# Unstable: May Change Without Warning

Version 6.0.1

May 5, 2014

This manual documents some of the libraries available in the `unstable` collection. See also the unstable GUI libraries documentation.

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

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

```
(require unstable/automata)
```
package: base

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)
```
package: unstable-lib

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)
```
package: unstable-lib

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)
```
package: unstable-lib

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)
```
package: unstable-lib

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)
```
package: unstable-lib

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)
```
package: `unstable-lib`

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

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

```
(require unstable/bytes)
```
package: `unstable-list-lib`

```
(read/bytes b) → printable/c
  b : bytes?
```

reads a value from `b` and returns it.

```
(write/bytes v) → bytes?
  v : printable/c
```

writes `v` to a bytes and returns it.

# 4 Contracts

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

```
(require unstable/contract)
    package: unstable-contract-lib
```

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

Returns `#t` if *x* is a string and is not empty; returns `#f` otherwise.

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

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

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

Equivalent to `(between/c 0 65535)`.

```
path-piece? : 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.

```
(maybe/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>.

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

This contract recognizes Racket 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.

```
(sequence/c [#:min-count min-count]
            elem/c ...)            → contract?
  min-count : (or/c #f exact-nonnegative-integer?) = #f
  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?.

If *min-count* is a number, the stream is required to have at least that many elements in it.

Examples:

```
> (define/contract predicates
    (sequence/c (-> any/c boolean?))
    (in-list (list integer?
                   string->symbol)))

> (for ([P predicates])
    (printf "~s\n" (P "cat")))
#f
predicates: broke its contract
  promised: boolean?
  produced: 'cat
  in: the range of
      an element of
      (sequence/c (-> any/c boolean?))
  contract from: (definition predicates)
  blaming: (definition predicates)
  at: eval:2.0
> (define/contract numbers&strings
    (sequence/c number? string?)
    (in-dict (list (cons 1 "one")
                   (cons 2 "two")
                   (cons 3 'three))))

> (for ([(N S) numbers&strings])
    (printf "~s: ~a\n" N S))
1: one
2: two
numbers&strings: broke its contract
  promised: string?
  produced: 'three
  in: an element of
      (sequence/c number? string?)
  contract from: (definition numbers&strings)
  blaming: (definition numbers&strings)
  at: eval:4.0
> (define/contract a-sequence
    (sequence/c #:min-count 2 char?)
    "x")

> (for ([x a-sequence]
        [i (in-naturals)])
    (printf "~a is ~a\n" i x))
0 is x
a-sequence: broke its contract
  promised: a sequence that contains at least 2 values
  produced: "x"
```

*in: (sequence/c #:min-count 2 char?)*
*contract from: (definition a-sequence)*
*blaming: (definition a-sequence)*
*at: eval:6.0*

The subsequent bindings were added by Neil Toronto <neil.toronto@gmail.com>.

```
(treeof elem-contract) → contract?
  elem-contract : contract?
```

Identifies values that meet the contract *elem-contract*, lists of such values, lists of lists, and so on.

Examples:

```
> (define number-tree/c (treeof number?))

> (flat-contract? number-tree/c)
#t
> (define number-tree? (flat-contract-predicate number-tree/c))

> (number-tree? 4)
#t
> (number-tree? '(4 5))
#t
> (number-tree? '((4 5) 6))
#t
> (number-tree? '(4 . 5))
#f
```

# 5 Contracts for Macro Subexpressions

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

```
(require unstable/wrapc)
```
package: base

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

```
(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: broke its
contract
  promised: parameter?
  produced: 'whoops
  in: parameter?
  contract from: top-level
  blaming: top-level
  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
  in: the 1st argument of
      (-> number? number?)
  contract from: top-level
  blaming: (quote mod)
  at: eval:8.0
> (app (lambda (x) 'pear) 5)
the function argument of app: broke its contract
  promised: number?
  produced: 'pear
  in: the range of
      (-> number? number?)
  contract from: top-level
```

*blaming: top-level*
*at: eval:9.0*

# 6 Debugging

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

```
(require unstable/debug)
```
package: `unstable-debug-lib`

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`.

# 7 Definitions

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

`(require unstable/define)` package: `unstable-lib`

Provides macros for creating and manipulating definitions.

`(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)
f: undefined;
 cannot reference an identifier before its definition
  in module: 'Failure
> (module Success scheme
    (require unstable/define)
    (at-end (f 5))
    (define (f x) x))

> (require 'Success)
```

`(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).

# 8 Errors

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

```
(require unstable/error)
```
package: base

```
(error* name
        message
        field
        value ...
        ...
       [#:continued continued-message]) → any
  name : symbol?
  message : string?
  field :  (let ([option/c (or/c 'value 'multi 'maybe)])
             (or/c string? (cons/c string? (listof option/c))))
  value : any/c
  continued-message : (or/c string? (listof string?)) = null
```

Raises an exception with a message composed according to the Racket error message convention. The raised exception is an instance of `exn:fail`.

The composed error message includes details consisting of the alternating *field* and *value* arguments. By default, *value* is formatted as if by `display` unless it is `#f`, in which case the detail line is omitted. The following options affect the formatting of the detail line:

- `'multi` formats each element in the corresponding *value*, which must be a list, as a separate line; if `'maybe` is also provided, then the detail line is omitted if the list is empty
- `'value` formats the value using `error-value->string-handler`; the detail line is not omittable unless `'maybe` or `'multi` is also provided

Examples:

```
> (error* 'mcbean "too many stars upon thars"
          '("given" value) 'star-bellied-sneetch
          '("stars" value) 3)
mcbean: too many stars upon thars
  given: 'star-bellied-sneetch
  stars: 3
> (error* 'hal "unable to open pod bay doors"
          #:continued "this mission is too important to let you
jeopardize it"
          "threat" "David Bowman"
```

```
          "detection" "lip reading")
hal: unable to open pod bay doors;
 this mission is too important to let you jeopardize it
  threat: David Bowman
  detection: lip reading
> (error* 'car "missing car keys"
          '("searched" multi)
          (list "dresser" "desk" "kitchen table" "under sofa"
                "behind microwave" "in washing machine")
          "last seen"
          #f)
car: missing car keys
  searched:
    dresser
    desk
    kitchen table
    under sofa
    behind microwave
    in washing machine
```

```
(raise-syntax-error* message
                     expr
                     sub-expr
                     field
                     value ...
                     ...
                    [#:continued continued-message]) → any
  message : string?
  expr : (or/c syntax? #f)
  sub-expr : (or/c syntax? #f)
  field :  (let ([option/c (or/c 'value 'multi 'maybe)])
             (or/c string? (cons/c string? (listof option/c))))
  value : any/c
  continued-message : (or/c string? (listof string?)) = null
```

Like `raise-syntax-error` but with the formatting of `error*`. The raised exception is an instance of `exn:fail:syntax`. Like `raise-syntax-error`, the inclusion of *expr* and *sub-expr* in the details of the error message is controlled by the `error-print-source-location` paramter; if they included, they are included before the other details specified by *field* and *value*. Unlike `raise-syntax-error`, both *expr* and *sub-expr* are mandatory arguments.

```
(compose-error-message  name
                        message
                        field
                        value ...
                        ...
                       [#:continued continued-message]) → string?
  name : (or/c symbol? #f)
  message : string?
  field :  (let ([option/c (or/c 'value 'multi 'maybe)])
             (or/c string? (cons/c string? (listof option/c))))
  value : any/c
  continued-message : (or/c string? (listof string?)) = null
```

Like `error*`, but produces a string conforming to the Racket error message convention.

```
(compose-error-detail field options value) → string?
  field : string?
  options : (listof (or/c 'value 'multi 'maybe))
  value : any/c
```

Formats a single detail for an error message. The *options* behave as described in `error*`.

The resulting string begins with a newline unless it is empty, so it can be appended to the end of a base error message.

# 9 Futures

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

```
(require unstable/future)
```
package: `unstable-lib`

```
(for/async (for-clause ...) body ...+)
(for*/async (for-clause ...) body ...+)
```

Like `for` and `for*`, but each iteration of the *body* is executed in a separate `future`, and the futures may be `touch`ed in any order.

# 10 Functions

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

```
(require unstable/function)
```
package: `unstable-list-lib`

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

## 10.1 Higher Order Predicates

```
((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
```

# 11 Hash Tables

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

```
(require unstable/hash)
```
package: `unstable-list-lib`

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

> h
'#hash((2 . (two dos zwei deux)) (1 . (one uno ein une)))
```

# 12 Interface-Oriented Programming for Classes

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

```
(require unstable/class-iop)
```
package: unstable-list-lib

```
(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<%>)
'(pop push empty?)
```

# 13 Lazy Require

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

```
(require unstable/lazy-require)
```
package: `unstable-lib`

```
(begin-on-demand #:export (fun-id ...)
   body ...+)
```

Defines each *fun-id* as a function that, when called, dynamically loads and executes the *body* forms. The *body* forms must contain definitions for each *fun-id*, and the value of each *fun-id* must be a function.

A *body* form may be any module-level form except `provide`. In particular, `require` forms are allowed.

The *body* forms are placed within a submodule that extends the scope of the enclosing module (ie, `module*` with `#f` in the language position). Consequently, any references to sibling submodules must include a with `".."` module path element.

# 14 Lists

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

```
(require unstable/list)
```
package: unstable-list-lib

```
(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)
```

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

The subsequent bindings were added by Sam Tobin-Hochstadt.

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

Example:

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

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

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

Example:

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

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

The subsequent bindings were added by Ryan Culpepper.

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 b> #t)
  (#<syntax:13:0 c> #t)
  (#<syntax:13:0 d> #t)
  (#<syntax:13:0 a> #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)
```

The subsequent bindings were added by Vincent St-Amour.

```
(group-by extract-key lst [=?]) → (listof (listof A))
  extract-key : (-> A B)
  lst : (listof A)
  =? : (-> B B any/c) = equal?
```

Groups the given list into equivalence classes, with equivalence being determined by *=?*.

Example:

```
> (group-by (lambda (x) (modulo x 3)) '(1 2 1 2 54 2 5 43 7 2 643 1 2 0))
'((0 54) (2 2 5 2 2 2) (1 643 7 43 1 1))
```

```
(cartesian-product lst ...) → (listof (listof A))
  lst : (listof A)
```

Computes the n-ary cartesian product of the given lists.

Examples:

```
> (cartesian-product '(1 2 3) '(a b c))
'((1 a) (1 b) (1 c) (2 a) (2 b) (2 c) (3 a) (3 b) (3 c))
> (cartesian-product '(4 5 6) '(d e f) '(#t #f))
'((4 d #t)
  (4 d #f)
  (4 e #t)
  (4 e #f)
  (4 f #t)
  (4 f #f)
  (5 d #t)
  (5 d #f)
  (5 e #t)
  (5 e #f)
```

```
  (5 f #t)
  (5 f #f)
  (6 d #t)
  (6 d #f)
  (6 e #t)
  (6 e #f)
  (6 f #t)
  (6 f #f))
```

The subsequent bindings were added by Eric Dobson.

```
(list-update lst index updater) → list?
  lst : list?
  index : (and/c (>=/c 0) (</c (length lst)))
  updater : (-> any/c any/c)
```

Returns a list that is the same as *lst* except at the specified index. The element at the specified index is (*updater* (list-ref *lst* *index*)).

Example:

```
> (list-update '(zero one two) 1 symbol->string)
'(zero "one" two)
```

```
(list-set lst index value) → list?
  lst : list?
  index : (and/c (>=/c 0) (</c (length lst)))
  value : any/c
```

Returns a list that is the same as *lst* except at the specified index. The element at the specified index is *value*.

Example:

```
> (list-set '(zero one two) 2 "two")
'(zero one "two")
```

# 15 Logging

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

```
(require unstable/logging)
```
package: typed-racket-lib

This module provides tools for logging.

```
(with-logging-to-port port proc log-spec ...) → any
  port : output-port?
  proc : (-> any)
  log-spec : (or/c 'fatal 'error 'warning 'info 'debug symbol? #f)
```

Runs *proc*, outputting any logging that would be received by `(make-log-receiver (current-logger) log-spec ...)` 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))
      'warning)
    (get-output-string my-log))
"Warning World!\n"
```

```
(with-intercepted-logging interceptor
                          proc
                          log-spec ...) → any
               (-> (vector/c
                    (or/c 'fatal 'error 'warning 'info 'debug)
                    string?
  interceptor :     any/c
                    (or/c symbol? #f))
                   any)
  proc : (-> any)
  log-spec : (or/c 'fatal 'error 'warning 'info 'debug symbol? #f)
```

Runs *proc*, calling *interceptor* on any log message that would be received by `(make-log-receiver (current-logger) log-spec ...)`. *interceptor* receives the entire log vectors (see §15.5.3 “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))
      'warning)
    warning-counter)
2
```

A lower-level interface to logging is also available.

```
(start-recording log-spec ...) → listener?
  log-spec : (or/c 'fatal 'error 'warning 'info 'debug symbol? #f)
(stop-recording listener)
    (listof (vector/c (or/c 'fatal 'error 'warning 'info 'debug)
 →                    string?
                      any/c
                      (or/c symbol? #f)))
  listener : listener?
```

`start-recording` starts recording log messages matching the given *log-spec* (see `make-log-receiver` for how *log-spec* is interpreted). Messages will be recorded until stopped by passing the returned listener object to `stop-recording`. `stop-recording` will then return a list of the log messages that have been reported.

Examples:

```
(define l (start-recording 'warning))

> (log-warning "1")

> (log-warning "2")

> (stop-recording l)
'(#(warning "1" #<continuation-mark-set> #f)
  #(warning "2" #<continuation-mark-set> #f))
```

# 16 Macro Testing

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

```
(require unstable/macro-testing)
    package: unstable-macro-testing-lib
```

```
(phase1-eval ct-expr maybe-quote maybe-catch?)

 maybe-quote =
             | #:quote quote-id

maybe-catch? =
             | #:catch? catch?
```

Evaluates *ct-expr* at compile time and quotes the result using *quote-id*, which defaults to `quote`. Another suitable argument for *quote-id* is `quote-syntax`.

If *catch?* is `#t`, then if the evaluation of *ct-expr* raises a compile-time exception, it is caught and converted to a run-time exception.

Examples:

```
> (struct point (x y))

> (phase1-eval (extract-struct-info (syntax-local-value #'point)))
'(struct:point point point? (point-y point-x) (#f #f) #t)
> (phase1-eval (extract-struct-info (syntax-local-value #'point))
               #:quote quote-syntax)
#<syntax (struct:point point point? (p...>
```

```
(convert-compile-time-error expr)
```

Equivalent to `(#%expression expr)` except if expansion of *expr* causes a compile-time exception to be raised; in that case, the compile-time exception is converted to a run-time exception raised when the expression is evaluated.

Use `convert-compile-time-error` to write tests for compile-time error checking like syntax errors:

Examples:

```
> (check-exn #rx"missing an \"else\" expression"
             (lambda () (convert-compile-time-error (if 1 2))))
```

```
> (check-exn #rx"missing formals and body"
             (lambda () (convert-compile-time-error (lambda))))
--------------------
FAILURE
message:       "Wrong exception raised"
exn-message:"eval:6:0: lambda: bad syntax\n  in: (lambda)"
exn:           #(struct:exn:fail:syntax "eval:6:0: lambda: bad syntax\n  in:
(lambda)" #<continuation-mark-set> (#<syntax:6:0 (lambda)>))
name:          check-exn
location:    (eval 6 0 6 1)
expression:  (check-exn #rx"missing formals and body" (lambda () (convert-compile-
time-error (lambda))))
params:        (#rx"missing formals and body" #<procedure:temp10>)

Check failure
--------------------
```

Without the use of `convert-compile-time-error`, the checks above would not be executed because the test program would not compile.

```
(convert-syntax-error expr)
```

Like `convert-compile-time-error`, but only catches compile-time `exn:fail:syntax?` exceptions and sets `error-print-source-location` to `#f` around the expansion of *expr* to make the message easier to match exactly.

Example:

```
> (check-exn #rx"^lambda: bad syntax$"
             (lambda () (convert-syntax-error (lambda))))
```

# 17 Mark Parameters

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

```
(require unstable/markparam)
```
package: base

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.

# 18 Match

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

```
(require unstable/match)
```
package: typed-racket-lib

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)
```

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

The subsequent bindings were added by Asumu Takikawa <asumu@racket-lang.org>.

Similar to `match?`, but uses `match*` and accepts multiple *val-expr* and corresponding *pat* in each clause to match on.

Examples:

```
> (match*? (1 2 3)
    (a b c)
    (x #f z))
#t
```

```
> (match*? (1 2 3)
    (a (? odd?) c)
    (x y z))
#t
> (match*? (#f #f #f)
    (1 2 3)
    (4 5 6))
#f
```

```
(object maybe-class field-clause ...)

 maybe-class =
             | class-expr

field-clause = (field field-id maybe-pat)

   maybe-pat =
             | pat
```

A match expander that checks if the matched value is an object and contains the fields named by the *field-id*s. If *pat*s are provided, the value in each field is matched to its corresponding *pat*. If a *pat* is not provided, it defaults to the name of the field.

If *class-expr* is provided, the match expander will also check that the supplied object is an instance of the class that the given expression evaluates to.

Examples:

```
(define point%
  (class object%
    (super-new)
    (init-field x y)))

> (match (make-object point% 3 5)
    [(object point% (field x) (field y))
     (sqrt (+ (* x x) (* y y)))])
5.830951894845301
> (match (make-object point% 0 0)
    [(object (field x (? zero?))
             (field y (? zero?)))
     'origin])
'origin
> (match (make-object object%)
    [(object (field x) (field y))
     'ok]
    [_ 'fail])
'fail
```

# 19 Open `place` expressions

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

```
(require unstable/open-place)
```

package: `unstable-lib`

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

```
(open-place id body ...+)
```

Like `(place id body ...)`, but *body* `...` may have free lexical variables, which are automatically sent to the newly-created place. Note that these variables must have values accepted by `place-message-allowed?`, otherwise an `exn:fail:contract` exception is raised.

# 20 Options

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

```
(require unstable/options)
    package: unstable-options-lib
```

```
(option/c  c
           [#:with-contract with
            #:tester tester
            #:invariant invariant
            #:immutable immutable
            #:flat? flat?
            #:struct struct-id])   → contract?
  c : contract?
  with : boolean? = #f
  tester : (or/c (-> any boolean?) 'dont-care) = 'dont-care
  invariant : (or/c (-> any boolean?) 'dont-care) = 'dont-care
  immutable : (or/c #t #f 'dont-care) = 'dont-care
  flat? : boolean? = #f
  struct-id : (or/c identifier? 'none) = 'none
```

Returns a contract that recognizes vectors or hashes or instances of struct *struct-id*. The data structure must match *c* and pass the *tester*.

When an option/c contract is attached to a value, the value is checked against the *tester*, if *tester* is a predicate. After that, contract checking is disabled for the value, if *with* is #f. If *with* is #t contract checking for the value remains enabled for *c*.

If waive-option is applied to a value guarded by an option/c contract, then waive-option returns the value after removing the option/c guard. If exercise-option is applied to a value guarded by an option/c contract, then exercise-option returns the value with contract checking enabled for *c*. If the *invariant* argument is a predicate, then exercise-option returns the value with contract checking enabled for (invariant/c *c invariant* #:immutable *immutable* #:flat? *flat?* #:struct *struct-id*).

The arguments *flat?* and *immutable* should be provided only if *invariant* is a predicate. In any other case, the result is a contract error.

Examples:

```
> (module server0 racket
    (require unstable/options)
    (provide
     (contract-out
```

```
      [vec (option/c (vectorof number?))]))
    (define vec (vector 1 2 3 4)))

> (require 'server0)

> (vector-set! vec 1 'foo)

> (vector-ref vec 1)
'foo
> (module server1 racket
    (require unstable/options)
    (provide
     (contract-out
      [vec (option/c (vectorof number?) #:with-contract #t)]))
    (define vec (vector 1 2 3 4)))

> (require 'server1)

> (vector-set! vec 1 'foo)
vec: contract violation
  expected: number?
  given: 'foo
  in: an element of
      the option of
      (option/c
       (vectorof number?)
       #:with-contract
       #t)
  contract from: server1
  blaming: top-level
  at: eval:6.0
> (module server2 racket
    (require unstable/options)
    (provide
     (contract-out
      [vec (option/c (vectorof number?) #:tester sorted?)]))
    (define vec (vector 1 42 3 4))
    (define (sorted? vec)
      (for/and ([el vec]
                [cel (vector-drop vec 1)])
               (<= el cel))))

> (require 'server2)
vec: broke its contract

  in: option contract tester #<procedure:sorted?> of
```

```
      (option/c
        (vectorof number?)
        #:tester
        #<procedure:sorted?>)
  contract from: server2
  blaming: server2
  at: eval:9.0
```

```
(exercise-option x) → any/c
  x : any/c
```

Returns `x` with contract ckecking enabled if an `option/c` guards `x`. In any other case it returns `x`. The result of `exercise-option` loses the guard related to `option/c`, if it has one to begin with, and thus its contract checking status cannot change further.

Examples:

```
> (module server3 racket
    (require unstable/options)
    (provide (contract-out [foo (option/c (-> number? symbol?))]))
    (define foo (λ (x) x)))

> (require 'server3 unstable/options)

(define e-foo (exercise-option foo))

> (foo 42)
42
> (e-foo 'wrong)
foo: contract violation
  expected: number?
  given: 'wrong
  in: the 1st argument of
      the option of
      (option/c (-> number? symbol?))
  contract from: server3
  blaming: top-level
  at: eval:11.0
> ((exercise-option e-foo) 'wrong)
foo: contract violation
  expected: number?
  given: 'wrong
  in: the 1st argument of
      the option of
      (option/c (-> number? symbol?))
  contract from: server3
```

```
  blaming: top-level
  at: eval:11.0
```

```
transfer/c : contract?
```

A contract that accepts any value. If the value is guarded with an `option/c` contract, `transfer/c` modifies the blame information for the `option/c` contract by adding the providing module and its client to the positive and negative blame parties respectively. If the value is not a value guarded with an `option/c` contract, then `transfer/c` is equivalent to `any/c`.

Examples:

```
> (module server4 racket
    (require unstable/options)
    (provide (contract-out [foo (option/c (-> number? symbol?))]))
    (define foo (λ (x) x)))

> (module middleman racket
    (require unstable/options 'server4)
    (provide (contract-out [foo transfer/c])))

> (require 'middleman unstable/options)

(define e-foo (exercise-option foo))

> (e-foo 1)
foo: broke its contract
  promised: symbol?
  produced: 1
  in: the range of
      the option of
      (option/c (-> number? symbol?))
  contract from: server4
  blaming multiple parties:
  middleman
  server4
  at: eval:17.0
> (module server5 racket
    (require unstable/options)
    (provide (contract-out [boo transfer/c]))
    (define (boo x) x))

> (require 'server5)

> (boo 42)
42
```

```
(waive-option x) → any/c
  x : any/c
```

If an option/c guards x, then waive-option returns x without the option/c guard. In any other case it returns x. The result of waive-option loses the guard related to option/c, if it had one to begin with, and thus its contract checking status cannot change further.

Examples:

```
> (module server6 racket
    (require unstable/options)
    (provide (contract-out [bar (option/c (-> number? symbol?))]))
    (define bar (λ (x) x)))

> (require 'server6 unstable/options)

(define e-bar (waive-option bar))

> (e-bar 'wrong)
'wrong
> ((waive-option e-bar) 'wrong)
'wrong
```

```
(tweak-option x) → any/c
  x : any/c
```

If an option/c guards x and contract checking for x is enabled, then tweak-option returns x with contract checking for x disabled. If an option/c guards x and contract checking for x is disabled, then tweak-option returns x with contract checking for x enabled. In any other case it returns x. The result of tweak-option retains the guard related to option/c if it has one to begin with and thus its contract checking status can change further using tweak-option, exercise-option or waive-option.

Examples:

```
> (module server7 racket
    (require unstable/options)
    (provide (contract-out [bar (option/c (-> number? symbol?))]))
    (define bar (λ (x) x)))

> (require 'server7 unstable/options)

(define t-bar (tweak-option bar))
```

```
> (t-bar 'wrong)
bar: contract violation
  expected: number?
  given: 'wrong
  in: the 1st argument of
      the option of
      (option/c (-> number? symbol?))
  contract from: server7
  blaming: top-level
  at: eval:30.0
> ((tweak-option t-bar) 'wrong)
'wrong
> ((waive-option t-bar) 'wrong)
'wrong
> ((exercise-option t-bar) 'wrong)
bar: contract violation
  expected: number?
  given: 'wrong
  in: the 1st argument of
      the option of
      (option/c (-> number? symbol?))
  contract from: server7
  blaming: top-level
  at: eval:30.0
```

```
(has-option? v) → boolean?
  v : any/c
```

Returns `#t` if `v` has an option contract.

```
(has-option-with-contract? v) → boolean?
  v : any/c
```

Returns `#t` if `v` has an option contract with contract checking enabled.

```
(invariant/c  c
              invariant
             [#:immutable immutable
              #:flat? flat?
              #:struct struct-id])  → contract?
  c : contract?
  invariant : (-> any boolean?)
  immutable : (or/c #t #f 'dont-care) = 'dont-care
  flat? : boolean? = #f
  struct-id : (or/c identifier? 'none) = 'none
```

Returns a contract that recognizes vectors or hashes or instances of struct *struct-id*. The data structure must match *c* and satisfy the *invariant* argument.

If the *flat?* argument is #t, then the resulting contract is a flat contract, and the *c* arguments must also be flat contracts. Such flat contracts will be unsound if applied to a mutable data structure, as they will not check future operations on the vector.

If the *immutable* argument is #t and the *c* arguments are flat contracts, the result will be a flat contract. If the *c* arguments are chaperone contracts, then the result will be a chaperone contract.

Examples:

```
> (module server8 racket
    (require unstable/options)
    (provide
      change
      (contract-out
       [vec (invariant/c
             any/c
             sorted?)]))
   (define vec (vector 1 2 3 4 5))
   (define (change) (vector-set! vec 2 42))
   (define (sorted? vec)
     (for/and ([el vec]
               [cel (vector-drop vec 1)])
       (<= el cel))))

> (require 'server8)

> (vector-set! vec 2 42)
vec: contract violation
  expected vector that satisfies #<procedure:sorted?> given:
'#(1 2 42 4 5)
  in: (invariant/c any/c #<procedure:sorted?>)
  contract from: server8
  blaming: top-level
  at: eval:37.0
> (change)

> (vector-ref vec 2)
vec: broke its contract
  expected vector that satisfies #<procedure:sorted?> given:
'#(1 2 42 4 5)
  in: (invariant/c any/c #<procedure:sorted?>)
  contract from: server8
  blaming: server8
```

*at: eval:37.0*

# 21 Parameter Groups

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

```
(require unstable/parameter-group)
    package: unstable-parameter-group-lib
```

Parameter groups are parameter-like *views* that represent multiple parameters.

Examples:

```
> (require unstable/parameter-group)

> (define param1 (make-parameter 1))

> (define param2 (make-parameter 2))

> (define-parameter-group params (param1 param2))

> (params)
(params-value 1 2)
> (parameterize/group ([params  (params-value 10 20)])
    (list (param1) (param2)))
'(10 20)
> (params)
(params-value 1 2)
> (params (params-value 100 200))

> (list (param1) (param2))
'(100 200)
```

Use parameter groups to conveniently set multiple parameters. For example, the `plot` library uses parameter groups to save and restore appearance-controlling parameters when it must draw plots within a thunk.

```
(parameter-group? v) → boolean?
  v : any/c
```

Returns `#t` when *v* is a parameter group.

```
(define-parameter-group name (param-or-group-expr ...) options)

options =
        | #:struct struct-name

  param-or-group-expr : (or/c parameter? parameter-group?)
```

Defines a new parameter group.

If *struct-name* is not given, `define-parameter-group` defines a new struct `<name>-value` to hold the values of parameters.

If *struct-name* is given, it must have a constructor `(`*struct-name param-or-group-expr* `...)` that accepts as many arguments as there are parameters in the group, and a *struct-name* match expander that accepts as many patterns as there are parameters.

Examples:

```
> (struct two-params (p1 p2) #:transparent)

> (define-parameter-group params* (param1 param2) #:struct two-
params)

> (params*)
(two-params 100 200)
```

```
(parameterize/group ([param-or-group-expr  value-expr] ...)
  body-expr ...+)

  param-or-group-expr : (or/c parameter? parameter-group?)
```

Corresponds to `parameterize`, but can parameterize parameter groups as well as parameters.

```
(parameterize*/group ([param-or-group-expr  value-expr] ...)
  body-expr ...+)

  param-or-group-expr : (or/c parameter? parameter-group?)
```

Corresponds to `parameterize*`, but can parameterize parameter groups as well as parameters.

# 22 Pretty-Printing

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

```
(require unstable/pretty)
```
package: unstable-pretty-lib

This module provides tools for pretty-printing.

```
(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"
```

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

```
(break-lines s [columns]) → string?
  s : string?
  columns : exact-nonnegative-integer? = (pretty-print-columns)
```

Splits the string *s* into multiple lines, each of width at most *columns*, splitting only at whitespace boundaries.

Example:

```
> (display (break-lines "This string is more than 80 characters
long. It is 98 characters long, nothing more, nothing less."))
This string is more than 80 characters long. It is 98 characters
long,
nothing more, nothing less.
```

# 23 Re-Contracting Identifiers

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

```
(require unstable/recontract)
```
package: `unstable-lib`

```
(recontract-out id ...)
```

Provides each *id* with its existing contract, but changes the positive blame party of the contract to the enclosing module, instead of the module that originally attached the contract to *id*. Each *id* must be imported from a module that exports it via `contract-out` or `recontract-out`; otherwise a syntax error is raised.

Use `recontract-out` when you want to use the same contracts both between different parts of a library and between the library and its clients. The library should use `recontract-out` in the public interface modules so that clients do not see references to private implementation modules in contract errors.

Examples:

```
> (module private racket
    (define (f x) (if (positive? x) x 'wrong))
    (provide (contract-out [f (-> real? real?)])))

> (module public racket
    (require 'private unstable/recontract)
    (provide (recontract-out f)))

> (require 'public)

> (f 1)
1
> (f -2)
f: broke its contract
   promised: real?
   produced: 'wrong
   in: the range of
         (-> real? real?)
   contract from: public
   blaming: public
   at: eval:3.0
> (f 'apple)
f: contract violation
   expected: real?
   given: 'apple
```

*in: the 1st argument of*
*(-> real? real?)*
*contract from: public*
*blaming: top-level*
*at: eval:3.0*

# 24 Sandbox

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

```
(require unstable/sandbox)
```
package: racket-doc

```
(make-log-based-eval log-file mode) → (-> any/c any)
  log-file : path-string?
  mode : (or/c 'record 'replay)
```

Creates an evaluator (like `make-base-eval`) that uses a log file to either record or replay evaluations.

If *mode* is `'record`, the evaluator records every interaction to *log-file*, replacing *log-file* if it already exists. The result of each interaction must be serializable.

If *mode* is `'replay`, the evaluator uses the contents of *log-file* instead of actually performing evaluatings. For each interaction, it compares the term to evaluate against the next interaction recorded in *log-file*. If the term matches, the stored result is returned; if not, the evaluator raises an error indicating that it is out of sync with *log-file*.

Use `make-log-based-eval` to document libraries when the embedded examples rely on external features that may not be present or appropriately configured on all machines.

# 25 Sequences

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

```
(require unstable/sequence)
```
package: base

```
(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)
```

The subsequent bindings were added by David Vanderson.

```
(in-slice length seq) → sequence?
  length : exact-positive-integer?
  seq : sequence?
```

Returns a sequence where each element is a list with *length* elements from the given sequence.

Example:

```
> (for/list ([e (in-slice 3 (in-range 8))]) e)
'((0 1 2) (3 4 5) (6 7))
```

# 26 Strings

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

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

package: `unstable-lib`

The subsequent bindings were added by Vincent St-Amour.

```
(regexp-filter pattern lst)
 → (listof (or/c string? bytes? path? input-port?))
  pattern : (or/c string? bytes? regexp? byte-regexp?)
  lst : (listof (or/c string? bytes? path? input-port?))
```

Keeps only the elements of *lst* that match *pattern*.

# 27 Structs

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

```
(require unstable/struct)
```
package: base

```
(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)
'()
```

# 28 Struct Printing

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

```
(require unstable/custom-write)
```
package: base

```
(make-constructor-style-printer get-constructor
                                get-contents)
 → (-> any/c output-port? (or/c #t #f 0 1) void?)
  get-constructor : (-> any/c (or/c symbol? string?))
  get-contents : (-> any/c sequence?)
```

Produces a function suitable as a value for `prop:custom-write`. The function prints values in "constructor style." When the value is `print`ed as an expression, it is shown as an application of the constructor (as returned by *`get-constructor`*) to the contents (as returned by *`get-contents`*). When given to `write`, it is shown as an unreadable value with the constructor separated from the contents by a colon.

Examples:

```
> (struct point (x y)
    #:property prop:custom-write
    (make-constructor-style-printer
     (lambda (obj) 'point)
     (lambda (obj) (list (point-x obj) (point-y obj)))))

> (print (point 1 2))
(point 1 2)

> (write (point 1 2))
#<point: 1 2>
```

The function also cooperates with `pretty-print`:

Examples:

```
> (parameterize ((pretty-print-columns 10))
    (pretty-print (point 3000000 4000000)))
(point
 3000000
 4000000)

> (parameterize ((pretty-print-columns 10))
    (pretty-write (point 3000000 4000000)))
```

```
#<point:
 3000000
 4000000>
```

### prop:auto-custom-write : (struct-type-property/c 'constructor)

When attached to a struct type, automatically generates a printer using `make-constructor-style-printer` and attaches it to the struct type's `prop:custom-write` property. It also sets the `prop:custom-print-quotable` property to `'never`.

Examples:

```
> (struct point3 (x y z)
    #:property prop:auto-custom-write 'constructor)

> (print (point3 3 4 5))
(point3 3 4 5)

> (write (point3 3 4 5))
#<point3: 3 4 5>
```

# 29 Syntax

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

```
(require unstable/syntax)
```
package: base

```
(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" "pre-base.rkt" "private/base.rkt" racket/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
```

```
(make-variable-like-transformer reference-stx
                                [setter-stx])
```

```
→ set!-transformer?
 reference-stx : syntax?
 setter-stx : (or/c syntax? #f) = #f
```

Creates a transformer that replaces references to the macro identifier with *reference-stx*. Uses of the macro in operator position are interpreted as an application with *reference-stx* as the function and the arguments as given.

If the macro identifier is used as the target of a `set!` form, then the `set!` form expands into the application of *setter-stx* to the `set!` expression's right-hand side, if *setter-stx* is syntax; otherwise, the identifier is considered immutable and a syntax error is raised.

Examples:

```
> (define the-box (box add1))

> (define-syntax op
    (make-variable-like-transformer
     #'(unbox the-box)
     #'(lambda (v) (set-box! the-box v))))

> (op 5)
6
> (set! op 0)

> op
0
```

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 Eric Dobson <eric.n.dobson@gmail.com>.

```
(syntax-length stx) → exact-nonnegative-integer?
  stx : syntax?
```

Performs `(length (syntax->list stx))`.

Example:

```
> (syntax-length #'(d e f))
3
```

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

## 29.1 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
```

# 30 Temporal Contracts: Explicit Contract Monitors

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

```
(require unstable/temp-c)
```
package: base

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.

## 30.1 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.

## 30.2 Monitors

```
(require unstable/temp-c/monitor)
```
package: unstable-lib

```
(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))])
```

## 30.3 Domain Specific Language

`(require unstable/temp-c/dsl)` package: `unstable-lib`

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 expression.) 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)))))))
```

# 31 Unix Domain Sockets

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

```
(require unstable/socket)
```
package: base

```
unix-socket-available? : boolean?
```

A boolean value that indicates whether unix domain sockets are available and supported on the current platform. The supported platforms are Linux and Mac OS X; unix domain sockets are not supported on Windows and other Unix variants.

```
(unix-socket-connect socket-path) → input-port? output-port?
  socket-path : unix-socket-path?
```

Connects to the unix domain socket associated with *socket-path* and returns an input port and output port for communicating with the socket.

```
(unix-socket-path? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a valid unix domain socket path for the current system, according to the following cases:

- If *v* is a path (`path-string?`), then the current platform must be either Linux or Mac OS X, and the length of *v*'s corresponding absolute path must be less than or equal to the platform-specific length (108 bytes on Linux, 104 bytes on Mac OS X). Example: `"/tmp/mysocket"`.
- If *v* is a bytestring (`bytes?`), then the current platform must be Linux, *v* must start with a `0` (NUL) byte, and its length must be less than or equal to 108 bytes. Such a value refers to a socket in the Linux abstract socket namespace. Example: `#"\0mysocket"`.

Otherwise, returns `#f`.

# 32 2D Syntax

`#lang unstable/2d` package: `base`

The `unstable/2d` language installs `#2d` reader support in the readtable, and then chains to the reader of another language that is specified immediately after `unstable/2d`.

The `#2d` syntax extension adds the ability use a two-dimensional grid syntax. That is, you can draw an ASCII-art grid and then treat that as an expression. For example, here is a simple equality function that operates on pairs and numbers, written using a `#2d` conditional expression:

```
#lang unstable/2d racket
(require unstable/2d/cond)

(define (same? a b)
  #2dcond
  ╔═══════════════╦═══════════════════════╦═══════════════╗
  ║               ║       (pair? a)       ║ (number? a)   ║
  ╠═══════════════╬═══════════════════════╬═══════════════╣
  ║ (pair? b)     ║ (and (same? (car a)   ║     #f        ║
  ║               ║              (car b))  ║               ║
  ║               ║      (same? (cdr a)   ║               ║
  ║               ║              (cdr b))) ║               ║
  ╠═══════════════╬═══════════════════════╬═══════════════╣
  ║ (number? b)   ║          #f           ║   (= a b)     ║
  ╚═══════════════╩═══════════════════════╩═══════════════╝)
```

This notation works in two stages: reading, and parsing (just as in Racket in general). The reading stage converts anything that begins with `#2d` into a parenthesized expression (possibly signaling errors if the ═ and ║ and ╬ characters do not line up in the right places).

Since the first line contains `#2dcond`, the reader will produce a sequence whose first position is the identifier `2dcond`.

That macro will take over and then expand into ordinary conditional expressions, in this case figuring out whether or not the inputs are pairs or numbers and evaluating the code in the appropriate cell.

At the reader level, the syntax `#2d` notation checks the number of columns in the first row and uses that as a guide for where subsequent rows may appear. Once that first row is set, it serves as a guide to where the columns may appear in subsequent rows, although following columns may be merged.

This merging can simplify some uses of `#2d` expressions. For example, consider this expres-

sion that captures subtyping relationships between a few of the Typed Racket numeric types, this time using a `#2d` match expression:

```
#lang unstable/2d racket
(require unstable/2d/match)

(define (subtype? a b)
  #2dmatch
  ╔═══════════╦═══════════╦═══════╦═══════════╗
  ║   a  b    ║ 'Integer  ║ 'Real ║ 'Complex  ║
  ╠═══════════╬═══════════╩═══════╩═══════════╣
  ║ 'Integer  ║             #t                ║
  ╠═══════════╬═══════════╗                   ║
  ║ 'Real     ║           ║                   ║
  ╠═══════════╣           ╚═══════╗           ║
  ║ 'Complex  ║         #f        ║           ║
  ╚═══════════╩═══════════════════╩═══════════╝)
```

There are a number of cell walls missing here, but this is still a well-formed `#2d` expression. In this case, the `2dmatch` treats any of the situations that fall into the larger regions as the same.

In general, a `#2d` expression, when read, turns into an expression with at least two sub-pieces (not counting the initial name). The first is a sequence of numbers giving the widths of the top row of cells; the second is also a sequence of numbers, this time giving the heights of the leftmost column of cells. The remaining sequence describe the cells content. The first element of each is itself a sequence of coordinates, one for each of the cells that are connected together. The remaining elements are the subexpressions in the given cells.

For example, this:

```
#lang unstable/2d racket
'#2dex
╔═══════╦═══════╗
║   0   ║   1   ║
╠═══════╬═══════╣
║   2   ║   3   ║
╚═══════╩═══════╝
```

evaluates to

```
'(2dex (10 10)
       (2 2)
       (((0 0)) 0)
```

```
        (((0 1)) 2)
        (((1 0)) 1)
        (((1 1)) 3))
```

and this

```
#lang unstable/2d racket
'#2dex
╔═══════╦═══════╦═══════╗
║   0   ║  1 2  ║  3 4  ║
╠═══════╬═══════╬═══════╣
║   5   ║       6       ║
╚═══════╩═══════╩═══════╝
```

evaluates to

```
'(2dex (10 10 10)
       (2 2)
       (((0 0)) 0)
       (((0 1)) 5)
       (((1 0)) 1 2)
       (((1 1) (2 1)) 6)
       (((2 0)) 3 4))
```

In addition, the cells coordinates pairs have source locations of the first character that is inside the corresponding cell. (Currently the span is always 1, but that may change.)

## 32.1 Editing 2D

DrRacket provides a number of keybindings to help editing `#2d` expressions. See DrRacket's keyboard shortcuts.

## 32.2 2D Cond

```
(require unstable/2d/cond)
```
package: unstable-2d

```
(2dcond cond-content)
```

```
 cond-content = question-row
                body-row
                ⋮
              | question-row
                body-row
                ⋮
                else-row

 question-row = empty-cell question-cell ···
              | empty-cell question-cell ··· else-cell

     body-row = question-cell exprs-cell ···

     else-row = question-cell exprs-cell ··· else-cell

question-cell = ╔═══════════════╗
                ‖ question-expr ‖
                ╚═══════════════╝

   empty-cell = ╔═══╗
                ‖   ‖
                ╚═══╝

   exprs-cell = ╔═══════════════╗
                ‖ expr expr ... ‖
                ╚═══════════════╝

    else-cell = ╔═══════╗
                ‖ else  ‖
                ╚═══════╝
```

Evaluates the first row of question expressions until one of them returns a true value (signaling an error if none do), then evaluates the first column of question expressions until one of them returns a true value (signaling an error if none do), and then evaluates the cell in the middle where both point to, returning the result of the last expression in that cell.

## 32.3 2D Match

```
(require unstable/2d/match)
```

package: unstable-2d

```
(2dmatch match-content)
```

```
 match-content = match-first-row
                 match-row
                 ⋮

match-first-row = two-expr-cell match-pat-cell ···

     match-row = match-pat-cell exprs-cell ···

 two-expr-cell = ╔═══════════════════╗
                 ‖col-expr row-expr‖
                 ╚═══════════════════╝

match-pat-cell = ╔═══════╗
                 ‖  pat  ‖
                 ╚═══════╝

    exprs-cell = ╔═══════════════╗
                 ‖expr expr ...‖
                 ╚═══════════════╝
```

Matches `col-expr` against each of patterns in the first column of the table and matches `row-expr` against each of the patterns in the row row, and then evaluates the corresponding *exprs-cell*, returning the value of the last expression in that cell.

## 32.4 2D Tabular

```
(require unstable/2d/tabular)
```
package: `unstable-2d`

```
(2dmatch tabular-content)
```

```
tabular-content = tabular-row
                  ⋮
                | tabular-row
                  ⋮
                  style-cell

    tabular-row = tabular-cell ···

   tabular-cell = ╔═══════════════════╗
                  ‖ tabular-expr ... ‖
                  ╚═══════════════════╝

     style-cell = ╔═══════════════════╗
                  ‖ style-content ... ‖
                  ╚═══════════════════╝

  style-content = #:style style-expr
                | #:sep sep-expr
                | #:ignore-first-row

  style-expr : style?
  sep-expr : (or/c block? content? #f)
  tabular-expr : (or/c block? content?)
```

Constructs a `tabular` matching the given cells.

If a cell spans multiple columns, then the resulting `tabular` has `'cont` in the corresponding list element. No cells may span rows.

The `#:style` and `#:sep` arguments are just passed to `tabular`.

If the `#:ignore-first-row` keyword is provided, then the first row of the `2dtabular` expression is ignored. This can be used in case the first row of the rendered table should not have all of the columns (as `#2d` syntax requires that the first row contain a cell for each column that appears in the table).