The Typed Racket Reference

Version 6.0.1

Sam Tobin-Hochstadt <samth@racket-lang.org>,
Vincent St-Amour <stamourv@racket-lang.org>,
Eric Dobson <endobson@racket-lang.org>,
and Asumu Takikawa <asumu@racket-lang.org>

May 5, 2014

This manual describes the Typed Racket language, a sister language of Racket with a static type-checker. The types, special forms, and other tools provided by Typed Racket are documented here.

For a friendly introduction, see the companion manual *The Typed Racket Guide*.

```
#lang typed/racket/base
 #lang typed/racket
```

package: `typed-racket-lib`

1 Type Reference

`Any`

Any Racket value. All other types are subtypes of `Any`.

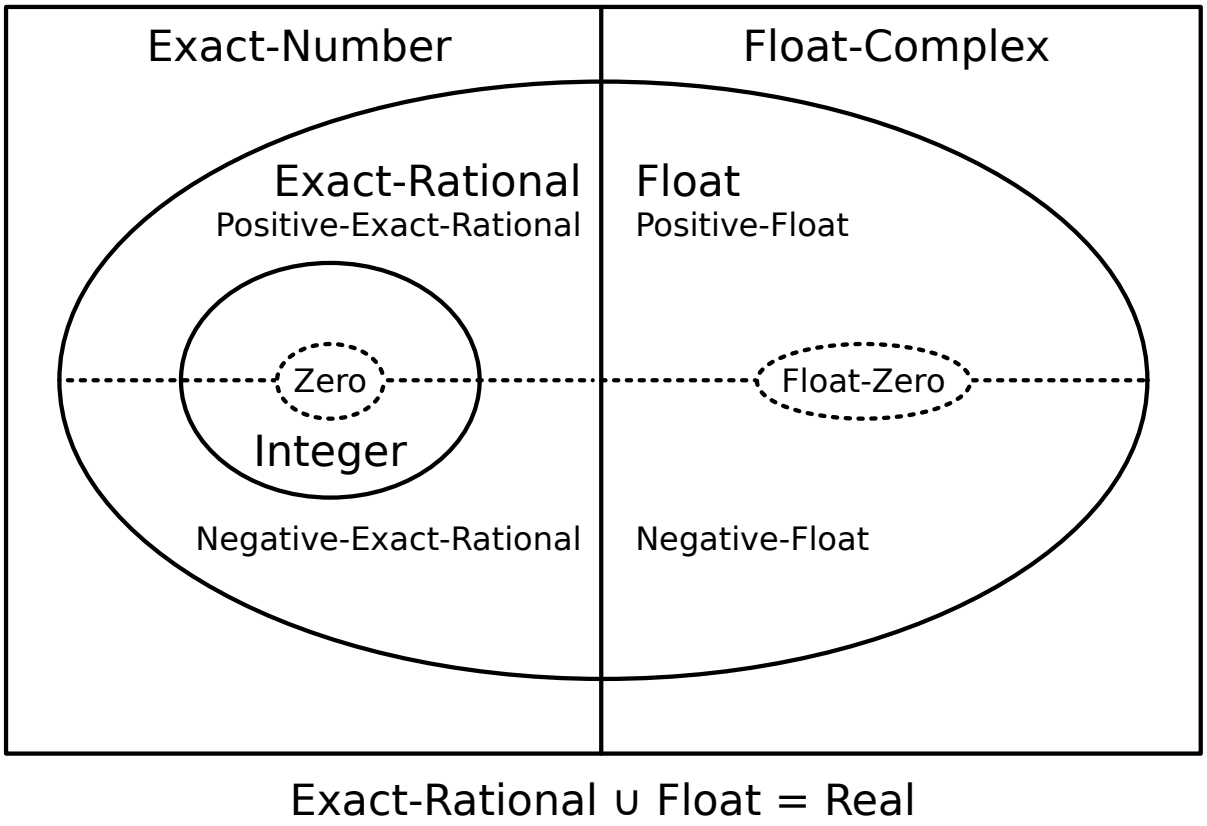
`Nothing`

The empty type. No values inhabit this type, and any expression of this type will not evaluate to a value.

1.1 Base Types

1.1.1 Numeric Types

These types represent the hierarchy of numbers of Racket. The diagram below shows the relationships between the types in the hierarchy.



The regions with a solid border are *layers* of the numeric hierarchy corresponding to sets of numbers such as integers or rationals. Layers contained within another are subtypes of the layer containing them. For example, `Exact-Rational` is a subtype of `Exact-Number`.

The `Real` layer is also divided into positive and negative types (shown with a dotted line). The `Integer` layer is subdivided into several fixed-width integers types, detailed later in this section.

```
Number
Complex
```

`Number` and `Complex` are synonyms. This is the most general numeric type, including all Racket numbers, both exact and inexact, including complex numbers.

```
Integer
```

Includes Racket's exact integers and corresponds to the `exact-integer?` predicate. This is the most general type that is still valid for indexing and other operations that require integral

values.

```
Float
Flonum
```

Includes Racket's double-precision (default) floating-point numbers and corresponds to the `flonum?` predicate. This type excludes single-precision floating-point numbers.

```
Single-Flonum
```

Includes Racket's single-precision floating-point numbers and corresponds to the `single-flonum?` predicate. This type excludes double-precision floating-point numbers.

```
Inexact-Real
```

Includes all of Racket's floating-point numbers, both single- and double-precision.

```
Exact-Rational
```

Includes Racket's exact rationals, which include fractions and exact integers.

```
Real
```

Includes all of Racket's real numbers, which include both exact rationals and all floating-point numbers. This is the most general type for which comparisons (e.g. `<`) are defined.

```
Exact-Number
Float-Complex
Single-Flonum-Complex
Inexact-Complex
```

These types correspond to Racket's complex numbers.

The above types can be subdivided into more precise types if you want to enforce tighter constraints. Typed Racket provides types for the positive, negative, non-negative and non-positive subsets of the above types (where applicable).

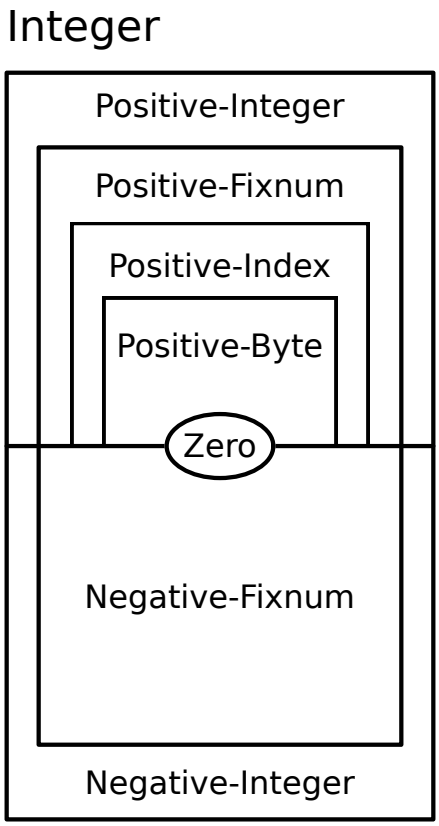
```
Positive-Integer
Exact-Positive-Integer
Nonnegative-Integer
Exact-Nonnegative-Integer
Natural
Negative-Integer
Nonpositive-Integer
```

```
Zero
Positive-Float
Positive-Flonum
Nonnegative-Float
Nonnegative-Flonum
Negative-Float
Negative-Flonum
Nonpositive-Float
Nonpositive-Flonum
Float-Negative-Zero
Flonum-Negative-Zero
Float-Positive-Zero
Flonum-Positive-Zero
Float-Zero
Flonum-Zero
Float-Nan
Flonum-Nan
Positive-Single-Flonum
Nonnegative-Single-Flonum
Negative-Single-Flonum
Nonpositive-Single-Flonum
Single-Flonum-Negative-Zero
Single-Flonum-Positive-Zero
Single-Flonum-Zero
Single-Flonum-Nan
Positive-Inexact-Real
Nonnegative-Inexact-Real
Negative-Inexact-Real
Nonpositive-Inexact-Real
Inexact-Real-Negative-Zero
Inexact-Real-Positive-Zero
Inexact-Real-Zero
Inexact-Real-Nan
Positive-Exact-Rational
Nonnegative-Exact-Rational
Negative-Exact-Rational
Nonpositive-Exact-Rational
Positive-Real
Nonnegative-Real
Negative-Real
Nonpositive-Real
Real-Zero
```

`Natural` and `Exact-Nonnegative-Integer` are synonyms. So are the integer and exact-integer types, and the float and flonum types. `Zero` includes only the integer `0`. `Real-Zero` includes exact `0` and all the floating-point zeroes.

These types are useful when enforcing that values have a specific sign. However, programs using them may require additional dynamic checks when the type-checker cannot guarantee that the sign constraints will be respected.

In addition to being divided by sign, integers are further subdivided into range-bounded types. The relationships between most of the range-bounded types are shown in this diagram:



Like the previous diagram, types nested inside of another in the diagram are subtypes of its containing types.

```
One
Byte
Positive-Byte
Index
Positive-Index
```

```
Fixnum
Positive-Fixnum
Nonnegative-Fixnum
Negative-Fixnum
Nonpositive-Fixnum
```

`One` includes only the integer `1`. `Byte` includes numbers from `0` to `255`. `Index` is bounded by `0` and by the length of the longest possible Racket vector. `Fixnum` includes all numbers represented by Racket as machine integers. For the latter two families, the sets of values included in the types are architecture-dependent, but typechecking is architecture-independent.

These types are useful to enforce bounds on numeric values, but given the limited amount of closure properties these types offer, dynamic checks may be needed to check the desired bounds at runtime.

Examples:

```
> 7
- : Integer [more precisely: Positive-Byte]
7
> 8.3
- : Flonum [more precisely: Positive-Flonum]
8.3
> (/ 8 3)
- : Exact-Rational [more precisely: Positive-Exact-Rational]
8/3
> 0
- : Integer [more precisely: Zero]
0
> -12
- : Integer [more precisely: Negative-Fixnum]
-12
> 3+4i
- : Exact-Number
3+4i
```

1.1.2 Other Base Types

```
Boolean
True
False
String
Keyword
Symbol
```

```
Char
Void
Input-Port
Output-Port
Port
Path
Path-For-Some-System
Regexp
PRegexp
Byte-Regexp
Byte-PRegexp
Bytes
Namespace
Namespace-Anchor
Variable-Reference
Null
EOF
Continuation-Mark-Set
Undefined
Module-Path
Module-Path-Index
Resolved-Module-Path
Compiled-Module-Expression
Compiled-Expression
Internal-Definition-Context
Pretty-Print-Style-Table
Special-Comment
Struct-Type-Property
Impersonator-Property
Read-Table
Bytes-Converter
Parameterization
Custodian
Inspector
Security-Guard
UDP-Socket
TCP-Listener
Logger
Log-Receiver
Log-Level
Thread
Thread-Group
Subprocess
Place
Place-Channel
```

```
Semaphore
Will-Executor
Pseudo-Random-Generator
```

These types represent primitive Racket data.

Examples:

```
> #t
- : Boolean [more precisely: True]
#t
> #f
- : False
#f
> "hello"
- : String
"hello"
> (current-input-port)
- : Input-Port
#<input-port:string>
> (current-output-port)
- : Output-Port
#<output-port:string>
> (string->path "/")
- : Path
#<path:/>
> #rx"a*b*"
- : Regexp
#rx"a*b*"
> #px"a*b*"
- : PRegexp
#px"a*b*"
> '#"bytes"
- : Bytes
#"bytes"
> (current-namespace)
- : Namespace
#<namespace:0>
> #\b
- : Char
#\b
> (thread (lambda () (add1 7)))
- : Thread
#<thread>
```

```
Path-String
```

The union of the `Path` and `String` types. Note that this does not match exactly what the predicate `path-string?` recognizes. For example, strings that contain the character `#\nul` have the type `Path-String` but `path-string?` returns `#f` for those strings. For a complete specification of which strings `path-string?` accepts, see its documentation.

1.2 Singleton Types

Some kinds of data are given singleton types by default. In particular, booleans, symbols, and keywords have types which consist only of the particular boolean, symbol, or keyword. These types are subtypes of `Boolean`, `Symbol` and `Keyword`, respectively.

Examples:

```
> #t
- : Boolean [more precisely: True]
#t
> '#:foo
- : '#:foo
'#:foo
> 'bar
- : Symbol [more precisely: 'bar]
'bar
```

1.3 Containers

The following base types are parametric in their type arguments.

`(Pairof s t)`

is the pair containing `s` as the `car` and `t` as the `cdr`

Examples:

```
> (cons 1 2)
- : (Pairof One Positive-Byte)
'(1 . 2)
> (cons 1 "one")
- : (Pairof One String)
'(1 . "one")
```

`(Listof t)`

Homogenous lists of `t`

```
(List t ...)
```

is the type of the list with one element, in order, for each type provided to the `List` type constructor.

```
(List t ... trest ... bound)
```

is the type of a list with one element for each of the `t`s, plus a sequence of elements corresponding to *trest*, where *bound* must be an identifier denoting a type variable bound with `...`.

```
(List* t t1 ... s)
```

is equivalent to `(Pairof t (List* t1 ... s))`.

Examples:

```
> (list 'a 'b 'c)
- : (Listof (U 'a 'b 'c)) [more precisely: (List 'a 'b 'c)]
'(a b c)
> (plambda: (a ...) ([sym : Symbol] boxes : (Boxof a) ... a)
    (ann (cons sym boxes) (List Symbol (Boxof a) ... a)))
- : (All (a ...)
      (-> Symbol (Boxof a) ... a (Pairof Symbol (List (Boxof a)
... a))))
#<procedure>
> (map symbol->string (list 'a 'b 'c))
- : (Listof String) [more precisely: (Pairof String (Listof
String))]
'("a" "b" "c")
```

```
(MListof t)
```

Homogenous mutable lists of `t`.

```
(MPairof t u)
```

Mutable pairs of `t` and `u`.

```
MPairTop
```

is the type of a mutable pair with unknown element types and is the supertype of all mutable pair types. This type typically appears in programs via the combination of occurrence typing and `mpair?`.

Example:

```
> (lambda: ([x : Any]) (if (mpair? x) x (error "not an mpair!")))
- : (-> Any MPairTop)
#<procedure>
```

```
(Boxof t)
```

A box of *t*

Example:

```
> (box "hello world")
- : (Boxof String)
'#&"hello world"
```

```
BoxTop
```

is the type of a box with an unknown element type and is the supertype of all box types. Only read-only box operations (e.g. `unbox`) are allowed on values of this type. This type typically appears in programs via the combination of occurrence typing and `box?`.

Example:

```
> (lambda: ([x : Any]) (if (box? x) x (error "not a box!")))
- : (-> Any BoxTop)
#<procedure>
```

```
(Vectorof t)
```

Homogenous vectors of *t*

```
(Vector t ...)
```

is the type of the list with one element, in order, for each type provided to the `Vector` type constructor.

Examples:

```
> (vector 1 2 3)
- : (Vector Integer Integer Integer)
'#(1 2 3)
> #(a b c)
- : (Vector Symbol Symbol Symbol)
'#(a b c)
```

FlVector

An flvector.

Example:

```
> (flvector 1.0 2.0 3.0)
- : FlVector
(flvector 1.0 2.0 3.0)
```

FxVector

An fxvector.

Example:

```
> (fxvector 1 2 3)
- : FxVector
(fxvector 1 2 3)
```

VectorTop

is the type of a vector with unknown length and element types and is the supertype of all vector types. Only read-only vector operations (e.g. `vector-ref`) are allowed on values of this type. This type typically appears in programs via the combination of occurrence typing and `vector?`.

Example:

```
> (lambda: ([x : Any]) (if (vector? x) x (error "not a vector!")))
- : (-> Any VectorTop)
#<procedure>
```

(HashTable *k* *v*)

is the type of a hash table with key type *k* and value type *v*.

Example:

```
> #hash((a . 1) (b . 2))
- : (HashTable Symbol Integer)
'#hash((a . 1) (b . 2))
```

HashTableTop

is the type of a hash table with unknown key and value types and is the supertype of all hash table types. Only read-only hash table operations (e.g. `hash-ref`) are allowed on values of this type. This type typically appears in programs via the combination of occurrence typing and `hash?`.

Example:

```
> (lambda: ([x : Any]) (if (hash? x) x (error "not a hash
table!")))
- : (-> Any HashTableTop)
#<procedure>
```

```
(Setof t)
```

is the type of a set of `t`.

Example:

```
> (set 0 1 2 3)
- : (Setof Byte)
(set 0 1 2 3)
```

```
(Channelof t)
```

A channel on which only `t`s can be sent.

Example:

```
> (ann (make-channel) (Channelof Symbol))
- : (Channelof Symbol)
#<channel>
```

```
ChannelTop
```

is the type of a channel with unknown message type and is the supertype of all channel types. This type typically appears in programs via the combination of occurrence typing and `channel?`.

Example:

```
> (lambda: ([x : Any]) (if (channel? x) x (error "not a
channel!")))
- : (-> Any ChannelTop)
#<procedure>
```

```
(Parameterof t)
(Parameterof s t)
```

A parameter of `t`. If two type arguments are supplied, the first is the type the parameter accepts, and the second is the type returned.

Examples:

```
> current-input-port
- : (Parameterof Input-Port)
#<procedure:current-input-port>
> current-directory
- : (Parameterof Path-String Path)
#<procedure:current-directory>
```

```
(Promise t)
```

A promise of `t`.

Example:

```
> (delay 3)
- : (Promise Positive-Byte)
#<promise:eval:43:0>
```

```
(Futureof t)
```

A future which produce a value of type `t` when touched.

```
(Sequenceof t)
```

A sequence that produces values of type `t` on each iteration.

```
(Custodian-Boxof t)
```

A custodian box of `t`.

```
(Thread-Cellof t)
```

A thread cell of `t`.

```
Thread-CellTop
```

is the type of a thread cell with unknown element type and is the supertype of all thread cell types. This type typically appears in programs via the combination of occurrence typing and `thread-cell?`.

Example:

```
> (lambda: ([x : Any]) (if (thread-cell? x) x (error "not a thread
cell!")))
- : (-> Any Thread-CellTop)
#<procedure>
```

```
(Ephemeronof t)
```

An ephemeron whose value is of type `t`.

```
(Evtof t)
```

A synchronizable event whose synchronization result is of type `t`.

Examples:

```
> always-evt
- : (Rec x (Evtof x))
#<always-evt>
> (system-idle-evt)
- : (Evtof Void)
#<evt>
> (ann (thread (λ () (displayln "hello world"))) (Evtof Thread))
- : (Evtof Thread)
#<thread>
```

1.4 Syntax Objects

The following types represent syntax objects and their content.

```
(Syntaxof t)
```

A syntax object with content of type `t`. Applying `syntax-e` to a value of type `(Syntaxof t)` produces a value of type `t`.

```
Identifier
```

A syntax object containing a symbol. Equivalent to `(Syntaxof Symbol)`.

```
Syntax
```

A syntax object containing only symbols, keywords, strings, characters, booleans, numbers, boxes containing `Syntax`, vectors of `Syntax`, or (possibly improper) lists of `Syntax`. Equivalent to `(Syntaxof Syntax-E)`.

`Syntax-E`

The content of syntax objects of type `Syntax`. Applying `syntax-e` to a value of type `Syntax` produces a value of type `Syntax-E`.

`(Sexpof t)`

The recursive union of `t` with symbols, keywords, strings, characters, booleans, numbers, boxes, vectors, and (possibly improper) lists.

`Sexp`

Applying `syntax->datum` to a value of type `Syntax` produces a value of type `Sexp`. Equivalent to `(Sexpof Nothing)`.

`Datum`

Applying `datum->syntax` to a value of type `Datum` produces a value of type `Syntax`. Equivalent to `(Sexpof Syntax)`.

1.5 Control

The following types represent prompt tags and keys for continuation marks for use with delimited continuation functions and continuation mark functions.

`(Prompt-Tagof s t)`

A prompt tag to be used in a continuation prompt whose body produces the type `s` and whose handler has the type `t`. The type `t` must be a function type.

The domain of `t` determines the type of the values that can be aborted, using `abort-current-continuation`, to a prompt with this prompt tag.

Example:

```
> (make-continuation-prompt-tag 'prompt-tag)
- : (Prompt-Tagof Any Any)
#<continuation-prompt-tag:prompt-tag>
```

`Prompt-TagTop`

is the type of a prompt tag with unknown body and handler types and is the supertype of all prompt tag types. This type typically appears in programs via the combination of occurrence typing and `continuation-prompt-tag?`.

Example:

```
> (lambda: ([x : Any]) (if (continuation-prompt-
tag? x) x (error "not a prompt tag!")))
- : (-> Any Prompt-TagTop)
#<procedure>
```

```
(Continuation-Mark-Keyof t)
```

A continuation mark key that is used for continuation mark operations such as `with-continuation-mark` and `continuation-mark-set->list`. The type `t` represents the type of the data that is stored in the continuation mark with this key.

Example:

```
> (make-continuation-mark-key 'mark-key)
hello world
- : (Continuation-Mark-Keyof Any)
#<continuation-mark-key>
```

```
Continuation-Mark-KeyTop
```

is the type of a continuation mark key with unknown element type and is the supertype of all continuation mark key types. This type typically appears in programs via the combination of occurrence typing and `continuation-mark-key?`.

Example:

```
> (lambda: ([x : Any]) (if (continuation-mark-
key? x) x (error "not a mark key!")))
- : (-> Any Continuation-Mark-KeyTop)
#<procedure>
```

1.6 Other Type Constructors

```
(-> dom ... rng)
(-> dom ... rest * rng)
(-> dom ... rest ooo bound rng)
(-> dom rng : pred)
(dom ... -> rng)
(dom ... rest * -> rng)
(dom ... rest ooo bound -> rng)
(dom -> rng : pred)
```

```
          ooo = ...

          dom = type
              | mandatory-kw
              | optional-kw

mandatory-kw = keyword type

 optional-kw = [keyword type]
```

The type of functions from the (possibly-empty) sequence *dom* `...` to the *rng* type.

Examples:

```
> (λ: ([x : Number]) x)
- : (-> Number Number)
#<procedure>
> (λ: () 'hello)
- : (-> 'hello)
#<procedure>
```

The second form specifies a uniform rest argument of type *rest*, and the third form specifies a non-uniform rest argument of type *rest* with bound *bound*. The bound refers to the type variable that is in scope within the rest argument type.

Examples:

```
> (λ: ([x : Number]    y : String *)  (length y))
- : (-> Number String * Index)
#<procedure>
> ormap
- : (All (a c b ...)
      (-> (-> a b ... b c) (Listof a) (Listof b) ... b (U False
c)))
#<procedure:ormap>
```

In the third form, the `...` introduced by *ooo* is literal, and *bound* must be an identifier denoting a type variable.

In the fourth form, there must be only one *dom* and *pred* is the type checked by the predicate. The type *pred* is known as a *filter* for the function type (for an introduction to filters, see §5.2 “Filters and Predicates”).

Example:

```
> string?
- : (-> Any Boolean : String)
#<procedure:string?>
```

The *dom*s can include both mandatory and optional keyword arguments. Mandatory keyword arguments are a pair of keyword and type, while optional arguments are surrounded by a pair of parentheses.

Examples:

```
> (:print-type file->string)
(-> Path-String [#:mode (U 'binary 'text)] String)

> (: is-zero? : (-> Number #:equality (-> Number Number Any) [#:zero Number] Any))

> (define (is-zero? n #:equality equality #:zero [zero 0])
    (equality n zero))

> (is-zero? 2 #:equality =)
- : Any
#f
> (is-zero? 2 #:equality eq? #:zero 2.0)
- : Any
#f
```

The type of functions can also be specified with an *infix* `->` which comes immediately before the *rng* type. The fifth through eighth forms match the first four cases, but with the infix style of arrow.

Examples:

```
> (: add2 (Number -> Number))

> (define (add2 n) (+ n 2))
```

```
(->* (mandatory-dom ...) optional-doms rest rng)

mandatory-dom = type
              | keyword type

optional-doms =
              | (optional-dom ...)

 optional-dom = type
              | keyword type

         rest =
              | #:rest type
```

Constructs the type of functions with optional or rest arguments. The first list of *mandatory-dom*s correspond to mandatory argument types. The list *optional-doms*, if provided, specifies the optional argument types.

Examples:

```
> (: append-bar (->* (String) (Positive-Integer) String))

> (define (append-bar str [how-many 1])
    (apply string-append str (make-list how-many "bar")))
```

If provided, the *rest* expression specifies the type of elements in the rest argument list.

Examples:

```
> (: +all (->* (Integer) #:rest Integer (Listof Integer)))

> (define (+all inc . rst)
    (map (λ: ([x : Integer]) (+ x inc)) rst))

> (+all 20 1 2 3)
- : (Listof Integer)
'(21 22 23)
```

Both the mandatory and optional argument lists may contain keywords paired with types.

Examples:

```
> (: kw-f (->* (#:x Integer) (#:y Integer) Integer))

> (define (kw-f #:x x #:y [y 0]) (+ x y))
```

The syntax for this type constructor matches the syntax of the `->*` contract combinator, but with types instead of contracts.

```
Top
Bot
```

These are filters that can be used with `->`. `Top` is the filter with no information. `Bot` is the filter which means the result cannot happen.

```
Procedure
```

is the supertype of all function types. The `Procedure` type corresponds to values that satisfy the `procedure?` predicate. Because this type encodes *only* the fact that the value is a procedure, and *not* its argument types or even arity, the type-checker cannot allow values of this type to be applied.

For the types of functions with known arity and argument types, see the `->` type constructor.

Examples:

```
> (: my-list Procedure)

> (define my-list list)

> (my-list "zwiebelkuchen" "socca")
eval:73:0: Type Checker: Cannot apply expression of type
Procedure, since it is not a function type
  in: "socca"
```

```
(U t ...)
```

is the union of the types `t ...`.

Example:

```
> (λ: ([x : Real])(if (> 0 x) "yes" 'no))
- : (-> Real (U String 'no))
#<procedure>
```

```
(case-> fun-ty ...)
```

is a function that behaves like all of the *fun-ty*s, considered in order from first to last. The *fun-ty*s must all be function types constructed with `->`.

Example:

```
> (: add-map : (case->
                [(Listof Integer) -> (Listof Integer)]
                [(Listof Integer) (Listof Integer) -> (Listof Integer)]))
```

For the definition of `add-map` look into `case-lambda:`.

```
(t t1 t2 ...)
```

is the instantiation of the parametric type `t` at types `t1 t2 ...`

```
(All (a ...) t)
(All (a ... a ooo) t)
```

is a parameterization of type `t`, with type variables `v ...`. If `t` is a function type constructed with infix `->`, the outer pair of parentheses around the function type may be omitted.

Examples:

```
> (: list-length : (All (A) (Listof A) -> Natural))

> (define (list-length lst)
    (if (null? lst)
        0
        (add1 (list-length (cdr lst)))))

> (list-length (list 1 2 3))
- : Integer [more precisely: Nonnegative-Integer]
3
```

```
(Values t ...)
```

is the type of a sequence of multiple values, with types `t ...`. This can only appear as the return type of a function.

Example:

```
> (values 1 2 3)
- : (values Integer Integer Integer) [more precisely: (Values One
Positive-Byte Positive-Byte)]
1
2
3
```

```
v
```

where `v` is a number, boolean or string, is the singleton type containing only that value

```
(quote val)
```

where `val` is a Racket value, is the singleton type containing only that value

```
i
```

where `i` is an identifier can be a reference to a type name or a type variable

```
(Rec n t)
```

is a recursive type where `n` is bound to the recursive type in the body `t`

Examples:

```
> (define-type IntList (Rec List (Pair Integer (U List Null))))

> (define-type (List A) (Rec List (Pair A (U List Null))))
```

`(Struct st)`

is a type which is a supertype of all instances of the potentially-polymorphic structure type *st*. Note that structure accessors for *st* will *not* accept `(Struct st)` as an argument.

`(Struct-Type st)`

is a type for the structure type descriptor value for the structure type *st*. Values of this type are used with reflective operations such as `struct-type-info`.

Examples:

```
> struct:arity-at-least
- : (StructType arity-at-least)
#<struct-type:arity-at-least>
> (struct-type-info struct:arity-at-least)
- : (values
     Symbol
     Integer
     Integer
     (-> arity-at-least Nonnegative-Integer Any)
     (-> arity-at-least Nonnegative-Integer Nothing Void)
     (Listof Nonnegative-Integer)
     (U False Struct-TypeTop)
     Boolean)
[more precisely: (values
                  Symbol
                  Nonnegative-Integer
                  Nonnegative-Integer
                  (-> arity-at-least Nonnegative-Integer Any)
                  (-> arity-at-least Nonnegative-Integer Nothing
Void)
                  (Listof Nonnegative-Integer)
                  (U False Struct-TypeTop)
                  Boolean)]
'arity-at-least
1
0
#<procedure:arity-at-least-ref>
```

```
#<procedure:arity-at-least-set!>
'(0)
#f
#f
```

`Struct-TypeTop`

is the supertype of all types for structure type descriptor values. The corresponding structure type is unknown for values of this top type.

Example:

```
> (struct-info (arity-at-least 0))
- : (values (U False Struct-TypeTop) Boolean)
#<struct-type:arity-at-least>
#f
```

`→`

An alias for `->`.

`case→`

An alias for `case->`.

`∀`

An alias for `All`.

1.7 Other Types

`(Option t)`

Either `t` or `#f`

`(Opaque t)`

A type constructed using the `#:opaque` clause of `require/typed`.

2 Special Form Reference

Typed Racket provides a variety of special forms above and beyond those in Racket. They are used for annotating variables with types, creating new types, and annotating expressions.

2.1 Binding Forms

loop, *f*, *a*, and *var* are names, *type* is a type. *e* is an expression and *body* is a block.

```
(let maybe-tvars (binding ...) . body)
(let loop maybe-ret (binding ...) . body)

    binding = [var e]
            | [var : type e]

maybe-tvars =
            | #:forall (tvar ...)
            | #:∀ (tvar ...)

  maybe-ret =
            | : type0
```

Local bindings, like `let`, each with associated types. In the second form, *type0* is the type of the result of *loop* (and thus the result of the entire expression as well as the final expression in *body*). Type annotations are optional.

Examples:

```
> (: filter-even : (-> (Listof Natural) (Listof Natural) (Listof Natural)))

> (define (filter-even lst accum)
    (if (null? lst)
        accum
        (let ([first : Natural (car lst)]
              [rest  : (Listof Natural) (cdr lst)])
              (if (even? first)
                  (filter-even rest (cons first accum))
                  (filter-even rest accum)))))

> (filter-even (list 1 2 3 4 5 6) null)
- : (Listof Nonnegative-Integer)
'(6 4 2)
```

Examples:

```
> (: filter-even-loop (-> (Listof Natural) (Listof Natural)))

> (define (filter-even-loop lst)
    (let loop : (Listof Natural)
         ([accum : (Listof Natural) null]
          [lst   : (Listof Natural) lst])
         (cond
           [(null? lst)       accum]
           [(even? (car lst)) (loop (cons (car lst) accum) (cdr lst))]
           [else              (loop accum (cdr lst))])))

> (filter-even-loop (list 1 2 3 4))
- : (Listof Nonnegative-Integer)
'(4 2)
```

If polymorphic type variables are provided, they are bound in the type expressions for variable bindings.

Example:

```
> (let #:forall (A) ([x : A 0]) x)
- : Integer [more precisely: Zero]
0
```

```
(letrec (binding ...) . body)
(let* (binding ...) . body)
(let-values ([(var+type ...) e] ...) . body)
(letrec-values ([(var+type ...) e] ...) . body)
(let*-values ([(var+type ...) e] ...) . body)
```

Type-annotated versions of `letrec`, `let*`, `let-values`, `letrec-values`, and `let*-values`. As with `let`, type annotations are optional.

```
(let/cc v : t . body)
(let/ec v : t . body)
```

Type-annotated versions of `let/cc` and `let/ec`.

2.2 Anonymous Functions

```
(lambda maybe-tvars formals maybe-ret . body)
```

```
   formals = (formal ...)
           | (formal ... . rst)

    formal = var
           | [var default-expr]
           | [var : type]
           | [var : type default-expr]
           | keyword var
           | keyword [var : type]
           | keyword [var : type default-expr]

       rst = var
           | [var : type *]
           | [var : type ooo bound]

maybe-tvars =
           | #:forall (tvar ...)
           | #:∀ (tvar ...)
           | #:forall (tvar ... ooo)
           | #:∀ (tvar ... ooo)

 maybe-ret =
           | : type
```

Constructs an anonymous function like the `lambda` form from `racket/base`, but allows type annotations on the formal arguments. If a type annotation is left out, the formal will have the type `Any`.

Examples:

```
> (lambda ([x : String]) (string-append x "bar"))
- : (-> String String)
#<procedure>
> (lambda (x [y : Integer]) (add1 y))
- : (-> Any Integer Integer)
#<procedure>
> (lambda (x) x)
- : (-> Any Any)
#<procedure>
```

Type annotations may also be specified for keyword and optional arguments:

Examples:

```
> (lambda ([x : String "foo"]) (string-append x "bar"))
- : (->* () (String) String)
#<procedure>
```

```
> (lambda (#:x [x : String]) (string-append x "bar"))
- : (-> #:x String String)
#<procedure:eval:13:0>
> (lambda (x #:y [y : Integer 0]) (add1 y))
- : (-> Any [#:y Integer] Integer)
#<procedure:eval:14:0>
> (lambda ([x 'default]) x)
- : (->* () (Any) Any)
#<procedure>
```

The `lambda` expression may also specify polymorphic type variables that are bound for the type expressions in the formals.

Examples:

```
> (lambda #:forall (A) ([x : A]) x)
- : (All (A) (-> A A))
#<procedure>
> (lambda #:∀ (A) ([x : A]) x)
- : (All (A) (-> A A))
#<procedure>
```

In addition, a type may optionally be specified for the rest argument with either a uniform type or using a polymorphic type. In the former case, the rest argument is given the type `(Listof` *type*`)` where *type* is the provided type annotation.

Examples:

```
> (lambda (x . rst) rst)
- : (-> Any Any * (Listof Any))
#<procedure>
> (lambda (x rst : Integer *) rst)
- : (-> Any Integer * (Listof Integer))
#<procedure>
> (lambda #:forall (A ...) (x rst : A ... A) rst)
- : (All (A ...) (-> Any A ... A (List A ... A)))
#<procedure>
```

(λ *formals* . *body*)

An alias for the same form using `lambda`.

(case-lambda *maybe-tvars* [*formals* *body*] ...)

A function of multiple arities. The *formals* are identical to those accepted by the `lambda` form except that keyword and optional arguments are not allowed.

Polymorphic type variables, if provided, are bound in the type expressions in the formals.

Note that each *formals* must have a different arity.

Example:

```
> (define add-map
    (case-lambda
     [([lst : (Listof Integer)])
      (map add1 lst)]
     [([lst1 : (Listof Integer)]
       [lst2 : (Listof Integer)])
      (map + lst1 lst2)]))
```

To see how to declare a type for `add-map`, see the `case->` type constructor.

2.3 Loops

```
(for type-ann-maybe (for:-clause ...)
  expr ...+)

type-ann-maybe =
               | : u

    for-clause = [id : t seq-expr]
               | [(binding ...) seq-expr]
               | [id seq-expr]
               | #:when guard

       binding = id
               | [id : t]
```

Like `for` from `racket/base`, but each *id* has the associated type *t*. Since the return type is always `Void`, annotating the return type of a `for` form is optional. Type annotations in clauses are optional for all `for` variants.

```
(for/list type-ann-maybe (for-clause ...) expr ...+)
(for/hash type-ann-maybe (for-clause ...) expr ...+)
(for/hasheq type-ann-maybe (for-clause ...) expr ...+)
(for/hasheqv type-ann-maybe (for-clause ...) expr ...+)
(for/vector type-ann-maybe (for-clause ...) expr ...+)
(for/flvector type-ann-maybe (for-clause ...) expr ...+)
(for/and type-ann-maybe (for-clause ...) expr ...+)
(for/or   type-ann-maybe (for-clause ...) expr ...+)
```

```
(for/first type-ann-maybe (for-clause ...) expr ...+)
(for/last type-ann-maybe (for-clause ...) expr ...+)
(for/sum type-ann-maybe (for-clause ...) expr ...+)
(for/product type-ann-maybe (for-clause ...) expr ...+)
(for*/list type-ann-maybe (for-clause ...) expr ...+)
(for*/hash type-ann-maybe (for-clause ...) expr ...+)
(for*/hasheq type-ann-maybe (for-clause ...) expr ...+)
(for*/hasheqv type-ann-maybe (for-clause ...) expr ...+)
(for*/vector type-ann-maybe (for-clause ...) expr ...+)
(for*/flvector type-ann-maybe (for-clause ...) expr ...+)
(for*/and type-ann-maybe (for-clause ...) expr ...+)
(for*/or   type-ann-maybe (for-clause ...) expr ...+)
(for*/first type-ann-maybe (for-clause ...) expr ...+)
(for*/last type-ann-maybe (for-clause ...) expr ...+)
(for*/sum type-ann-maybe (for-clause ...) expr ...+)
(for*/product type-ann-maybe (for-clause ...) expr ...+)
```

These behave like their non-annotated counterparts, with the exception that `#:when` clauses can only appear as the last *for-clause*. The return value of the entire form must be of type `u`. For example, a `for/list` form would be annotated with a `Listof` type. All annotations are optional.

```
(for/lists type-ann-maybe ([id : t] ...)
  (for-clause ...)
  expr ...+)
(for/fold  type-ann-maybe ([id : t init-expr] ...)
  (for-clause ...)
  expr ...+)
```

These behave like their non-annotated counterparts. Unlike the above, `#:when` clauses can be used freely with these.

```
(for* void-ann-maybe (for-clause ...)
  expr ...+)
(for*/lists type-ann-maybe ([id : t] ...)
  (for-clause ...)
  expr ...+)
(for*/fold  type-ann-maybe ([id : t init-expr] ...)
  (for-clause ...)
  expr ...+)
```

These behave like their non-annotated counterparts.

```
(do : u ([id : t init-expr step-expr-maybe] ...)
        (stop?-expr finish-expr ...)
  expr ...+)
```

```
step-expr-maybe =
                | step-expr
```

Like `do` from `racket/base`, but each *id* having the associated type *t*, and the final body *expr* having the type *u*. Type annotations are optional.

2.4 Definitions

```
(define maybe-tvars v maybe-ann e)
(define maybe-tvars header maybe-ann . body)

     header = (function-name . formals)
            | (header . formals)

    formals = (formal ...)
            | (formal ... . rst)

     formal = var
            | [var default-expr]
            | [var : type]
            | [var : type default-expr]
            | keyword var
            | keyword [var : type]
            | keyword [var : type default-expr]

        rst = var
            | [var : type *]
            | [var : type ooo bound]

maybe-tvars =
            | #:forall (tvar ...)
            | #:∀ (tvar ...)
            | #:forall (tvar ... ooo)
            | #:∀ (tvar ... ooo)

  maybe-ann =
            | : type
```

Like `define` from `racket/base`, but allows optional type annotations for the variables.

The first form defines a variable *v* to the result of evaluating the expression *e*. The variable may have an optional type annotation.

Examples:

```
> (define foo "foo")

> (define bar : Integer 10)
```

If polymorphic type variables are provided, then they are bound for use in the type annotation.

Example:

```
> (define #:forall (A) mt-seq : (Sequenceof A) empty-sequence)
```

The second form allows the definition of functions with optional type annotations on any variables. If a return type annotation is provided, it is used to check the result of the function.

Like `lambda`, optional and keyword arguments are supported.

Examples:

```
> (define (add [first : Integer]
               [rest  : Integer]) : Integer
    (+ first rest))

> (define #:forall (A)
          (poly-app [func : (A A -> A)]
                    [first : A]
                    [rest  : A]) : A
    (func first rest))
```

The function definition form also allows curried function arguments with corresponding type annotations.

Examples:

```
> (define ((addx [x : Number]) [y : Number]) (+ x y))

> (define add2 (addx 2))

> (add2 5)
- : Number
7
```

Note that unlike `define` from `racket/base`, `define` does not bind functions with keyword arguments to static information about those functions.

2.5 Structure Definitions

```
(struct maybe-type-vars name-spec ([f : t] ...) options ...)

maybe-type-vars =
                | (v ...)

      name-spec = name
                | name parent

        options = #:transparent
                | #:mutable
```

Defines a structure with the name *name*, where the fields *f* have types *t*, similar to the behavior of `struct` from `racket/base`. When *parent* is present, the structure is a sub-structure of *parent*. When *maybe-type-vars* is present, the structure is polymorphic in the type variables *v*. If *parent* is also a polymorphic struct, then there must be at least as many type variables as in the parent type, and the parent type is instantiated with a prefix of the type variables matching the amount it needs.

Options provided have the same meaning as for the `struct` form from `racket/base`.

```
(define-struct maybe-type-vars name-spec ([f : t] ...) options ...)

maybe-type-vars =
                | (v ...)

      name-spec = name
                | (name parent)

        options = #:transparent
                | #:mutable
```

Legacy version of `struct`, corresponding to `define-struct` from `racket/base`.

```
(define-struct/exec name-spec ([f : t] ...) [e : proc-t])

name-spec = name
          | (name parent)
```

Like `define-struct`, but defines a procedural structure. The procdure *e* is used as the value for `prop:procedure`, and must have type *proc-t*.

2.6 Names for Types

```
(define-type name t maybe-omit-def)
(define-type (name v ...) t maybe-omit-def)

maybe-omit-def = #:omit-define-syntaxes
               |
```

The first form defines *name* as type, with the same meaning as *t*. The second form is equivalent to `(define-type name (All (v ...) t))`. Type names may refer to other types defined in the same or enclosing scopes.

Examples:

```
> (define-type IntStr (U Integer String))

> (define-type (ListofPairs A) (Listof (Pair A A)))
```

If `#:omit-define-syntaxes` is specified, no definition of *name* is created. In this case, some other definition of *name* is necessary.

If the body of the type definition refers to itself, then the type definition is recursive. Recursion may also occur mutually, if a type refers to a chain of other types that eventually refers back to itself.

Examples:

```
> (define-type BT (U Number (Pair BT BT)))

> (let ()
    (define-type (Even A) (U Null (Pairof A (Odd A))))
    (define-type (Odd A) (Pairof A (Even A)))
    (: even-lst (Even Integer))
    (define even-lst '(1 2))
    even-lst)
- : (Even Integer)
'(1 2)
```

However, the recursive reference may not occur immediately inside the type:

Examples:

```
> (define-type Foo Foo)
eval:34:0: Type Checker: Error in macro expansion --
Recursive types are not allowed directly inside their
definition
  in: Foo
> (define-type Bar (U Bar False))
```

eval:35:0: Type Checker: Error in macro expansion --
Recursive types are not allowed directly inside their
definition
in: False

2.7 Generating Predicates Automatically

```
(make-predicate t)
```

Evaluates to a predicate for the type `t`, with the type `(Any -> Boolean : t)`. `t` may not contain function types, or types that may refer to mutable data such as `(Vectorof Integer)`.

```
(define-predicate name t)
```

Equivalent to `(define name (make-predicate t))`.

2.8 Type Annotation and Instantiation

```
(: v t)
(: v : t)
```

This declares that `v` has type `t`. The definition of `v` must appear after this declaration. This can be used anywhere a definition form may be used.

Examples:

```
> (: var1 Integer)
```

```
> (: var2 String)
```

The second form allows type annotations to elide one level of parentheses for function types.

Examples:

```
> (: var3 : -> Integer)
```

```
> (: var4 : String -> Integer)
```

```
(provide: [v t] ...)
```

This declares that the `v`s have the types `t`, and also provides all of the `v`s.

```
#{v : t}
```

This declares that the variable `v` has type `t`. This is legal only for binding occurrences of `v`.

```
(ann e t)
```

Ensure that `e` has type `t`, or some subtype. The entire expression has type `t`. This is legal only in expression contexts.

```
#{e :: t}
```

A reader abbreviation for `(ann e t)`.

```
(cast e t)
```

The entire expression has the type `t`, while `e` may have any type. The value of the entire expression is the value returned by `e`, protected by a contract ensuring that it has type `t`. This is legal only in expression contexts.

Examples:

```
> (cast 3 Integer)
- : Integer
3
> (cast 3 String)
- : String
3: broke its contract
   promised: String
   produced: 3
   in: String
   contract from: cast
   blaming: cast
   at: eval:41.0
> (cast (lambda: ([x : Any]) x) (String -> String))
- : (-> String String)
#<procedure:val>
```

```
(inst e t ...)
(inst e t ... t ooo bound)
```

Instantiate the type of `e` with types `t ...` or with the poly-dotted types `t ... t ooo bound`. `e` must have a polymorphic type that can be applied to the supplied number of type variables. This is legal only in expression contexts.

Examples:

```
> (foldl (inst cons Integer Integer) null (list 1 2 3 4))
- : (Listof Integer)
'(4 3 2 1)
> (: fold-list : (All (A) (Listof A) -> (Listof A)))

> (define (fold-list lst)
    (foldl (inst cons A A) null lst))

> (fold-list (list "1" "2" "3" "4"))
- : (Listof String)
'("4" "3" "2" "1")
> (: my-values : (All (A B ...) (A B ... -> (values A B ... B))))

> (define (my-values arg . args)
    (apply (inst values A B ... B) arg args))
```

```
#{e @ t ...}
```

A reader abbreviation for `(inst e t ...)`.

```
#{e @ t ... t ooo bound}
```

A reader abbreviation for `(inst e t ... t ooo bound)`.

2.9 Require

Here, *m* is a module spec, *pred* is an identifier naming a predicate, and *maybe-renamed* is an optionally-renamed identifier.

```
(require/typed m rt-clause ...)

    rt-clause = [maybe-renamed t]
              | [#:struct name ([f : t] ...)
                    struct-option ...]
              | [#:struct (name parent) ([f : t] ...)
                    struct-option ...]
              | [#:opaque t pred]

maybe-renamed = id
              | (orig-id new-id)

struct-option = #:constructor-name constructor-id
              | #:extra-constructor-name constructor-id
```

This form requires identifiers from the module `m`, giving them the specified types.

The first case requires *`maybe-renamed`*, giving it type `t`.

The second and third cases require the struct with name *`name`* with fields `f ...`, where each field has type `t`. The third case allows a `parent` structure type to be specified. The parent type must already be a structure type known to Typed Racket, either built-in or via `require/typed`. The structure predicate has the appropriate Typed Racket filter type so that it may be used as a predicate in `if` expressions in Typed Racket.

Examples:

```
> (module UNTYPED racket/base
    (define n 100)

    (struct IntTree
      (elem left right))

    (provide n (struct-out IntTree)))

> (module TYPED typed/racket
    (require/typed 'UNTYPED
                   [n Natural]
                   [#:struct IntTree
                     ([elem  : Integer]
                      [left  : IntTree]
                      [right : IntTree])]))
```

The fourth case defines a new type `t`. `pred`, imported from module `m`, is a predicate for this type. The type is defined as precisely those values to which `pred` produces `#t`. `pred` must have type `(Any -> Boolean)`. Opaque types must be required lexically before they are used.

Examples:

```
> (require/typed racket/base
                 [#:opaque Evt evt?]
                 [alarm-evt (Real -> Evt)]
                 [sync (Evt -> Any)])
eval:51:0: Type Checker: Type (-> Real Error) could not be
converted to a contract: contract generation not supported
for this type
  in: (Real -> Evt)
> evt?
eval:51:0: Type Checker: parse error in type;
 type name 'Evt' is unbound
```

```
  in: Evt
> (sync (alarm-evt (+ 100 (current-inexact-milliseconds))))
- : (Rec x (Evtof x))
#<alarm-evt>
```

In all cases, the identifiers are protected with contracts which enforce the specified types. If this contract fails, the module `m` is blamed.

Some types, notably the types of predicates such as `number?`, cannot be converted to contracts and raise a static error when used in a `require/typed` form. Here is an example of using `case->` in `require/typed`.

```
(require/typed racket/base
               [file-or-directory-modify-seconds
                (case->
                  [String -> Exact-Nonnegative-Integer]
                  [String (Option Exact-Nonnegative-Integer)
                          ->
                          (U Exact-Nonnegative-Integer Void)]
                  [String (Option Exact-Nonnegative-
Integer) (-> Any)
                          ->
                          Any])])
```

`file-or-directory-modify-seconds` has some arguments which are optional, so we need to use `case->`.

```
(require/typed/provide m rt-clause ...)
```

Similar to `require/typed`, but also provides the imported identifiers. Uses outside of a module top-level raise an error.

Examples:

```
> (module evts typed/racket
    (require/typed/provide racket/base
                           [#:opaque Evt evt?]
                           [alarm-evt (Real -> Evt)]
                           [sync (Evt -> Any)]))

> (require 'evts)

> (sync (alarm-evt (+ 100 (current-inexact-milliseconds))))
- : Any
#<alarm-evt>
```

2.10 Other Forms

`with-handlers`

Identical to `with-handlers` from `racket/base` but provides additional annotations to assist the typechecker.

```
(default-continuation-prompt-tag)
 → (-> (Prompt-Tagof Any (Any -> Any)))
```

Identical to `default-continuation-prompt-tag`, but additionally protects the resulting prompt tag with a contract that wraps higher-order values, such as functions, that are communicated with that prompt tag. If the wrapped value is used in untyped code, a contract error will be raised.

Examples:

```
> (module typed typed/racket
    (provide do-abort)
    (: do-abort (-> Void))
    (define (do-abort)
      (abort-current-continuation
       ; typed, and thus contracted, prompt tag
       (default-continuation-prompt-tag)
       (λ: ([x : Integer]) (+ 1 x)))))

> (module untyped racket
    (require 'typed)
    (call-with-continuation-prompt
      (λ () (do-abort))
      (default-continuation-prompt-tag)
      ; the function cannot be passed an argument
      (λ (f) (f 3))))

> (require 'untyped)
default-continuation-prompt-tag: broke its contract
  Attempted to use a higher-order value passed as ‘Any‘ in
untyped code: #<procedure>
  in: the range of
      (-> (prompt-tag/c Any #:call/cc Any))
  contract from: untyped
  blaming: untyped
```

`(#%module-begin form ...)`

Legal only in a module begin context. The `#%module-begin` form of `typed/racket` checks all the forms in the module, using the Typed Racket type checking rules. All `provide`

forms are rewritten to insert contracts where appropriate. Otherwise, the `#%module-begin` form of `typed/racket` behaves like `#%module-begin` from `racket`.

```
(#%top-interaction . form)
```

Performs type checking of forms entered at the read-eval-print loop. The `#%top-interaction` form also prints the type of *form* after type checking.

3 Libraries Provided With Typed Racket

The typed/racket language corresponds to the racket language—that is, any identifier provided by racket, such as modulo is available by default in typed/racket.

```
#lang typed/racket
(modulo 12 2)
```

The typed/racket/base language corresponds to the racket/base language.

Some libraries have counterparts in the typed collection, which provide the same exports as the untyped versions. Such libraries include srfi/14, net/url, and many others.

```
#lang typed/racket
(require typed/srfi/14)
(char-set= (string->char-set "hello")
           (string->char-set "olleh"))
```

Other libraries can be used with Typed Racket via require/typed.

```
#lang typed/racket
(require/typed version/check
               [check-version (-> (U Symbol (Listof Any)))])
(check-version)
```

The following libraries are included with Typed Racket in the typed collection:

(require typed/file) package: typed-racket-more

(require typed/net/base64) package: typed-racket-more

(require typed/net/cgi) package: typed-racket-more

(require typed/net/cookie) package: typed-racket-more

(require typed/net/dns) package: typed-racket-more

(require typed/net/ftp) package: typed-racket-more

(require typed/net/gifwrite) package: typed-racket-more

(require typed/net/head) package: typed-racket-more

(require typed/net/imap) package: typed-racket-more

(require typed/net/mime) package: typed-racket-more

```
(require typed/net/nntp)       package: typed-racket-more

(require typed/net/pop3)       package: typed-racket-more

(require typed/net/qp)       package: typed-racket-more

(require typed/net/sendmail)       package: typed-racket-more

(require typed/net/sendurl)       package: typed-racket-more

(require typed/net/smtp)       package: typed-racket-more

 (require typed/net/uri-codec)
        package: typed-racket-more

(require typed/net/url)       package: typed-racket-more

(require typed/pict)       package: typed-racket-more

(require typed/rackunit)       package: typed-racket-more

(require typed/srfi/14)       package: typed-racket-more

(require typed/syntax/stx)       package: typed-racket-more
```

Other libraries included in the main distribution that are either written in Typed Racket or have adapter modules that are typed:

```
(require math)       package: math-lib

(require plot/typed)       package: plot-gui-lib
```

3.1 Porting Untyped Modules to Typed Racket

To adapt a Racket library not included with Typed Racket, the following steps are required:

- Determine the data manipulated by the library, and how it will be represented in Typed Racket.
- Specify that data in Typed Racket, using `require/typed` and `#:opaque` and/or `#:struct`.
- Use the data types to import the various functions and constants of the library.
- Provide all the relevant identifiers from the new adapter module.

For example, the following module adapts the untyped `racket/bool` library:

```
#lang typed/racket
(require/typed racket/bool
               [true Boolean]
               [false Boolean]
               [symbol=? (Symbol Symbol -> Boolean)]
               [boolean=? (Boolean Boolean -> Boolean)]
               [false? (Any -> Boolean)])
(provide true false symbol=? boolean=? false?)
```

More substantial examples are available in the `typed` collection.

4 Typed Classes

Warning: the features described in this section are experimental and may not work correctly. Some of the features will change by the next release. In particular, typed-untyped interaction for classes will not be backwards compatible so do not rely on the current semantics.

Typed Racket provides support for object-oriented programming with the classes and objects provided by the `racket/class` library.

4.1 Special forms

`(require typed/racket/class)` package: `typed-racket-lib`

The special forms below are provided by the `typed/racket/class` and `typed/racket` modules but not by `typed/racket/base`. The `typed/racket/class` module additional provides all other bindings from `racket/class`.

```
(class superclass-expr
  maybe-type-parameters
  class-clause ...)
```

```
        class-clause = (inspect inspector-expr)
                     | (init init-decl ...)
                     | (init-field init-decl ...)
                     | (init-rest id/type)
                     | (field field-decl ...)
                     | (inherit-field field-decl ...)
                     | (public maybe-renamed/type ...)
                     | (pubment maybe-renamed/type ...)
                     | (override maybe-renamed/type ...)
                     | (augment maybe-renamed/type ...)
                     | (private id/type ...)
                     | (inherit id ...)
                     | method-definition
                     | definition
                     | expr
                     | (begin class-clause ...)

maybe-type-parameters =
                     | #:forall (type-variable ...)
                     | #:∀ (type-variable ...)

           init-decl = id/type
                     | [renamed]
                     | [renamed : type-expr]
                     | [maybe-renamed default-value-expr]
                     | [maybe-renamed : type-expr default-value-expr]

          field-decl = (maybe-renamed default-value-expr)
                     | (maybe-renamed : type-expr default-value-expr)

             id/type = id
                     | [id : type-expr]

  maybe-renamed/type = maybe-renamed
                     | [maybe-renamed : type-expr]

       maybe-renamed = id
                     | renamed

             renamed = (internal-id external-id)
```

Produces a class with type annotations that allows Typed Racket to type-check the methods, fields, and other clauses in the class.

The meaning of the class clauses are the same as in the `class` form from the `racket/class` library with the exception of the additional optional type annotations. Additional class clause

forms from `class` that are not listed in the grammar above are not currently supported in Typed Racket.

Examples:

```
> (define fish%
    (class object%
      (init [size : Real])

      (: current-size Real)
      (define current-size size)

      (super-new)

      (: get-size (-> Real))
      (define/public (get-size)
        current-size)

      (: grow (Real -> Void))
      (define/public (grow amt)
        (set! current-size (+ amt current-size)))

      (: eat ((Object [get-size (-> Real)]) -> Void))
      (define/public (eat other-fish)
        (grow (send other-fish get-size)))))

> (define dory (new fish% [size 5.5]))
```

Within a typed class form, one of the class clauses must be a call to `super-new`. Failure to call `super-new` will result in a type error. In addition, dynamic uses of `super-new` (e.g., calling it in a separate function within the dynamic extent of the class form's clauses) are restricted.

Example:

```
> (class object%
    ; Note the missing ‘super-new‘
    (init-field [x : Real 0] [y : Real 0]))
Type Checker: typed classes must call super-new at the class
top-level
  in: #%top-interaction
```

If any identifier with an optional type annotation is left without an annotation, the type-checker will assume the type `Any` (or `Procedure` for methods) for that identifier.

Examples:

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

> point%
- : (Class (init (x Any) (y Any)) (field (x Any) (y Any)))
#<class:point%>
```

When *type-variable* is provided, the class is parameterized over the given type variables. These type variables are in scope inside the body of the class. The resulting class can be instantiated at particular types using `inst`.

Examples:

```
> (define cons%
    (class object%
      #:forall (X Y)
      (super-new)
      (init-field [car : X] [cdr : Y])))

> cons%
- : (All (X Y) (Class (init (car X) (cdr Y)) (field (car X) (cdr
Y))))
#<class:cons%>
> (new (inst cons% Integer String) [car 5] [cdr "foo"])
- : (Object (field (car Integer) (cdr String)))
(object:cons% ...)
```

Initialization arguments may be provided by-name using the `new` form, by-position using the `make-object` form, or both using the `instantiate` form.

As in ordinary Racket classes, the order in which initialization arguments are declared determines the order of initialization types in the class type.

Furthermore, a class may also have a typed `init-rest` clause, in which case the class constructor takes an unbounded number of arguments by-position. The type of the `init-rest` clause must be either a `List` type, `Listof` type, or any other list type.

Examples:

```
> (define point-copy%
    ; a point% with a copy constructor
    (class object%
      (super-new)
      (init-rest [rst : (U (List Integer Integer)
                           (List (Object (field [x Integer]
                                                [y Integer]))))])
```

```
      (field [x : Integer 0] [y : Integer 0])
      (match rst
        [(list (? integer? *x) *y)
         (set! x *x) (set! y *y)]
        [(list (? (negate integer?) obj))
         (set! x (get-field x obj))
         (set! y (get-field y obj))])))

> (define p1 (make-object point-copy% 1 2))

> (make-object point-copy% p1)
- : (Object (field (x Integer) (y Integer)))
(object:point-copy% ...)
```

4.2 Types

```
(Class class-type-clause ...)

class-type-clause = name+type
                  | (init init-type ...)
                  | (init-field init-type ...)
                  | (init-rest name+type)
                  | (field name+type ...)
                  | (augment name+type ...)
                  | #:implements type-alias-id
                  | #:row-var row-var-id

        init-type = name+type
                  | [id type #:optional]

        name+type = [id type]
```

The type of a class with the given initialization argument, method, and field types.

The types of methods are provided either without a keyword, in which case they correspond to public methods, or with the `augment` keyword, in which case they correspond to a method that can be augmented.

An initialization argument type specifies a name and type and optionally a `#:optional` keyword. An initialization argument type with `#:optional` corresponds to an argument that does not need to be provided at object instantiation.

The order of initialization arguments in the type is significant, because it determines the types of by-position arguments for use with `make-object` and `instantiate`.

When *type-alias-id* is provided, the resulting class type includes all of the initialization argument, method, and field types from the specified type alias (which must be an alias for a class type). Multiple `#:implements` clauses may be provided for a single class type.

Examples:

```
> (define-type Point<%> (Class (field [x Real] [y Real])))

> (: colored-point% (Class #:implements Point<%>
                           (field [color String])))
```

When *row-var-id* is provided, the class type is an abstract type that is row polymorphic. A row polymorphic class type can be instantiated at a specific row using `inst`. Only a single `#:row-var` clause may appear in a class type.

```
ClassTop
```

The supertype of all class types. A value of this type cannot be used for subclassing, object creation, or most other class functions. Its primary use is for reflective operations such as `is-a?`.

```
(Object object-type-clause ...)

object-type-clause = name+type
                   | (field name+type ...)
```

The type of an object with the given field and method types.

Examples:

```
> (new object%)
- : (Object)
(object)
> (new (class object% (super-new) (field [x : Real 0])))
- : (Object (field (x Real)))
(object ...)
```

```
(Instance class-type-expr)
```

The type of an object that corresponds to *class-type-expr*.

This is the same as an `Object` type that has all of the method and field types from *class-type-expr*. The types for the `augment` and `init` clauses in the class type are ignored.

5 Utilities

Typed Racket provides some additional utility functions to facilitate typed programming.

```
(assert v) → A
  v : (U #f A)
(assert v p?) → B
  v : A
  p? : (A -> Any : B)
```

Verifies that the argument satisfies the constraint. If no predicate is provided, simply checks that the value is not `#f`.

See also the `cast` form.

Examples:

```
> (define: x : (U #f String) (number->string 7))

> x
- : (U False String)
"7"
> (assert x)
- : String
"7"
> (define: y : (U String Symbol) "hello")

> y
- : (U Symbol String)
"hello"
> (assert y string?)
- : String
"hello"
> (assert y boolean?)
- : Any [more precisely: Nothing]
Assertion #<procedure:boolean?> failed on "hello"
```

```
(with-asserts ([id maybe-pred] ...) body ...+)

maybe-pred =
           | predicate
```

Guard the body with assertions. If any of the assertions fail, the program errors. These assertions behave like `assert`.

```
(defined? v) → boolean?
  v : any/c
```

A predicate for determining if `v` is *not* `#<undefined>`.

```
(index? v) → boolean?
  v : any/c
```

A predicate for the `Index` type.

```
(typecheck-fail orig-stx maybe-msg maybe-id)

maybe-msg =
          | msg-string

 maybe-id =
          | #:covered-id id
```

Explicitly produce a type error, with the source location or *orig-stx*. If *msg-string* is present, it must be a literal string, it is used as the error message, otherwise the error message `"Incomplete case coverage"` is used. If *id* is present and has type `T`, then the message `"missing coverage of T"` is added to the error message.

Examples:

```
> (define-syntax (cond* stx)
    (syntax-case stx ()
      [(_ x clause ...)
       #`(cond clause ... [else (typecheck-fail #,stx "incomplete
coverage"
                                                #:covered-
id x)])]))

> (define: (f [x  : (U String Integer)]) : Boolean
    (cond* x
           [(string? x) #t]
           [(exact-nonnegative-integer? x) #f]))
Type Checker: type mismatch
  expected: Boolean
  given: (Syntaxof
          (List
           'typecheck-fail-internal
           (List
            'cond*
            'x
            (List (List 'string? 'x) True)
            (List (List 'exact-nonnegative-integer? 'x)
False))
           String
```

```
                'x))
  in: #%top-interaction
```

5.1 Untyped Utilities

```
(require typed/untyped-utils)
        package: typed-racket-more
```

These utilities help interface typed with untyped code, particularly typed libraries that use types that cannot be converted into contracts, or export syntax transformers that must expand differently in typed and untyped contexts.

```
(require/untyped-contract maybe-begin module [name subtype] ...)

maybe-begin =
            | (begin expr ...)
```

Use this form to import typed identifiers whose types cannot be converted into contracts, but have *subtypes* that can be converted into contracts.

For example, suppose we define and provide the Typed Racket function

```
(: negate (case-> (-> Index Fixnum)
                  (-> Integer Integer)))
(define (negate x) (- x))
```

Trying to use `negate` within an untyped module will raise an error because the cases cannot be distinguished by arity alone.

If the defining module for `negate` is `"my-numerics.rkt"`, it can be imported and used in untyped code this way:

```
(require/untyped-contract
 "my-numerics.rkt"
 [negate  (-> Integer Integer)])
```

The type `(-> Integer Integer)` is converted into the contract used for `negate`.

The `require/untyped-contract` form expands into a submodule with language `typed/racket/base`. Identifiers used in *`subtype`* expressions must be either in Typed Racket's base type environment (e.g. `Integer` and `Listof`) or defined by an expression in the *`maybe-begin`* form, which is spliced into the submodule. For example, the

math/matrix module imports and reexports matrix-expt, which has a case-> type, for untyped use in this way:

```
(provide matrix-expt)

(require/untyped-contract
 (begin (require "private/matrix/matrix-types.rkt"))
 "private/matrix/matrix-expt.rkt"
 [matrix-expt  ((Matrix Number) Integer -> (Matrix Number))])
```

The (require "private/matrix/matrix-types.rkt") expression imports the Matrix type.

If an identifier *name* is imported using require/untyped-contract, reexported, and imported into typed code, it has its original type, not *subtype*. In other words, *subtype* is used only to generate a contract for *name*, not to narrow its type.

Because of limitations in the macro expander, require/untyped-contract cannot currently be used in typed code.

(define-typed/untyped-identifier *name typed-name untyped-name*)

Defines an identifier *name* that expands to *typed-name* in typed contexts and to *untyped-name* in untyped contexts. Each subform must be an identifier.

Suppose we define and provide a Typed Racket function with this type:

```
(: my-filter (All (a) (-> (-> Any Any : a) (Listof Any) (Listof a))))
```

This type cannot be converted into a contract because it accepts a predicate. Worse, require/untyped-contract does not help because (All (a) (-> (-> Any Any) (Listof Any) (Listof a))) is not a subtype.

In this case, we might still provide my-filter to untyped code using

```
(provide my-filter)

(define-typed/untyped-identifier my-filter
  typed:my-filter
  untyped:my-filter)
```

where typed:my-filter is the original my-filter, but imported using prefix-in, and untyped:my-filter is either a Typed Racket implementation of it with type (All (a)

`(-> (-> Any Any) (Listof Any) (Listof a))` or an untyped Racket implementation.

Avoid this if possible. Use only in cases where a type has no subtype that can be converted to a contract; i.e. cases in which `require/untyped-contract` cannot be used.

`(syntax-local-typed-context?)` → `boolean?`

Returns `#t` if called while expanding code in a typed context; otherwise `#f`.

This is the nuclear option, provided because it is sometimes, but rarely, useful. Avoid.

6 Exploring Types

In addition to printing a summary of the types of REPL results, Typed Racket provides interactive utilities to explore and query types. The following bindings are only available at the Typed Racket REPL.

```
(:type maybe-verbose t)

maybe-verbose =
              | #:verbose
```

Prints the type `t`. If `t` is a type alias (e.g., `Number`), then it will be expanded to its representation when printing. Any further type aliases in the type named by `t` will remain unexpanded.

If `#:verbose` is provided, all type aliases are expanded in the printed type.

Examples:

```
> (:type Number)
(U Exact-Number Inexact-Complex Real Float-Imaginary Single-
Flonum-Imaginary)
[can expand further: Exact-Number Inexact-Complex Real]

> (:type Real)
(U Nonpositive-Real Nonnegative-Real)
[can expand further: Nonpositive-Real Nonnegative-Real]

> (:type #:verbose Number)
(U 0
   1
   Byte-Larger-Than-One
   Positive-Index-Not-Byte
   Positive-Fixnum-Not-Index
   Negative-Fixnum
   Positive-Integer-Not-Fixnum
   Negative-Integer-Not-Fixnum
   Positive-Rational-Not-Integer
   Negative-Rational-Not-Integer
   Float-Nan
   Float-Positive-Zero
   Float-Negative-Zero
   Positive-Float-No-NaN
   Negative-Float-No-NaN
   Single-Flonum-Nan
   Single-Flonum-Positive-Zero
```

```
    Single-Flonum-Negative-Zero
    Positive-Single-Flonum-No-Nan
    Negative-Single-Flonum-No-Nan
    Exact-Imaginary
    Exact-Complex
    Float-Imaginary
    Single-Flonum-Imaginary
    Float-Complex
    Single-Flonum-Complex)
```

```
(:print-type e)
```

Prints the type of `e`. This prints the whole type, which can sometimes be quite large.

Examples:

```
> (:print-type (+ 1 2))
Positive-Index

> (:print-type map)
(All (c a b ...)
  (case->
   (-> (-> a c) (Pairof a (Listof a)) (Pairof c (Listof c)))
   (-> (-> a b ... b c) (Listof a) (Listof b) ... b (Listof c))))
```

```
(:query-type/args f t ...)
```

Given a function `f` and argument types `t`, shows the result type of `f`.

Example:

```
> (:query-type/args + Integer Number)
(-> Integer Number Number)
```

```
(:query-type/result f t)
```

Given a function `f` and a desired return type `t`, shows the arguments types `f` should be given to return a value of type `t`.

Examples:

```
> (:query-type/result + Integer)
(-> Integer * Integer)
```

```
> (:query-type/result + Float)
(case->
 (-> Flonum Flonum * Flonum)
 (-> Real Real Flonum Real * Flonum)
 (-> Real Flonum Real * Flonum)
 (-> Flonum Real Real * Flonum))
```

7 Typed Racket Syntax Without Type Checking

`#lang typed/racket/no-check` package: `typed-racket-lib`
`#lang typed/racket/base/no-check`

On occasions where the Typed Racket syntax is useful, but actual typechecking is not desired, the `typed/racket/no-check` and `typed/racket/base/no-check` languages are useful. They provide the same bindings and syntax as `typed/racket` and `typed/racket/base`, but do no type checking.

Examples:

```
#lang typed/racket/no-check
(: x Number)
(define x "not-a-number")
```

8 Typed Regions

The `with-type` form allows for localized Typed Racket regions in otherwise untyped code.

```
(with-type result-spec fv-clause body ...+)
(with-type export-spec fv-clause body ...+)

  fv-clause =
            | #:freevars ([id fv-type] ...)

result-spec = #:result type

export-spec = ([export-id export-type] ...)
```

The first form, an expression, checks that *body ...+* has the type *type*. If the last expression in *body ...+* returns multiple values, *type* must be a type of the form `(values t ...)`. Uses of the result values are appropriately checked by contracts generated from *type*.

The second form, which can be used as a definition, checks that each of the *export-id*s has the specified type. These types are also enforced in the surrounding code with contracts.

The *id*s are assumed to have the types ascribed to them; these types are converted to contracts and checked dynamically.

Examples:

```
> (with-type #:result Number 3)
3
> ((with-type #:result (Number -> Number)
     (lambda: ([x : Number]) (add1 x)))
   #f)
contract violation:
  expected: Number
  given: #f
  in: the 1st argument of
      (-> Number any)
  contract from: (region typed-region)
  blaming: top-level
> (let ([x "hello"])
    (with-type #:result String
      #:freevars ([x String])
      (string-append x ", world")))
"hello, world"
> (let ([x 'hello])
    (with-type #:result String
      #:freevars ([x String])
```

```
      (string-append x ", world")))
x: broke its contract
  promised: String
  produced: 'hello
  in: String
  contract from: top-level
  blaming: top-level
  at: eval:5.0
> (with-type ([fun (Number -> Number)]
              [val Number])
    (define (fun x) x)
    (define val 17))

> (fun val)
17
```

9 Optimization in Typed Racket

[1]

Typed Racket provides a type-driven optimizer that rewrites well-typed programs to potentially make them faster. It should in no way make your programs slower or unsafe.

Typed Racket's optimizer is turned on by default. If you want to deactivate it (for debugging, for instance), you must add the `#:no-optimize` keyword when specifying the language of your program:

```
#lang typed/racket #:no-optimize
```

[1] See §7 "Optimization in Typed Racket" in the guide for tips to get the most out of the optimizer.

10 Legacy Forms

The following forms are provided by Typed Racket for backwards compatibility.

```
(lambda: formals . body)

formals = ([v : t] ...)
        | ([v : t] ... v : t *)
        | ([v : t] ... v : t ooo bound)
```

A function of the formal arguments `v`, where each formal argument has the associated type. If a rest argument is present, then it has type `(Listof t)`.

```
(λ: formals . body)
```

An alias for the same form using `lambda:`.

```
(plambda: (a ...) formals . body)
(plambda: (a ... b ooo) formals . body)
```

A polymorphic function, abstracted over the type variables `a`. The type variables `a` are bound in both the types of the formal, and in any type expressions in the *body*.

```
(opt-lambda: formals . body)

formals = ([v : t] ... [v : t default] ...)
        | ([v : t] ... [v : t default] ... v : t *)
        | ([v : t] ... [v : t default] ... v : t ooo bound)
```

A function with optional arguments.

```
(popt-lambda: (a ...) formals . body)
(popt-lambda: (a ... a ooo) formals . body)
```

A polymorphic function with optional arguments.

```
case-lambda:
```

An alias for `case-lambda`.

```
(pcase-lambda: (a ...) [formals body] ...)
(pcase-lambda: (a ... b ooo) [formals body] ...)
```

A polymorphic function of multiple arities.

```
(let: ([v : t e] ...) . body)
(let: loop : t0 ([v : t e] ...) . body)
```

Local bindings, like `let`, each with associated types. In the second form, *t0* is the type of the result of *loop* (and thus the result of the entire expression as well as the final expression in *body*). Type annotations are optional.

Examples:

```
> (: filter-even : (Listof Natural) (Listof Natural) -> (Listof Natural))

> (define (filter-even lst accum)
    (if (null? lst)
        accum
        (let: ([first : Natural (car lst)]
               [rest  : (Listof Natural) (cdr lst)])
              (if (even? first)
                  (filter-even rest (cons first accum))
                  (filter-even rest accum)))))

> (filter-even (list 1 2 3 4 5 6) null)
- : (Listof Nonnegative-Integer)
'(6 4 2)
```

Examples:

```
> (: filter-even-loop : (Listof Natural) -> (Listof Natural))

> (define (filter-even-loop lst)
    (let: loop : (Listof Natural)
          ([accum : (Listof Natural) null]
           [lst   : (Listof Natural) lst])
          (cond
            [(null? lst)       accum]
            [(even? (car lst)) (loop (cons (car lst) accum) (cdr lst))]
            [else              (loop accum (cdr lst))])))

> (filter-even-loop (list 1 2 3 4))
- : (Listof Nonnegative-Integer)
'(4 2)
```

```
(plet: (a ...) ([v : t e] ...) . body)
```

A polymorphic version of `let:`, abstracted over the type variables `a`. The type variables `a` are bound in both the types of the formal, and in any type expressions in the *body*. Does not support the looping form of let.

```
(letrec: ([v : t e] ...) . body)
(let*: ([v : t e] ...) . body)
(let-values: ([([v : t] ...) e] ...) . body)
(letrec-values: ([([v : t] ...) e] ...) . body)
(let*-values: ([([v : t] ...) e] ...) . body)
```

Type-annotated versions of `letrec`, `let*`, `let-values`, `letrec-values`, and `let*-values`. As with `let:`, type annotations are optional.

```
(let/cc: v : t . body)
(let/ec: v : t . body)
```

Type-annotated versions of `let/cc` and `let/ec`.

```
(define: v : t e)
(define: (a ...) v : t e)
(define: (a ... a ooo) v : t e)
(define: (f . formals) : t . body)
(define: (a ...) (f . formals) : t . body)
(define: (a ... a ooo) (f . formals) : t . body)
```

These forms define variables, with annotated types. The first form defines `v` with type `t` and value `e`. The second form does the same, but allows the specification of type variables. The third allows for polydotted variables. The fourth, fifth, and sixth forms define a function `f` with appropriate types. In most cases, use of `:` is preferred to use of `define:`.

Examples:

```
> (define: foo : Integer 10)

> (define: (A) mt-seq : (Sequenceof A) empty-sequence)

> (define: (add [first : Integer]
                [rest  : Integer]) : Integer
    (+ first rest))

> (define: (A) (poly-app [func : (A A -> A)]
                         [first : A]
                         [rest  : A]) : A
    (func first rest))
```

```
struct:
```

An alias for `struct`.

```
define-struct:
```

An alias for `define-struct`.

```
define-struct/exec:
```

An alias for `define-struct/exec`.

```
for:
```

An alias for `for`.

```
for*/and:
for*/first:
for*/flvector:
for*/fold:
for*/hash:
for*/hasheq:
for*/hasheqv:
for*/last:
for*/list:
for*/lists:
for*/or:
for*/product:
for*/sum:
for*/vector:
for*:
for/and:
for/first:
for/flvector:
for/fold:
for/hash:
for/hasheq:
for/hasheqv:
for/last:
for/list:
for/lists:
for/or:
for/product:
for/sum:
for/vector:
```

Aliases for the same iteration forms without a `:`.

`do:`

An alias for `do`.

`define-type-alias`

Equivalent to `define-type`.

`define-typed-struct`

Equivalent to `define-struct:`

`require/opaque-type`

Similar to using the `opaque` keyword with `require/typed`.

`require-typed-struct`

Similar to using the `struct` keyword with `require/typed`.

`require-typed-struct/provide`

Similar to `require-typed-struct`, but also provides the imported identifiers.

`pdefine:`

Defines a polymorphic function.

`(pred t)`

Equivalent to `(Any -> Boolean : t)`.

`Un`

An alias for `U`.

`mu`

An alias for `Rec`.

`Tuple`

An alias for `List`.

`Parameter`

An alias for `Parameterof`.

`Pair`

An alias for `Pairof`.

`values`

An alias for `Values`.

11 Compatibility Languages

`#lang typed/scheme` package: `typed-racket-compatibility`
`#lang typed/scheme/base`
`#lang typed-scheme`

Typed versions of the

```
#lang scheme
```

and

```
#lang scheme/base
```

languages. The

```
#lang typed-scheme
```

language is equivalent to the

```
#lang typed/scheme/base
```

language.

```
(require/typed m rt-clause ...)

    rt-clause = [r t]
              | [struct name ([f : t] ...)
                    struct-option ...]
              | [struct (name parent) ([f : t] ...)
                    struct-option ...]
              | [opaque t pred]

struct-option = #:constructor-name constructor-id
              | #:extra-constructor-name constructor-id
```

Similar to `require/typed`, but as if `#:extra-constructor-name make-name` was supplied.

```
require-typed-struct
```

Similar to using the `struct` keyword with `require/typed`.

12 Experimental Features

These features are currently experimental and subject to change.

```
(declare-refinement id)
```

Declares *id* to be usable in refinement types.

```
(Refinement id)
```

Includes values that have been tested with the predicate *id*, which must have been specified with `declare-refinement`.

```
(define-typed-struct/exec forms ...)
```

Defines an executable structure.