return events to the labeled function *n* with return values *a*.

```
(with-monitor contract-expr re-pat)
```

Defines a monitored contract where the structural portion of the contract is the *contract-expr* (which may included embedded `label` expressions) and where the temporal portion of the contract is the regular expression given by *re-pat*. (Note: *re-pat* is not a Racket expression that evaluates to a regular expression. It is a literal regular expession.) An optional `#:concurrent` may be added between the contract and the regular expression to ensure that the machine is safe against race-conditions.

```
(label id contract-expr)
```

Labels a portion of a structural contract inside of `with-monitor` with the label *id*.

Here is a short example for *malloc* and *free*:

```
(with-monitor
    (cons/c (label 'malloc (-> addr?))
            (label 'free (-> addr? void?)))
  (complement
   (seq (star _)
        (dseq (call 'free addr)
              (seq
               (star (not (ret 'malloc (== addr))))
               (call 'free (== addr)))))))
```

32 GUI libraries

32.1 DrRacket Language Levels

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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])
 → (is-a?/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% : (and/c (implementation?/c drracket:language:language<%>)
                                (implementation?/c drracket:language:module-based-language<%>)
                                (implementation?/c drracket:language:simple-module-based-langu
```

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?)
 → (make-mixin-contract 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)
 → (make-mixin-contract 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 : (make-mixin-contract drracket:language:language<%>)
```

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

```
language-level-eval-as-module-mixin : (make-mixin-contract drracket:language:language<%>
                                                           drracket:language:module-based-lang
```

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 : (make-mixin-contract drracket:language:language<%>)
```

This mixin enables the macro stepper for its language level.

```
language-level-check-expect-mixin : (make-mixin-contract 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)
 → (make-mixin-contract 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`*.

32.2 Notify-boxes

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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.

32.3 Preferences

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* 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`*.

32.4 Slideshow Presentations

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

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

32.4.1 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*.

32.4.2 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.

32.4.3 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 accommodate 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.

32.4.4 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.

32.4.5 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 *rows*, 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).

32.4.6 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.

32.5 Progressive Picts and Slides

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

32.5.1 Progressive Picts

```
(require unstable/gui/ppict)
```

A *progressive pict* or "ppict" is a kind of `pict` that has an associated "pict placer," which generally represents a position and alignment. New picts can be placed on the progressive pict by calling `ppict-add`, and the placer can be updated by calling `ppict-go`. The `ppict-do` form provides a compact notation for sequences of those two operations.

```
(ppict-do base-expr ppict-do-fragment ...)
(ppict-do* base-expr ppic-do-fragment ...)

ppict-do-fragment = #:go placer-expr
                  | #:set pict-expr
                  | #:next
                  | #:alt (ppict-do-fragment ...)
                  | elem-expr

  base-expr : pict?
  placer-expr : placer?
  pict-expr : pict?
  elem-expr : (or/c pict? real? #f)
```

Builds a pict (and optionally a list of intermediate picts) progressively. The `ppict-do` form returns only the final pict; any uses of `#:next` are ignored. The `ppict-do*` form returns two values: the final pict and a list of all partial picts emitted due to `#:next` (the final pict is not included).

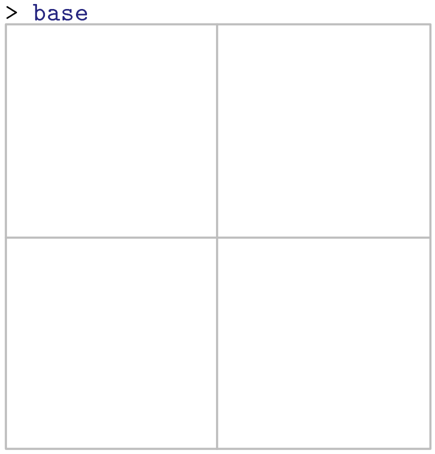
A `#:go` fragment changes the current placer. A `#:set` fragment replaces the current pict state altogether with a new computed pict. A `#:next` fragment saves a pict including only the contents emitted so far (but whose alignment takes into account picts yet to come). A `#:alt` fragment saves the current pict state, executes the sub-sequence that follows, saves the result (as if the sub-sequence ended with `#:next`), then restores the saved pict state before continuing.

The *`elem-expr`*s are interpreted by the current placer. A numeric *`elem-expr`* usually represents a spacing change, but some placers do not support them. A spacing change only affects added picts up until the next placer is installed; when a new placer is installed, the spacing is reset, usually to `0`.

The `ppict-do-state` form tracks the current state of the pict. It is updated before a `#:go` or `#:set` fragment or before a sequence of *`elem-expr`*s. It is not updated in the middle of a chain of *`elem-expr`*s, however.

Examples:

```
> (define base
    (ppict-do (colorize (rectangle 200 200) "gray")
              #:go (coord 1/2 1/2 'cc)
              (colorize (hline 200 1) "gray")
              #:go (coord 1/2 1/2 'cc)
              (colorize (vline 1 200) "gray")))
> base
```

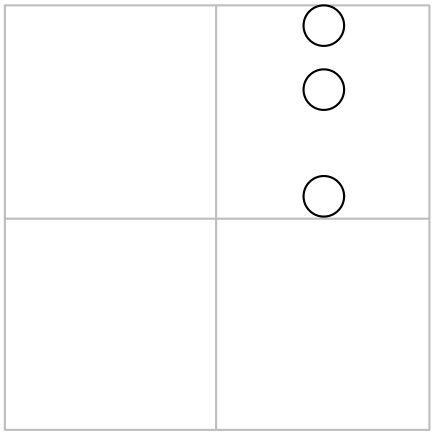


The use of `ppict-do` in the defnition of `base` above is equivalent to

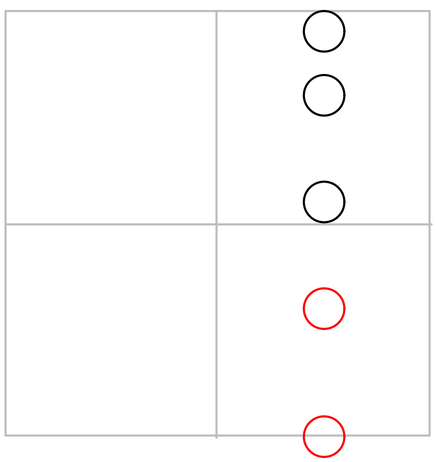
```
(let* ([pp (colorize (rectangle 200 200) "gray")]
       [pp (ppict-go pp (coord 1/2 1/2 'cc))]
       [pp (ppict-add pp (colorize (hline 200 1) "gray"))]
       [pp (ppict-go pp (coord 1/2 1/2 'cc))]
       [pp (ppict-add pp (colorize (vline 1 200) "gray"))])
  pp)
```

Examples:

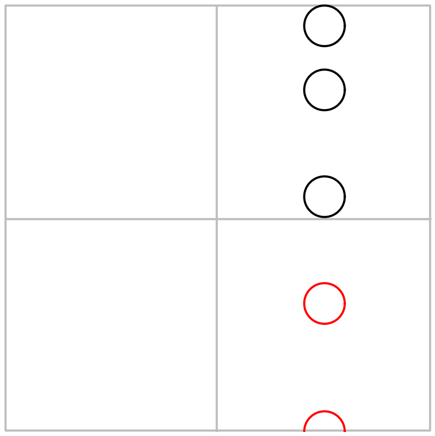
```
> (define circles-down-1
    (ppict-do base
              #:go (grid 2 2 2 1 'ct)
              10
              (circle 20)
              (circle 20)
              30
              (circle 20)))
> circles-down-1
```



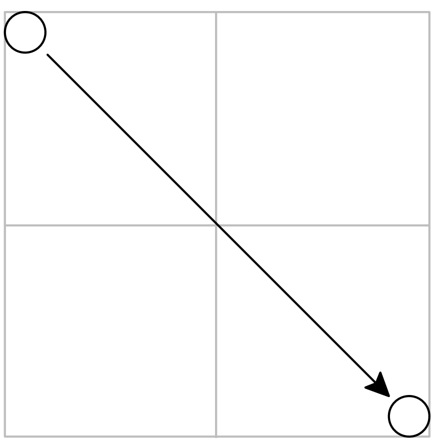
```
> (define circles-down-2
    (ppict-do circles-down-1
              (colorize (circle 20) "red")
              40
              (colorize (circle 20) "red")))
> (inset circles-down-2 20) ; draws outside its bounding box
```



```
> (inset (clip circles-down-2) 20)
```

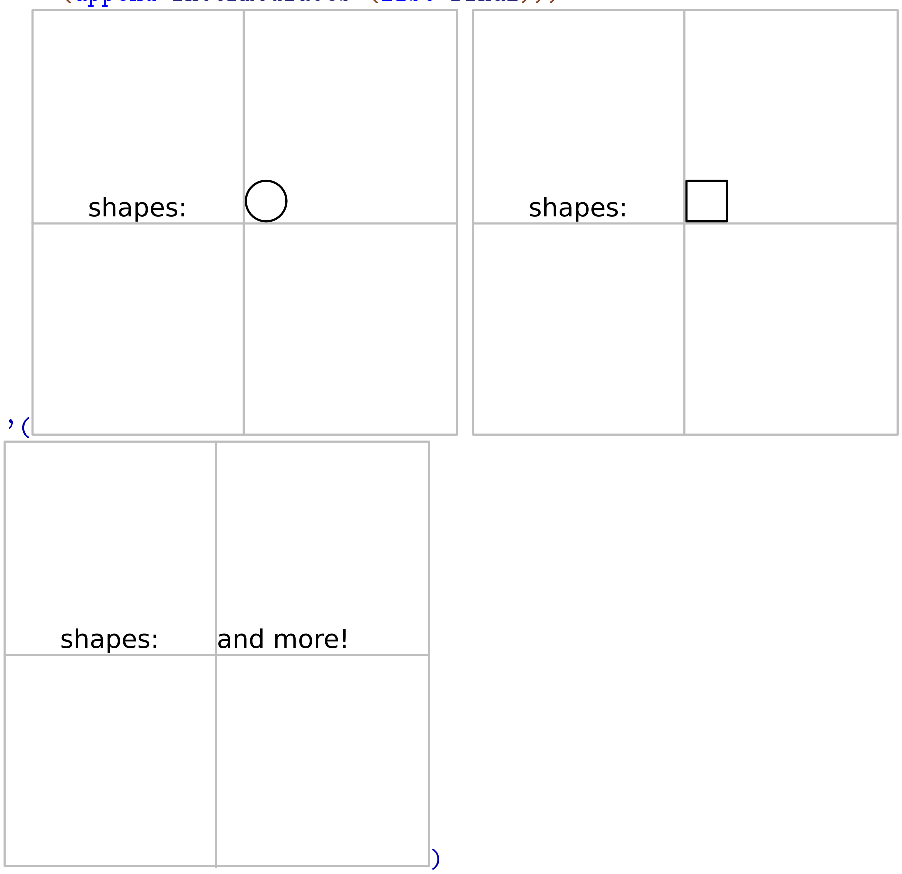


```
> (ppict-do base
            #:go (coord 0 0 'lt)
            (tag-pict (circle 20) 'circA)
            #:go (coord 1 1 'rb)
            (tag-pict (circle 20) 'circB)
            #:set (let ([p ppict-do-state])
                    (pin-arrow-line 10 p
                                    (find-tag p 'circA) rb-find
                                    (find-tag p 'circB) lt-find)))
```



```
> (let-values ([(final intermediates)
                (ppict-do* base
                           #:go (coord 1/4 1/2 'cb)
                           (text "shapes:")
                           #:go (coord 1/2 1/2 'lb)
                           #:alt [(circle 20)]
                           #:alt [(rectangle 20 20)]
```

```
                                        (text "and more!"))])
    (append intermediates (list final)))
```

'(


)

More examples of `ppict-do` are scattered throughout this section.

`ppict-do-state`

Tracks the current state of a `ppict-do` or `ppict-do*` form.

```
(ppict? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a progressive pict, `#f` otherwise.

```
(ppict-go p pl) → ppict?
  p : pict?
  pl : placer?
```

Creates a progressive pict with the given base pict *p* and the placer *pl*.

```
(ppict-add pp elem ...) → pict?
  pp : ppict?
  elem : (or/c pict? real? #f)
```

Creates a new pict by adding each *elem* pict on top of *pp* according to *pp*'s placer. The result pict may or may not be a progressive pict, depending on the placer used.

An *elem* that is a real number changes the spacing for subsequent additions. A *elem* that is `#f` is discarded; it is permitted as a convenience for conditionally including sub-picts. Note that `#f` is not equivalent to `(blank 0)`, since the latter will cause spacing to be added around it.

```
(placer? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a placer, `#f` otherwise.

```
(refpoint-placer? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a placer based on a reference point, `#f` otherwise.

```
(coord  rel-x
        rel-y
       [align
        #:abs-x abs-x
        #:abs-y abs-y
        #:compose composer]) → refpoint-placer?
  rel-x : real?
  rel-y : real?
  align : (or/c 'lt 'ct 'rt 'lc 'cc 'rc 'lb 'cb 'rb) = 'cc
  abs-x : real? = 0
  abs-y : real? = 0
  composer : procedure? = computed from align
```

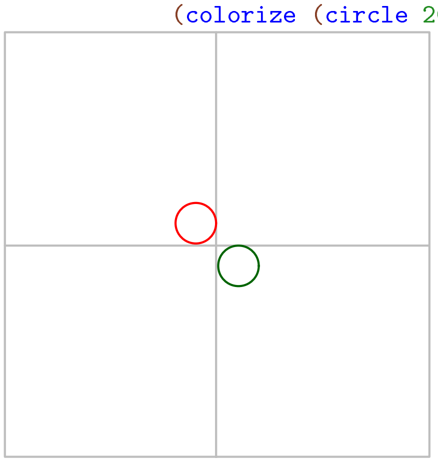
Returns a placer that places picts according to *rel-x* and *rel-y*, which are interpeted as fractions of the width and height of the base progressive pict. That is, `0`, `0` is the top left corner of the base's bounding box, and `1`, `1` is the bottom right. Then *abs-x* and *abs-y* offsets are added to get the final reference point.

Additions are aligned according to *align*, a symbol whose name consists of a horizontal alignment character followed by a vertical alignment character. For example, if *align* is 'lt, the pict is placed so that its left-top corner is at the reference point; if *align* is 'rc, the pict is placed so that the center of its bounding box's right edge coincides with the reference point.

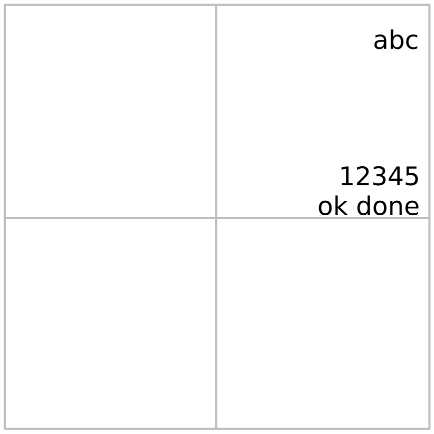
By default, if there are multiple picts to be placed, they are vertically appended, aligned according to the horizontal component of *align*. For example, if *align* is 'cc, the default *composer* is vc-append; for 'lt, the default *composer* is vl-append. The spacing is initially 0.

Examples:

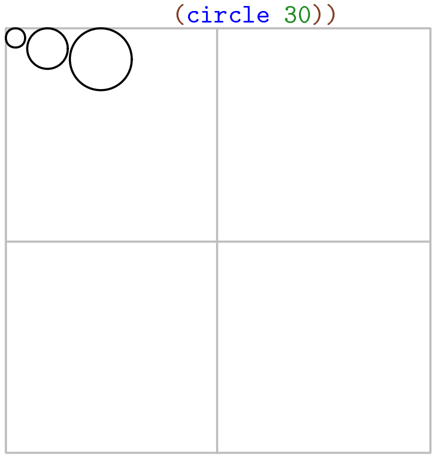
```
> (ppict-do base
            #:go (coord 1/2 1/2 'rb)
            (colorize (circle 20) "red")
            #:go (coord 1/2 1/2 'lt)
            (colorize (circle 20) "darkgreen"))
```



```
> (ppict-do base
            #:go (coord 1 0 'rt #:abs-x -5 #:abs-y 10)
            50 ; change spacing
            (text "abc")
            (text "12345")
            0  ; and again
            (text "ok done"))
```




```
> (ppict-do base
            #:go (coord 0 0 'lt #:compose ht-append)
            (circle 10)
            (circle 20)
            (circle 30))
```



```
(grid  cols
       rows
       col
       row
      [align
       #:abs-x abs-x
       #:abs-y abs-y
       #:compose composer]) → refpoint-placer?
  cols : exact-positive-integer?
  rows : exact-positive-integer?
  col : exact-integer?
  row : exact-integer?
```

```
align : (or/c 'lt 'ct 'rt 'lc 'cc 'rc 'lb 'cb 'rb) = 'cc
abs-x : real? = 0
abs-y : real? = 0
composer : procedure? = computed from align
```

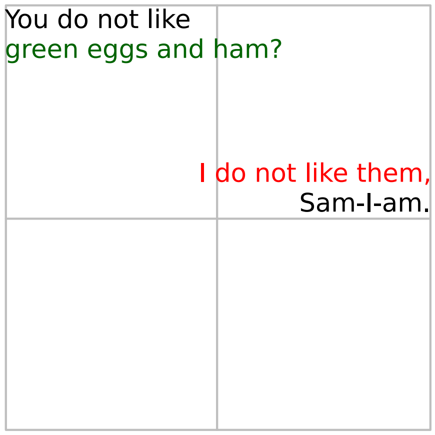
Returns a placer that places picts according to a position in a virtual grid. The *row* and *col* indexes are numbered starting at `1`.

Uses of `grid` can be translated into uses of `coord`, but the translation depends on the alignment. For example, `(grid 2 2 1 1 'lt)` is equivalent to `(coord 0 0 'lt)`, but `(grid 2 2 1 1 'rt)` is equivalent to `(coord 1/2 0 'rt)`.

Examples:

```
> (define none-for-me-thanks
    (ppict-do base
              #:go (grid 2 2 1 1 'lt)
              (text "You do not like")
              (colorize (text "green eggs and ham?") "darkgreen")))
> none-for-me-thanks
```


```
> (ppict-do none-for-me-thanks
            #:go (grid 2 2 2 1 'rb)
            (colorize (text "I do not like them,") "red")
            (text "Sam-I-am."))
```




```
(cascade [step-x step-y]) → placer?
  step-x : (or/c real? 'auto) = 'auto
  step-y : (or/c real? 'auto) = 'auto
```

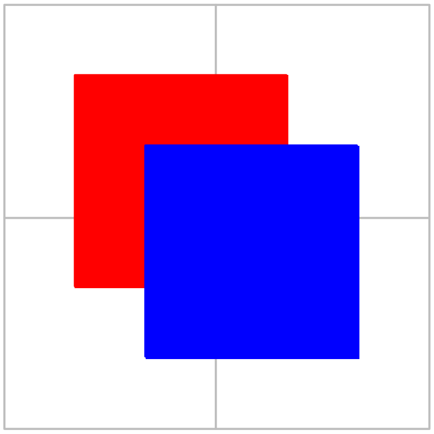
Returns a placer that places picts by evenly spreading them diagonally across the base pict in "cascade" style. This placer does not support changing the spacing by including a real number within the pict sequence.

When a list picts is to be placed, their bounding boxes are normalized to the maximum width and height of all picts in the list; each pict is centered in its new bounding box. The picts are then cascaded so there is `step-x` space between each of the picts' left edges; there is also `step-x` space between the base pict's left edge and the first pict's left edge. Similarly for `step-y` and the vertical spacing.

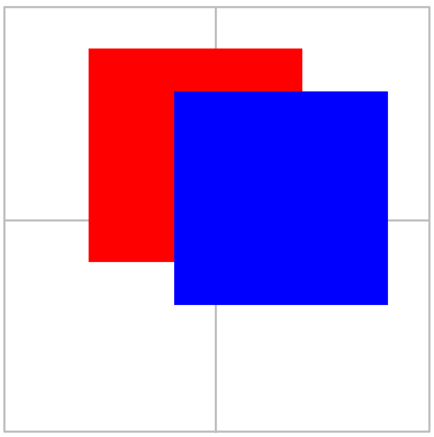
If `step-x` or `step-y` is `'auto`, the spacing between the centers of the picts to be placed is determined automatically so that the inter-pict spacing is the same as the spacing between the last pict and the base.

Examples:

```
> (ppict-do base
            #:go (cascade)
            (colorize (filled-rectangle 100 100) "red")
            (colorize (filled-rectangle 100 100) "blue"))
```



```
> (ppict-do base
            #:go (cascade 40 20)
            (colorize (filled-rectangle 100 100) "red")
            (colorize (filled-rectangle 100 100) "blue"))
```



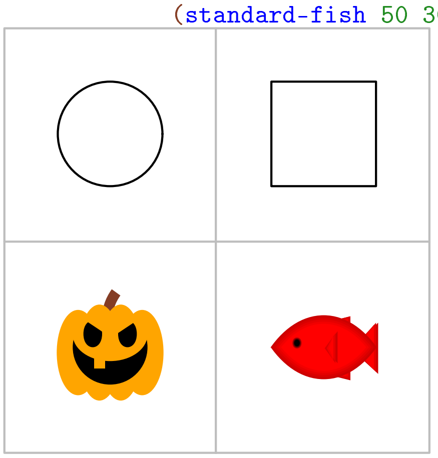
```
(tile cols rows) → placer?
  cols : exact-positive-integer?
  rows : exact-positive-integer?
```

Returns a placer that places picts by tiling them in a grid `cols` columns wide and `rows` rows high.

Example:

```
> (ppict-do base
            #:go (tile 2 2)
            (circle 50)
            (rectangle 50 50)
            (jack-o-lantern 50)
```

```
(standard-fish 50 30 #:color "red"))
```

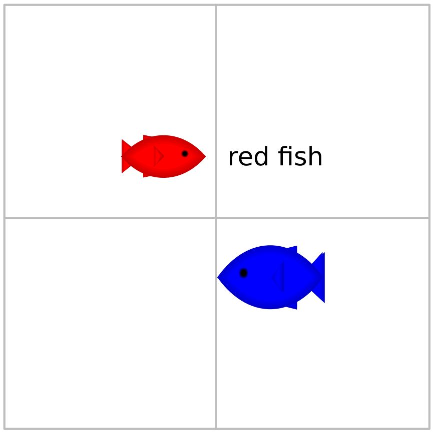


```
(at-find-pict  find-path
              [finder
               align
               #:abs-x abs-x
               #:abs-y abs-y
               #:compose composer]) → refpoint-placer?
  find-path : (or/c tag-path? pict-path?)
  finder : procedure? = cc-find
  align : (or/c 'lt 'ct 'rt 'lc 'cc 'rc 'lb 'cb 'rb) = 'cc
  abs-x : real? = 0
  abs-y : real? = 0
  composer : procedure? = computed from align
```

Returns a placer that places picts according to a reference point based on an existing pict within the base.

Example:

```
> (ppict-do base
            #:go (cascade)
            (tag-pict (standard-fish 40 20 #:direction 'right #:color "red") 'red-
fish)
            (tag-pict (standard-fish 50 30 #:direction 'left #:color "blue") 'blue-
fish)
            #:go (at-find-pict 'red-fish rc-find 'lc #:abs-x 10)
            (text "red fish"))
```

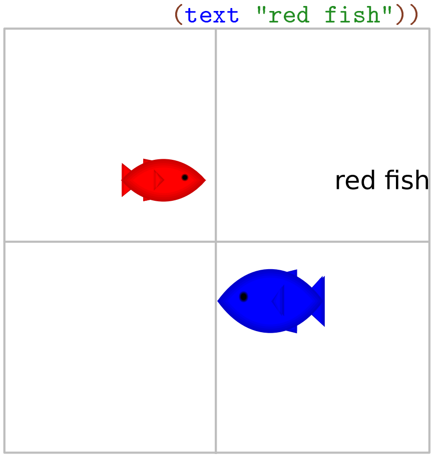



```
(merge-refpoints x-placer y-placer) → refpoint-placer?
  x-placer : refpoint-placer?
  y-placer : refpoint-placer?
```

Returns a placer like `x-placer` except that the y-coordinate of its reference point is computed by `y-placer`.

Example:

```
> (ppict-do base
            #:go (cascade)
            (tag-pict (standard-fish 40 20 #:direction 'right #:color "red") 'red-
fish)
            (tag-pict (standard-fish 50 30 #:direction 'left #:color "blue") 'blue-
fish)
            #:go (merge-refpoints (coord 1 0 'rc)
                                  (at-find-pict 'red-fish))
            (text "red fish"))
```



Tagging picts

```
(tag-pict p tag) → pict?
  p : pict?
  tag : symbol?
```

Returns a pict like *p* that carries a symbolic tag. The tag can be used with `find-tag` to locate the pict.

```
(find-tag p find) → (or/c pict-path? #f)
  p : pict?
  find : tag-path?
```

Locates a sub-pict of *p*. Returns a pict-path that can be used with functions like `lt-find`, etc.

```
(tag-path? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a symbol or a non-empty list of symbols, `#f` otherwise.

32.5.2 Progressive Slides

```
(require unstable/gui/pslide)
```

```
(pslide ppict-do-fragment ...)
```

Produce slide(s) using progressive picts. See `ppict-do` for an explanation of *ppict-do-fragment*s.

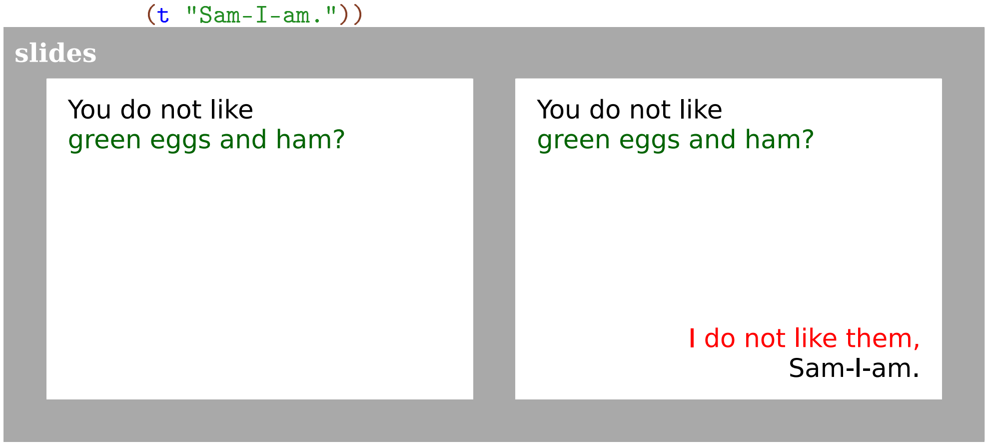
Note that like `slide` but unlike `ppict-do*`, the number of slides produced is one greater than the number of `#:next` uses; that is, a slide is created for the final pict.

Remember to include `gap-size` after updating the current placer if you want `slide`-like spacing.

Example:

```
> (pslide #:go (coord 0 0 'lt)
          (t "You do not like")
          (colorize (t "green eggs and ham?") "darkgreen")
          #:next
          #:go (coord 1 1 'rb)
```

```
(colorize (t "I do not like them,") "red")
(t "Sam-I-am."))
```



Note that the text is not flush against the sides of the slide, because `pslide` uses a base pict the size of the client area, excluding the margins.

```
(pslide-base-pict) → (-> pict)
(pslide-base-pict make-base-pict) → void?
  make-base-pict : (-> pict)
```

Controls the initial pict used by `pslide`. The default value is

```
(lambda () (blank client-w client-h))
```

```
(pslide-default-placer) → placer?
(pslide-default-placer placer) → void?
  placer : placer?
```

Controls the initial placer used by `pslide`. The default value is

```
(coord 1/2 1/2 'cc)
```