Conspiracy Library

May 31, 2022

Conspiracy is an object-oriented paradigm. An object is defined as a collection of property ids and value pairs (i.e. properties) associated with a symbol name registered in the objects database.

1 Getting Started

1.1 Object Modules

The following documents the modules that can be *required* as part of the *Object* Library

1.1.1 Main Module

For the purpose of this documentation it is assumed that the library has been made into a collection. We will be using the registration syntax provided by this module.

The 'main' module must always be required in using any Conspiracy Object module. It provides the database and various functions as well as `Object` itself.

```
> (require conspiracy/object)
'Object
```

The first thing to notice is that requiring the object module returns `Object`. That's the symbol representing the 'root' object in the database.

1.1.2 `%object` Module

This module provides a traditional version of registration syntax.

- (`object` ...) can be used instead of (`%` ...) for object definition.
- (`replace-object` ...) can be used instead of (`%=` for object replacement.
- (`modify-object` ...) can be used instead of (`%+` ...) for object modification.
- (`remove-object` ...) can be used instead of (`%-` ...) for object removal.

```
> (require conspiracy/object
           (submod conspiracy/object %object))
'Object
```

1.1.3 `@app` Module

This module will map any form beginning with an unbound operator to the `@` operation. Any bound operator will be executed normally.

```
> (require conspiracy/object
           (submod conspiracy/object @"@"app))
'Object
```

Examples:

```
> (list a b c)
'(a b c)
> (of-kind Object Object)
#<undefined>
> (of-kind? Object Object)
#t
```

1.1.4 `Nothing` Module

This module defines a basic 'object tree' mechanism.

```
> (require conspiracy/object
           (submod conspiracy/object Nothing))
'Object
'ε-Nothing
'Nothing
```

1.1.5 `Rule` Module

This module defines a basic 'Rules' mechanism.

```
> (require conspiracy/object
           (submod conspiracy/object Rule))
'Object
'Rule
```

1.2 Automatic Quoting of Unbound Symbols

Conspiracy doesn't bind the symbols you choose for object or property names. It adopts the position of automatically quoting unbound symbols. While in 'normal' Racket you would receive an error message, in Conspiracy the unbound symbol is bounced back.

2 The Database

Although its been said that objects can be represented as nothing more than structs with associated functions, Conspiracy takes a somewhat different approach of defining them as keys in a hash table whose values are hash tables of property keys and their values.

As mentioned in the previous section on the quoting of unbound symbols, object and property names needn't have meaning outside the objects database.

```
> Object
'Object
```

On the other hand, it's perfectly possible to define an object named 'pi' or 'λ' in the objects database with meanings independent of their normal Racket bound values. In those cases, however, it will be important to remember to quote the symbol names to avoid operating on their bound values, rather than on their symbols.

But the practical upshot of this is that neither objects, nor properties, are first class values. You can't pass an object into a function, because it doesn't live outside of the database.

2.1 `object-names`

At any point the object names registered in the objects database can be retrieved with the object-names macro like this:

```
> object-names
'(Object)
```

Although it's a list of object names, the list is constructed in arbitrary order and doesn't represent a historic perspective of object definition.

2.2 `objects`

The `objects` macro provides a 'raw' view of the objects database. By using it, you can glimpse 'under the hood' or 'behind the curtain' at the hash tables that comprise it.

```
> objects
'#hasheq((Object
          .
          #hasheq((assert . #<procedure:...nspiracy/object.rkt:270:42>)
                  (assert! . #<procedure:...nspiracy/object.rkt:270:42>)
                  ...
                  (set-method! . #<procedure:[...nspiracy/object.rkt:270:42>)
                  (show . #<procedure:...onspiracy/object.rkt:270:42>))))
```

We've abbreviated the properties listing for Object. But you can clearly see the key/value pairs of the hash tables involved, both of which use symbols for keys.

So far we only have `Object` registered in the database. But before we move on to defining, we need to discuss messaging in the *messaging* section.

2.3 Filtering

As stated earlier, database object and property symbols are not bound by the library. This means you have to pass the database and its macro and function operators to modules that want to make use of Conspiracy objects.

Sometimes you might want a subset of the database instead. Conspiracy provides 2 forms that parameterize the database allowing you to filter it at the same time.

The following macros parameterize on:

- objects

- default-kind
- debug

Parameterizing these values means that within the context of the `with-objects` and `without-objects` macro bodies those values can be modified, but will be restored to their original values once control returns outside of the macros.

2.3.1 with-objects

Filter objects by kinds and their *inherited* objects.

Syntax:

(**with-objects** (*kind* ...) body ...)

Examples:

```
> object-names
'(Object)
> (with-objects () (% A) (% B) object-names)
'(B A Object)
> object-names
'(Object)
> (% A)
'A
> (% B)
'B
> (% C (kinds A B))
'C
> (with-objects (A) object-names)
'(A Object)
> (% D (kinds C))
'D
> (with-objects (C) object-names)
'(C B A Object)
> object-names
'(C B D A Object)
```

2.3.2 without-objects

Filter-not objects by kinds and their *modified* and *inheriting* objects.

Syntax:

(**without-objects** (*kind* ...) body ...)

Examples:

```
          > object-names
'(Object)
> (% A)
'A
> (%+ A (p0 14))
'A
```

```
> (% B)
'B
> (% C (kinds A))
'C
> object-names
'(B mod#5944484 C A Object)
> (without-objects (A) object-names)
'(B Object)
> object-names
'(B mod#5944484 C A Object)
```

3 Messaging Objects

Once an object has been registered in the database it can be interacted with by use of the following property messages:

- Retrieval
- Application
- Setting

With small variations the syntax of Conspiracy messaging is the same for each message type.

(*op propname objname* [*arg* ...] [`#:when-undefined`] [`#:assert`])

```
op               : (or/c ? @"@" !)
propname         : symbol?
objname          : object?
arg              : any/c
#:when-undefined : any/c
#:assert         : contract?
```

A Conspiracy method can return any number of values:

- Zero : (values)
- Single : (values any/c)
- Multiple: (values any/c any/c ...)

3.1 Retrieval

This message retrieves the value of the property for the object.

This message does not accept arguments, but `#:when-undefined` and `#:assert` are both valid with the message.

`#:when-undefined`: This can be any value that then becomes the result of the message when the message returns `undefined`. This can occur if the property was assigned that value, or if the property was not directly-defined or inherited by this object.

#:assert: This can be any contract that is then applied to the result of the message (after the optional #:when-undefined value was applied) to assert some condition on the result. If the condition is true the result is returned by the message, otherwise the message throws an error.

Examples:

```
          > (? show Object)
#<procedure:...onspiracy/object.rkt:270:42>
> (? foo Object)
#<undefined>
> (? foo Object #:when-undefined 42)
42
> (? foo Object #:when-undefined 42 #:assert string?)
 ...onspiracy/object.rkt:774:4: broke its own contract
  promised: string?
  produced: 42
  in: string?
  contract from: Object.foo
  blaming: Object.foo
   (assuming the contract is correct)
```

3.2 Application

This message applies the value of the property for the object to the arguments.

This message only enforces the arity of arguments when the value is a procedure, otherwise it ignores the arguments altogether.

#:when-undefined is valid with the message. #:assert is not valid.

Examples:

```
> (@ get-kind-order Object)
'(Object)
> (@ get-prop-params Object show)
'(3 () () #f)
```

3.3 Setting

This message sets the value of the property for a mutable object.

Examples:

```
          > (! kinds Object '(Object))
 !: contract violation
  expected: mutable-object?
  given: 'Object
  in: the 2nd argument of
      (-> symbol? mutable-object? any/c any)
  contract from: (function !)
  blaming: <pkgs>/conspiracy/object.rkt
   (assuming the contract is correct)
  at: <pkgs>/conspiracy/object.rkt:784.18
> (! foo Object #t)
 !: contract violation
  expected: mutable-object?
  given: 'Object
  in: the 2nd argument of
      (-> symbol? mutable-object? any/c any)
  contract from: (function !)
  blaming: <pkgs>/conspiracy/object.rkt
   (assuming the contract is correct)
  at: <pkgs>/conspiracy/object.rkt:784.18
```

3.4 Debugging

By default the debug parameter of the library is set to #f. At times you may want to set debugging on, which will display library variables during messaging.

To get and set debug, use `debug` and `debug!`.

Syntax:

debug

(**debug!** [*value*])

value : `boolean?`

Examples:

```
> debug
#f
> (debug!)
> debug
#t
> (debug! #t)
> debug
#t
> (debug! #f)
> debug
#f
```

With debug on you'll see something like the following (the meanings of which will be explained in later sections.)

```
          > (% A (p0 42))
[@: construct A ()]
[construct:#<undefined> | target=A self=A | args=#<undefined>]
[construct:#<undefined> | target=A self=A | args=#<undefined>]
[Parameters:]
        self: A
        target-prop: construct
        target-obj: A
        kind-order: (A Object)
        defining-obj: Object
        invokee: #<procedure:...nspiracy/object.rkt:260:42>
        invokee-arity: #(struct:arity-at-least 0)
[construct:Object | target=A self=A | args=()]
[@: flags? A (no-assert)]
[flags?:Object | target=A self=A | args=()]
[flags?:Object | target=A self=A | args=()]
[Parameters:]
        self: A
        target-prop: flags?
        target-obj: A
        kind-order: (A Object)
        defining-obj: Object
        invokee: #<procedure:...onspiracy/object.rkt:260:42>
        invokee-arity: #(struct:arity-at-least 0)
[flags?:Object | target=A self=A | args=(no-assert)]
[@: prop-defined A (flags prop-def-directly)]
[prop-defined:Object | target=A self=A | args=(no-assert)]
[prop-defined:Object | target=A self=A | args=(no-assert)]
[Parameters:]
        self: A
        target-prop: prop-defined
        target-obj: A
        kind-order: (A Object)
        defining-obj: Object
        invokee: #<procedure:...onspiracy/object.rkt:260:42>
        invokee-arity: (1 2)
[prop-defined:Object | target=A self=A | args=(flags prop-def-directly)]
[@: assert! A ()]
[assert!:Object | target=A self=A | args=()]
[assert!:Object | target=A self=A | args=()]
[Parameters:]
        self: A
        target-prop: assert!
        target-obj: A
        kind-order: (A Object)
        defining-obj: Object
        invokee: #<procedure:...nspiracy/object.rkt:260:42>
        invokee-arity: 0
[assert!:Object | target=A self=A | args=()]
[@: ntf-reqs A ()]
[ntf-reqs:Object | target=A self=A | args=()]
[ntf-reqs:Object | target=A self=A | args=()]
[Parameters:]
        self: A
        target-prop: ntf-reqs
        target-obj: A
        kind-order: (A Object)
        defining-obj: Object
        invokee: #<procedure:...nspiracy/object.rkt:260:42>
        invokee-arity: 0
[ntf-reqs:Object | target=A self=A | args=()]
[@: prop-defined A (implements prop-def-directly)]
[prop-defined:Object | target=A self=A | args=()]
[prop-defined:Object | target=A self=A | args=()]
[Parameters:]
        self: A
        target-prop: prop-defined
        target-obj: A
        kind-order: (A Object)
        defining-obj: Object
        invokee: #<procedure:...onspiracy/object.rkt:260:42>
        invokee-arity: (1 2)
[prop-defined:Object | target=A self=A | args=(implements prop-def-
directly)]
[@: prop-defined Object (implements prop-def-directly)]
[prop-defined:Object | target=Object self=Object | args=()]
[prop-defined:Object | target=Object self=Object | args=()]
[Parameters:]
        self: Object
        target-prop: prop-defined
        target-obj: Object
        kind-order: (Object)
        defining-obj: Object
        invokee: #<procedure:...onspiracy/object.rkt:260:42>
        invokee-arity: (1 2)
[prop-defined:Object | target=Object self=Object | args=(implements
prop-def-directly)]
[@: assert A ()]
[assert:Object | target=A self=A | args=()]
[assert:Object | target=A self=A | args=()]
[Parameters:]
        self: A
        target-prop: assert
        target-obj: A
        kind-order: (A Object)
        defining-obj: Object
        invokee: #<procedure:...nspiracy/object.rkt:260:42>
        invokee-arity: 0
[assert:Object | target=A self=A | args=()]
[@: ntf-reqs A ()]
[ntf-reqs:Object | target=A self=A | args=()]
[ntf-reqs:Object | target=A self=A | args=()]
```

```
[Parameters:]
        self: A
        target-prop: ntf-reqs
        target-obj: A
        kind-order: (A Object)
        defining-obj: Object
        invokee: #<procedure:...nspiracy/object.rkt:260:42>
        invokee-arity: 0
[ntf-reqs:Object | target=A self=A | args=()]
[@: prop-defined A (implements prop-def-directly)]
[prop-defined:Object | target=A self=A | args=()]
[prop-defined:Object | target=A self=A | args=()]
[Parameters:]
        self: A
        target-prop: prop-defined
        target-obj: A
        kind-order: (A Object)
        defining-obj: Object
        invokee: #<procedure:...onspiracy/object.rkt:260:42>
        invokee-arity: (1 2)
[prop-defined:Object | target=A self=A | args=(implements prop-def-
directly)]
[@: prop-defined Object (implements prop-def-directly)]
[prop-defined:Object | target=Object self=Object | args=()]
[prop-defined:Object | target=Object self=Object | args=()]
[Parameters:]
        self: Object
        target-prop: prop-defined
        target-obj: Object
        kind-order: (Object)
        defining-obj: Object
        invokee: #<procedure:...onspiracy/object.rkt:260:42>
        invokee-arity: (1 2)
[prop-defined:Object | target=Object self=Object | args=(implements
prop-def-directly)]
'A
> (? p0 A)
[?: p0 A ()]
[p0:#<undefined> | target=A self=A | args=#<undefined>]
[p0:#<undefined> | target=A self=A | args=#<undefined>]
42
```

4 Registering Objects

Initially the objects database only contains `Object`. In this section we briefly go over the ways that the database can be changed through four forms of registration

4.1 Object Definition

Registers an object in the objects database. Returns the object name.

Syntax:

```
(% [objname] property ...)
  objname  : (and/c symbol? (not object?))
  property : (flags flag [flag ...])
           | (kinds kind [kind ...])
           | (implements ntf [ntf ...])
           | (propname argspec ctcspec body0 body ...)
           | (propname (value ...))
           | (propname value)
  flag      : symbol?
  kind      : object?
  ntf       : object?
  propname  : symbol?
  argspec   : Racket λ kw-formal
  ctcspec   : Racket function contract
```

When *objname* is not provided the library will generate a name from a gensym prefixed with 'obj# that is then interned for referencing.

Properties will be discussed further in separate sections.

Examples:

```
> (%)
'obj#2125281
> (% A (sqr (n) (-> integer? integer?) (sqr n)))
'A
> (@ sqr A 25)
 Library/Mobile Documents/com~apple~CloudDocs/Racket/utils/conspiracy/object.rkt:801:5:
@: contract violation
  expected: symbol?
  given: #<procedure:sqr>
> (@ 'sqr A 25)
625
> (@ show A)
(% A
        (kinds (Object))
        (sqr #<procedure:...nspiracy/object.rkt:270:42>))
```

4.2 Anonymous Definition

Registers an 'anonymous' object in the objects database. Returns the object name. This is an alternative form of the `%` 'anonymous' definition above.

4.2.1 `%*`

Syntax:

```
(%* kind ... property ...)
  kind      : object?
  property  : (flags flag [flag ...])
            | (implements ntf [ntf ...])
            | (propname argspec ctcspec body0 body ...)
            | (propname (value ...))
            | (propname value)
  flag      : symbol?
  ntf       : object?
  propname  : symbol?
  argspec   : Racket λ kw-formal
  ctcspec   : Racket function contract
```

This syntax is a shorter version of the anonymous variation of `%`, but the results are the same. Note that `kinds` is not a valid property in this syntax.

Examples:

```
> (% A (p0 0))
'A
> (% B (p1 10))
'B
> (%* A B (p2 20))
'obj#26607112
> (@ show obj#26607112 #:flag +kinds)
(% obj#26607112
        (kinds (A B))
        (p2 20))
(% A
        (kinds (Object))
        (p0 0))
(% B
        (kinds (Object))
        (p1 10))
> (%*)
'obj#26628183
> (@ show obj#26628183)
(% obj#26628183
        (kinds (Object)))
> (%* B A (p42 42) (kinds A B))
obj#26671717: kinds property not valid in definition.
```

```
> (%* B A (p42 42))
'obj#26672992
> (@ show obj#26672992 +kinds)
(% obj#26672992
        (kinds (B A))
        (p42 42))
(% B
        (kinds (Object))
        (p1 10))
(% A
```

4.2.2 τ

Anonymous object syntax especially useful for objects designed to behave as 'templates'.

Syntax:

(τ *kind* [*kind* ...] property ...)

kind = object?

property = *propname*: val [val ...]

propname : symbol?

Note:

- property names are identified by the *:* suffix.
- `kinds` is not a valid property in this form.
- A single value will produce a propname/value pair. Any properties requiring a list will neeed to have the value wrapped in a list function. This applies to special properties `flags` and `implements` as well.

Examples:

```
> (% A)
'A
> (@ show A)
(% A
        (kinds (Object)))
> (τ A)
'obj#15994270
> (@ show obj#15994270)
(% obj#15994270
        (kinds (A)))
> (τ A p0: foo p1: bar baz p2: x y z)
'obj#16020613
> (@ show obj#16020613)
(% obj#16020613
        (kinds (A))
        (p0 foo)
        (p1 (bar baz))
        (p2 (x y z)))
> (τ A p0: (list foo))
'obj#16030671
> (@ show obj#16030671)
(% obj#16030671
        (kinds (A))
        (p0 (foo)))
```

4.3 Object Replacement

Replaces an object in the objects database.

Syntax:

```
(%= objname property ...)
  objname  : object?
  property : (flags flag [flag ...])
           | (kinds kind [kind ...])
           | (implements ntf [ntf ...])
           | (propname argspec ctcspec body0 body ...)
           | (propname (value ...))
           | (propname value)
  flag     : symbol?
  kind     : object?
  ntf      : object?
  propname : symbol?
  argspec  : Racket λ kw-formal
  ctcspec  : Racket function contract
```

Properties will be discussed further in separate sections.

Examples:

```
> (% A)
'A
> (@ show A)
(% A
        (kinds (Object)))
> (%= A (p0 42))
'A
> (@ show A)
(% A
        (kinds (Object))
        (p0 42))
> (% A (flags immutable))
'A
> (! p0 A 10)
 !: contract violation
  expected: mutable-object?
  given: 'A
  in: the 2nd argument of
      (-> symbol? mutable-object? any/c any)
  contract from: (function !)
  blaming: <pkgs>/conspiracy/object.rkt
   (assuming the contract is correct)
  at: <pkgs>/conspiracy/object.rkt:784.18
> (%=  A (p0 42))
'A
> (! p0 A 10)
> (@ show A)
(% A
        (kinds (Object))
        (p0 10))
```

4.4 Object Modification

Modifies an object in the objects database.

Syntax:

```
(%+ objname property [property ...])
  objname  : object?
  property : (flags flag [flag ...])
           | (kinds kind [kind ...])
           | (implements ntf [ntf ...])
           |(remove propname [propname ...])
           |(replace property [property ...])
           | (propname argspec ctcspec body0 body ...)
           | (propname (value ...))
           | (propname value)
  flag     : symbol?
  kind     : object?
  ntf      : object?
  propname : symbol?
  argspec  : Racket λ kw-formal
  ctcspec  : Racket function contract
```

Properties will be discussed further in separate sections.

Modification is a registration process that is more complicated than replacement. During modification the following steps occur:

- The directly-defined properties of the object are registered as a new *modobject* whose object name is a gensym prefixed by 'mod# and then interned for referencing.
- The property clauses of %+ become the object's directly-defined property list, with the addition of a `kinds` property pointing to the *modobject*.
- Any `replace` property values are removed from the *modobject* and added to the object's directly-defined property list.

- Any `remove` property values are removed from the object and its modifications.
- The object's `construct` property is called.

```
          > (%+ Object (p0 42))
object-modify: contract violation
 expected: mutable-object?
 given: 'Object
 in: the 1st argument of
     (->*
      (mutable-object?
       (non-empty-listof
        (cons/c
         (and/c symbol? (not/c 'kinds))
         any/c)))
      (#:construct? boolean?)
      symbol?)
 contract from: (function object-modify)
 blaming: <pkgs>/conspiracy/object.rkt
  (assuming the contract is correct)
 at: <pkgs>/conspiracy/object.rkt:449.18
```

4.4.1 Replacing Properties

When it is desirable to remove a property from an object's modifications, providing a new version in its directly-defined properties we can accomplish this with the `replace` property. The syntax of which is:

(**replace** *property* [*property* ...])

As we'll see later in discussions on inheritance, this will allow a method to pass control along the object's original `kind-order`.

Examples:

```
> (% A (p0 0))
'A
> (%+ A (p0 1))
'A
> (%+ A (p0 2))
'A
> (%+ A (replace (p0 3)))
'A
> (@ show A #:flag +mods)
(% A
        (kinds (mod#2864351))
        (p0 3))
(% mod#2864351
        (kinds (mod#2864205)))
(% mod#2864205
        (kinds (mod#2864132)))
(% mod#2864132
        (kinds (Object)))
```

4.4.2 Removing Properties

In object modification a property can be removed from its modifications using a property with the following syntax:

(**remove** *propname* [*propname* ...])

Examples:

```
> (% A (p0 0))
'A
> (%+ A (p1 1))
'A
> (%+ A (p2 2))
'A
> (%+ A (remove p0))
'A
> (@ show A #:flag +mods)
(% A
        (kinds (mod#2907069)))
(% mod#2907069
        (kinds (mod#2906919))
        (p2 2))
(% mod#2906919
        (kinds (mod#2906855))
        (p1 1))
(% mod#2906855
        (kinds (Object)))
```

4.5 Object Removal

Removes an object and its derivations and modifications from the objects database. Returns a list of all objects removed.

Syntax:

(**%-** *objname*)

objname : object?

Examples:

```
> (% A)
'A
> (%+ A (p0 42))
'A
> (% B (kinds A) (p0 100))
'B
> (%- A)
'(B A mod#2493110)
```

5 Special Properties

Conspiracy properties are pairs of property name and value, but certain property names are 'special' in that they play a role in registration, inheritance, mutability, undefined property handling, and interfaces.

5.1 Flags

The `flags` property has special usage within the library, as will be discussed below, and also has an unusual property syntax.

Syntax:

(**flags** *flag* [*flag* ...])
flag : `symbol?`

The property's value is a list, and it can be defined as above, or in the usual ways that lists are represented:

```
> (% A (flags x y z))
'A
> (% B (flags '(x y z)))
'B
> (% C (flags (x y z)))
'C
> (% D (flags (list x y z)))
'D
> (@ show A)
(% A
        (flags (x y z))
        (kinds (Object)))
> (@ show B)
(% B
        (flags (x y z))
        (kinds (Object)))
> (@ show C)
(% C
        (flags (x y z))
        (kinds (Object)))
> (@ show D)
(% D
        (flags (x y z))
        (kinds (Object)))
```

5.1.1 Interrogating Flags

The flags mechanism in Conspiracy can be thought of as asking a question pertaining to the object. The flag is only relevant if the object directly-defines it. Flag interrogation is not inherited. Flags can be thought of as boolean values: their presence indicating true, their absence false.

The object message `flags?` is used to interrogate the `flags` property of an object. This message constrains the values to the directly-defined properties of the object, ignoring whether the object might inherit the flags property from other objects.

syntax

(|@ **flags?** *objname* [#:al] *flag* [*flag* ...])
objname : object?
#:all : boolean?
flag : symbol?

The `#:all` boolean is #t by default. In that case the object must have all of the flags set in order for `flags?` to return true. When `#:all` is #f any flag match will cause `flags?` to return #t.

Examples:

```
> (% A (flags immutable x y z))
'A
> (@ flags? A immutable)
#t
> (@ flags? A w x y)
#f
> (@ flags? A #:all #f w x y)
#t
> (% B (kinds A))
'B
> (@ flags? B immutable)
#f
```

5.1.2 Library Flags

While any symbol can serve as a flag (even an object name), the library makes use of a handful of symbols for answering questions like:

- Is the object immutable?
- Does the object want to handle undefined properties itself?
- Does the object want to bypass asserting an interface it implements?
- Should any object that `implements` this object treat it as an μ-props rather than an ι-props?

`immutable`

By default Conspiracy objects are mutable.

Immutable objects cannot have their property values set with the `!` operator.

Objects are defined immutable by registering them with an `immutable` flag. Once this is done their *directly-defined* property values cannot be modified (unless they are of a datatype that is itself mutable and referenced elsewhere.)

`no-assert`

By default an object that `implements` an interface asserts that the requirements of that interface during its construction. But there are times when you may wish to define an object that implements an interface which is only asserted by its derivations. In this manner you can simulate a template or class behavior.

By giving the object the `no-assert` flag you tell the library that it must not enforce any directly-defined or inherited ι-`props` or μ-`props` interfaces.

Examples:

```
> (% A (p0 number?))
'A
> (% B (implements A))
B does not implement A ι-props property (? p0 B) -> #<procedure:number?>.
> (% B (flags no-assert)
     (implements A))
'B

> (@ show B)
(% B
        (flags (no-assert))
        (implements (A))
        (kinds (Object)))
> (% C (kinds B) (p0 foo))
C does not implement A ι-props property p0 ->:
        Expected: #<procedure:number?>
        Actual: foo
> (% C (kinds B)
     (p0 42))
'C
> (@ show C)
(% C
        (kinds (B))
        (p0 42))
```

B is not a template-object, which requires the `immutable` flag as well as `no-assert`.

`prop-not-defined`

The flag value `prop-not-defined` tells the library to redirect any application operation to the object's `prop-not-defined` method. This method can be directly-defined or inherited by the object, but the redirecting of undefined messages applies only when the object for whom the message is undefined provides the flag.

Examples:

```
> (% B)
'B
> (@ foo B)
#<undefined>
> (%+ B (flags prop-not-defined)
     (prop-not-defined vals (->* () #:rest list? any) #f))
'B
> (@ foo B)
#f
```

In the example above, B does not define or inherit a *foo* property. The library default returns `undefined` in that situation. We could use the #:when-undefined keyword on the `@` call, but in this case the object has been modified to handle undefined properties itself.

`ε-props`

An object can be designated an 'ε-props' with the `ε-props` flag (ε-props will be discussed in further detail in the `implements` property section.)

Examples:

```
> (% A (flags ε-props) (p0 undefined))
'A
> (% B (implements A) (p0 42))
B must not define ε-props A property p0
```

`ι-props`

An object can be designated an ι-props interface with the `ι-props` flag (ι-props will be discussed in further detail in the `implements` property section.

`μ-props`

An object can be designated an μ-props interface with the `μ-props` flag (μ-props will be discussed in further detail in the `implements` property section.)

Examples:

```
> (% A (flags μ-props)
     (p0 42))
'A
> (% B (implements A))
'B
> (@ show B)
(% B
        (implements (A))
        (kinds (Object))
        (p0 42))
```

5.2 Kinds

The `kinds` property has special usage within the library, as will be discussed below, and also has an unusual property syntax.

Syntax:

(**kinds** *kind* [*kind* ...])

kind : `object?`

The property's value is a list, and it can be defined as above, or in the usual ways that lists are represented:

```
> (% A)
'A
> (% B)
'B
> (% C (kinds A B))
'C
> (% D (kinds '(B A)))
'D
> (@ show C)
```

```
(% C
        (kinds (A B)))
> (@ show D)
(% D
        (kinds (B A)))
> (% E (kinds (C D)))
'E
> (@ show E)
(% E
        (kinds (C D)))
> (% F (kinds (list D E)))
'F
> (@ show F)
(% F
        (kinds (D E)))
```

In Conspiracy an object represents a 'kind' of thing. The `kinds` property is perhaps one of the most important that can be defined for an object as it determines from which existing objects the new one will inherit.

5.2.1 `object?`

The library defines the `object?` predicate as any symbol registered as an object name in the objects database.

```
> (% A (kinds B))
 Library/Mobile Documents/com~apple~CloudDocs/Racket/utils/conspiracy/object.rkt:384:11:
make-object: contract violation
  expected: (or/c (cons/c (quote kinds) (non-empty-listof (or/c (quote
Object) (and/c (not/c (quote A)) object?)))) (cons/c (and/c symbol?
(not/c (quote kinds))) any/c))
  given: '(kinds B)
  in: an element of
      the ps argument of
      (->i
       ((name
         (or/c
          #f
          (and/c symbol? (not/c object?))))
        (ps
         (name)
         (listof
          (or/c
           (cons/c 'kinds (non-empty-listof ...))
           (cons/c (and/c symbol? ...) any/c)))))
       (#:construct? (call () boolean?))
       (result symbol?))
  contract from: (function make-object)
  blaming: <pkgs>/conspiracy/object.rkt
   (assuming the contract is correct)
  at: <pkgs>/conspiracy/object.rkt:397.18
```

Initially the library bootstraps `Object` as a valid *kind* as well.

```
> (% A)
'A
> (object? A)
#t
> (object? B)
#f
```

5.2.2 `default-kind`

Objects that do not specify a `kinds` property in their definition are provided with one that uses the object(s) specified by the default-kind macro. Initially this value is set to 'Object, but it can be changed using the `default-kind!` macro.

Examples

```
> default-kind
'Object
> (% A)
'A
> (% B)
'B
> (default-kind! A B)
> (% C)
'C
> (@ show A)
(% A
        (kinds (Object)))
> (@ show B)
(% B
        (kinds (Object)))
> (@ show C)
(% C
        (kinds (A B)))
> default-kind
'(A B)
```

5.2.3 Inheritance Order

The ordering of `kinds`, like the naming of cats, is very important, as we'll learn in the section on inheritance. In the examples above the `kinds` property for C shows the order '(A B), which would be very different in terms of inheritance from '(B A). More on that later.

5.3 implements

The `implements` property allows an object to implement an interface: ι-`props`, μ-`props` or ε-`props`.

5.3.1 Implementing Interface Objects

The following should be kept in mind:

- Interface assertion occurs at object registration. Objects are used to specify the interface. Any changes to an interface object after the object's registration do not affect the implementing object.
- If an error occurs in the assertion of an interface during object registration, the object definition is rolled back.
- an object definition can choose not to assert ι-`props` and μ-`props` interfaces during its construction using the `no-assert` flag.
- Interfaces are inherited. An object's inheritance of interfaces follows its `kind-order`
- An implementation can be removed from an object's inheritance through use of the ~ prefixed to the interface object name.

Examples:

```
> (% A (flags μ-props)
     (p0 14))
'A
> (% B (implements A) (flags no-assert))
'B
> (@ show B)
(% B
        (flags (no-assert))
        (implements (A))
        (kinds (Object)))
> (% C (kinds B))
'C
> (@ show C)
(% C
        (kinds (B))
        (p0 14))
> (% D (kinds C) (implements ~A))
'D
> (@ show D)
(% D
        (implements (~A))
        (kinds (C)))
> (% E (kinds D))
```

```
'E
> (@ show E)
(% E
        (kinds (D)))
```

Object's implementing ι-`props` and μ-`props` interfaces can choose to not assert the interface at the time of their construction.

- μ-`props-assert!` is applied first to provide the object a chance to populate its directly-defined property list with required properties.
- (ι-`props-assert` is applied after μ-`props-assert!` to ensure the object provides the required properties.

5.3.2 Implementing ε-props objects

Sometimes it is desirable that an object *not* define certain properties. This is useful when the object serves as a kind of 'template', where the ε-`props` interfaces ensure the object will not define properties at registration that will later be overridden by object construction.

'Existential properties' interfaces are asserted after the object is registered, but before its `construct` is invoked. If the object directly defines the properties of the ε-`props` interface, then an error is thrown.

The ε-`props` flag can be removed from an implementation chain with the ~ prefixed to the *ntf* object name.

Examples:

```
> (% A (flags ε-props) (p0 undefined))
'A
> (% B (implements A))
'B
> (% C (kinds B) (p0 42))
C must not define ε-props A property p0
> (% C (kinds B) (p0 42) (implements ~A))
'C
```

6 Special Property Values

In Conspiracy all properties consist of pairs of property name and property value. The previous section discussed the library's unique meanings for the property names: `flags`, `implements`, and `kinds`. But it handles certain values as special-case syntax.

6.1 Methods

A method is a procedure, represented by a property, specific to an object, that can be applied to arguments. To do this we use the `@` operator. When this is done the 'arity' of the function, the number of parameters it requires must agree with the number of arguments it receives.

Earlier we discussed how the `?` operator could *assert* values on the results of its function. The `@` operator lacks that feature because all procedures defined using the object method syntax require function contracts.

This requirement of the method syntax is easy to forget, but its usefulness is that it ensures the method has maximum control over the data it receives and the data it produces.

Syntax:

```
(propname argspec ctcspec body0 body ...)
  propname : symbol?
  argspec  : Racket λ kw-formal
  ctcspec  : Racket function contract
```

In other words, the property is defined just as though it were a Racket function being defined with the define/contract form.

Examples:

```
> (% A (double (n) (-> number? number?) (* 2 n)))
'A
> (@ double A 13)
26
> (% B (add vs (->* () #:rest (listof number?) number?) (apply + vs)))
'B
> (@ add B 1 2 3)
6
```

6.2 Nested Objects

A property value can be an 'anonymous' object definition. Recall that an anonymous object is one for which the library provides the object name in the form of a gensym prefixed by *'obj#* and then interned for referencing.

The nested object can define any of the normal object properties, but gets a special `lexical-parent` property added in its directly-defined properties that points to the object it is nested in.

Examples:

```
> (% (p0 (% (p0 (%)))))
'obj#5302530
> (@ show obj#5302530)
(% obj#5302530
        (kinds (Object))
        (p0 obj#5302531))
> (@ show obj#5302531)
(% obj#5302531
        (kinds (Object))
        (lexical-parent obj#5302530)
        (p0 obj#5302532))
> (@ show obj#5302532)
(% obj#5302532
        (kinds (Object))
        (lexical-parent obj#5302531))
```

6.3 Lambda Forms

The lambda function definition is a valid object property value.

Examples:

```
> (% A (p0 (λ v v)))
'A
> (@ show A)
(% A
        (kinds (Object))
        (p0 #<procedure:...nspiracy/object.rkt:275:32>))
> ((? p0 A) "Hello, World!")
'("Hello, World!")
```

6.4 Quoted and Quasiquoted Values

You can quote and quasiquote the property value in the normal Racket way.

Examples:

```
> (% A
     (p0 'pi)
     (p1 `(x y ,pi)))
'A
> (@ show A)
(% A
        (kinds (Object))
        (p0 pi)
        (p1 (x y 3.141592653589793)))
```

6.5 *(...)* Values

This value special form behaves as follows:

- When empty it represents the empty list.
- When the first element is not a procedure it produces a list of the values.
- When the first element is a procedure, it is treated as a procedure call.

Examples:

```
> (% A
     (p0 (+ 1 2 3))
     (p1 ('+ x y z))
     (p2 ((λ x x) 1 2 3)))
'A
> (@ show A)
(% A
        (kinds (Object))
        (p0 6)
        (p1 (+ x y z))
        (p2 (1 2 3)))
```

7 Pseudo-Variables

We call these 'pseudo-variables' because they are 'read-only' and only meaningful during object messaging within the context of object methods.

Examples:

```
> (% A
     (p0 42)
     (p1 "Hello, World!")
     (get (prop) (-> symbol? any)
          (? prop self)))
'A
> (@ get A p0)
42
> (@ get A p1)
"Hello, World!"
> (@ get A p2)
#<undefined>
> self
#<undefined>
```

In *get* above we use the `self` pseudo-variable to retrieve the values of various properties on the object belonging to (or inherited by) the object *A*. Outside of the method, however, this pseudo-variable has an 'undefined' meaning.

Below is an alphabetical list of the library's pseudo-variables.

7.1 `defining-obj`

Provides access at run-time to the current method definer. This is the object that actually defines the method currently executing; in most cases, this is the object that defined the current method code in the source code of the program.

7.2 directly-defined-properties

Provides a reference to the object's directly-defined properties. This is an immutable hasheq table of property names and values.

7.3 `invokee`

Provides a pointer to the currently executing function.

7.4 `invokee-args`

Provides a pointer to the currently executing function arguments.

7.5 `invokee-arity`

Contains a normalized arity giving information about the number of by-position arguments accepted by `invokee`.

7.6 `kind-order`

Provides a reference to the collection of objects that form the `target-obj` inheritance sequence.

7.7 `self`

Provides a reference to the object whose method was originally invoked to reach the current method. Because of inheritance, this is not necessarily the object where the current method is actually defined.

7.8 `target-obj`

Provides access at run-time to the original target object of the current method. This is the object that was specified in the method call that reached the current method.

The target object remains unchanged when you use `inherited` to inherit a kind's method, because the method is still executing in the context of the original call to the inheriting method.

The `target-obj` value is the same as `self` in normal method calls, but not in calls initiated with the `delegated` keyword. When `delegated` is used, the value of `self` stays the same as it was in the delegating method, and `target-obj` gives the target of the delegated call.

7.9 `target-prop`

Provides access at run-time to the current target property, which is the property that was invoked to reach the current method. This complements `self`, which gives the object whose property was invoked.

7.10 `this`

Equivalent to `defining-obj`.

8 Inheritance

Conspiracy is a multiple inheritance paradigm. Conflict resolution is achieved from a deterministic algorithm that constructs at message application a `kind-order` for the object receiving the message. The process begins working from left to right along the `kind-order` looking at each *kind's* directly-defined property list until it finds a match or exhausts the list, at which point it returns `undefined`.

So how does this work in practice? Suppose we have 4 objects, defined as follows:

```
> (% Z (p 1))
'Z
> (% A (kinds Z))
'A
> (% B (kinds Z) (p 2))
'B
> (% C (kinds A B))
'C
```

The question now is whether the message *p* sent to *C* returns 1 or 2?

8.1 Resolving Inheritance Conflicts

Conspiracy resolves inheritance by constructing the `kind-order` in the following manner:

1. Start with an empty `kind-order`
2. Set `kind-order` index to 0.
3. Append the object to the end of the `kind-order`.
4. Get the object at `kind-order` index, if at end, stop
5. For each kind in the object's `kinds` list, working left to right, if it isn't in the `kind-order`, append it to the `kind-order`
6. increment the `kind-order` index
7. Go to step 4

For our example above the `kind-order` is (C A B Z Object) and the returned value is 2.

Once the system has built the kind-order inheritance begins looking for the desired property in the directly-defined properties of the objects from left to right. When it finds a match it stops — unless told by the property to do otherwise. Conspiracy provides two ways for a property to do so: through the `inherited` and `delegated` functions.

8.2 inherited

Passes control to an inherited object.

Syntax:

```
(inherited [#:when-undefined] [#:propname] [#:objname] arg ...)
  #:when-undefined : any/c
  propname          : symbol?
  objname           : object?
  arg               : any/c
```

By default `inherited` works its way down the `kind-order` beginning with the next object in the list after the one the statement is defined in.

`#:propname`: if supplied, it will look for that property name instead.

`#:objname`: if supplied, it will begin searching downstream (rightward in the `kind-order` from the object which is executing the `inherited`, looking from that point.

Here's an example that works its way down the `kind-order`:

```
> (% Z             (p () (-> any) (cons this (inherited #:when-undefined
'()))))
'Z
> (% A (kinds Z)   (p () (-> any) (cons this (inherited #:when-undefined
'()))))
'A
> (% B (kinds Z)   (p () (-> any) (cons this (inherited #:when-undefined
'()))))
'B
> (% C (kinds A B) (p () (-> any) (cons this (inherited #:when-undefined
'()))))
'C
> (@ p C)
'(C A B Z)
```

8.3 delegated

Passes control to the delegated object.

Syntax:

```
(delegated [#:when-undefined| [#:propname] #:objname arg ...)
  #:when-undefined : any/c
  propname          : symbol?
  objname           : object?
  arg               : any/c
```

Examples:

```
> (% Z (p0 () (-> any) this))
'Z
> (% A (kinds Z) (p0 () (-> any) (cons this (inherited))))
'A
> (% B (kinds Z) (p0 () (-> any) (cons this (inherited))))
```

```
'B
> (% C (kinds A B) (p0 () (-> any) (cons this (inherited))))
'C
> (% D (kinds A B) (p0 () (-> any) (cons this (delegated A))))
'D
> (@ p0 C)
'(C A B . Z)
> (@ p0 D)
'(D A . Z)
```

8.4 Differences between Inherited and Delegated

A key difference between delegation and inheritance can be observed from the examples above. Although both objects pass control to *A*, which then passes control in both cases to `inherited`, the inherited call to *A* continues down the `kind-order` of *C*, while the delegated call to *A* continues down the `kind-order` of *A*.

9 Templates

A template is an object definition that has the following:

- Is immutable.
- `implements` *ι-props* interfaces that require the existence of specified properties
- Does not satisfy the requirements of an *ι-porops* interface.
- `flags` no-assert

Examples:

```
> (% A (p0 number?))
'A
> (% B (implements A) (flags immutable no-assert))
'B
```

Now that we have defined B as a template-object, we can interrogate it.

9.1 `template-object?`

`Object` provides the `template-object?` method, which returns true if the object satisfies the above definition of a template object. In our case *B* does, *A* does not:

```
> (@ template-object? A)
#f
> (@ template-object? B)
#t
```

9.2 `template-objreqs`

`Object` provides this method, which returns the list of all missing object requirements for its ι-props interfaces, if the object is a *template*. The list is empty if the object is not a *template*.

```
> (@ template-objreqs A)
'()
> (@ template-objreqs B)
(list (objreq 'ι-props 'A 'p0 #<procedure:number?> #t))
```

9.3 `show` for *template-objects*

The `show` method provides a convenient way to interrogate template objects.

```
> (@ show B)
(% B
        (flags (immutable no-assert))
        (implements (A))
        (kinds (Object)))

• B template requirements:
        (p0 #<procedure:number?>)
```

9.4 τ form

As described in the section on *registration*, the τ form is a convenient way to define an 'anonymous' object deriving from a template (Although the syntax can be used for other 'anonymous' object definitions as well.)

```
> (τ B)
obj#30820730 (B) does not implement A ι-props property (? p0 obj#30820730)
-> #<procedure:number?>.
> (τ B p0: 42)
'obj#19611876
> (@ show obj#19611876)
(% obj#19611876
        (kinds (B))
        (p0 42))
```

10 Object

The Object library.

- Objects are 'classless', there is no need to instantiate them. They can, however, be *flagged* as immutable, effectively making them a template for other objects.

- Objects can inherit from multiple pre-defined objects. The specification of inheritance creates a deterministic `kind-order` in which properties are located.

- Message passing can be controlled through `inherited` and `delegated` forms that pass control down the inheritance order or delegate to other objects. Inherited or delegated messages can specify a default response when the desired property is undefined.
- All object properties are 'public'; an object doesn't hide its properties from other objects.
- Properties are 'untyped'; any datatype can be associated with the property, although type can be enforced if the object *implements an ι-`props` interface* requesting it to do so. An object may implement a μ-`props` that supplies a property and value for the object, if it does not already do so.
- Property values can be retrieved or applied, and unless the object is immutable, they can be set. Retrieval and application can specify what to return when the property is undefined. Additionally, an object can indicate a special `prop-not-defined` method to handle situations when the property is not directly-defined or inherited.
- An object can reference its own properties with the 'self' keyword.
- Objects can be dynamically *replaced* in the database, effecting the inheritance order and behaviors of other objects.
- Objects can be dynamically *modified* in the database, with a superseding definition that will inherit properties from the previous definition, or replace or remove them altogether.
- Objects can be dynamically *removed* from the database, along with their descendants and modifications.
- The objects database can be parameterized to include only specified objects and their inheritance ancestors, or exclude specified objects and their descendants.

10.1 Object Properties

The following is an alphabetical listing of properties directly-defined by `Object`.

10.1.1 `build-kind-order`

The inheritance order is deterministic (i.e., it will always be the same for a given situation), and it depends on the full kind tree of the original target object.

Syntax:

(@ **build-kind-obrder** *objname* [*kind*])
objname : object?
kind : (or/c **#f** object?)

10.1.2 `characteristics-template?`

Returns true if the object is a characteristics-template; otherwise it returns false.

Syntax:

(@ **characteristics-template?** *objname*)
 objname : `object?`

10.1.3 `construct`

Called during the object registration process to complete any work necessary to build the module definition.

10.1.4 `create-clone`

Creates a new object that is an identical copy of this object. The new object will have the same kinds as the original, and the identical set of properties defined in the original. No constructor is called in creating the new object, since the object is explicitly initialized by this method to have the exact property values of the original.

The clone is a 'shallow' copy of the original, which means that the clone refers to all of the same objects as the original. For example, if a property of the original points to a vector, the corresponding property of the clone points to the same vector, not a copy of the vector.

Syntax:

(@ **create-clone**)

10.1.5 `create-clone/`ε

Like `create-clone`, exceept that the object's `construct` is executed, any ε-props methods from the original object are removed prior to consturction. An *objname* can be provided, but must not already be registered in the database.

Syntax:

(@ **create-clone/**ε [*objname*])
 objname : `(and/c symbol? (not/c object?))`

10.1.6 `create-instance`

Creates a new instance of the target object. This method's arguments are passed directly to the constructor, if any, of the new object; this method doesn't make any other use of the arguments. The method creates the object, invokes the new object's constructor, then returns the new object.

Syntax:

(@ **create-instance** *objname* [*arg* ...])
 objname : `object?`
 arg : `any/c`

10.1.7 `create-instance-of`

Creates a new instance based on multiple kinds. This is a "static" (kind-level) method, so you can call it directly on kind. With no arguments, this simply creates a basic kind instance; this is equivalent to the create-instance method with no arguments.

The arguments give the kinds, in "dominance" order. The kinds appear in the argument list in the same order in which they'd appear in an object definition: the first argument corresponds to the leftmost kind in an ordinary object definition. Each argument is either a kind or a list. If an argument is a list, the first element of the list must be a kind, and the remainder of the elements are the arguments to pass to that kind's constructor. If an argument is simply a kind (not a list), then the constructor for this kind is not invoked at all.

Syntax:

(@ **create-instance-of** *objname* [*arg* ...])
objname : `object?`
arg : `any/c`

10.1.8 `directly-inherits-kind`

Produces a list of objects whose kinds property contains this object, or an empty list if there are none.

10.1.9 `directly-inherits-kind?`

Returns true if all the args directly inherit from this object; false otherwise.

Syntax:

(@ **directly-inherits-kind?** *objname* [*kind* ...])
objname : `object?`
kind : `object?`

10.1.10 `flags`

See section on `flags`. Object is set as `immutable`.

10.1.11 `flags!`

Toggles the specified flags for this object.

Syntax:

(@ **flags!** *objname* *flag* [*flag* ...])
objname : `object?`
flag : `symbol?`

10.1.12 `flags?`

Returns #t if the object directly defines a flags property and the flags list contains the flags appropriate to the value of the `#:all` keyword parameter.

Syntax:

(@ **flags?** *objname* [#:all] *flag* [*flag* ...])
 objname : object?
 #:all : boolean?
 flag : symbol?

When #:all is #t (the default), then the object must match all the flags indicated. When #:all is #f, then the method returns #t if any match. Otherwise returns #f.

10.1.13 `get-kind-order`

Returns the pre-built ordering of kinds created as part of the `objeect-call` process.

10.1.14 `get-kinds-list`

Returns a list containing the immediate kinds of the object. The list contains only the object's direct kinds, which are the objects that were explicitly listed in the object's `kinds` property.

10.1.15 `get-method`

Gets the procedure for one of the object's methods.

Syntax:

(@ **get-method** *objname propname*)
 objname : object?
 propname : symbol?

10.1.16 `get-objreqs`

Returns a list of elements representing the 'object requirements' for the object:

- a list of properties required by interfaces, but missing from this object's definition/inheritance.
- a list of properties required by interfaces The elements of these lists are triples consisting of:
- propname
- interface property value
- interface name

10.1.17 `get-prop-list`

Returns a list of the properties directly-defined by this object. Each entry in the list is a property symbol. The returned list contains only properties directly-defined by the object; inherited properties are not included, but may be obtained by explicitly traversing the `kinds` list and calling this method on each kind.

10.1.18 `get-prop-params`

Returns a list of information associated with the parameters and result of the property associated with this object, or #f if the property is undefined.

- mask encoding of by-position arguments
- list of required keyword arguments
- list of accepted keyword arguments
- result arity

Syntax:
(@ **get-prop-params** *objname* *prop*)
objname : `object?`
prop : `symbol?`

10.1.19 `inherits-kind`

Produces a list of objects inheriting from this object, or an empty list if there are none.

10.1.20 `inherits-kind?`

Returns true if all the args inherit from this object; false otherwise.

Syntax:
(@ **inherits-kind?** *objname* [*kind* ...])
objname : `object?`
kind : `object?`

10.1.21 `kinds`

See section on `kinds`. Object is the only object registered in the database whose `kinds` property lists itself.

10.1.22 `modifies`

Produces a list of objects modified by this object. If the object hasn't been modified the list is empty.

10.1.23 `next-non-modobj`

Returns the next non-modified object in the object's kind-order.

10.1.24 `of-kind?`

Determines if the object is an instance of the kind, or an instance inheriting from kind. Returns true if so, #f if not. This method always returns true if kind is Object, since every object ultimately derives from Object.

Syntax:

(@ **of-kind?** *objname kind*)
 objname : `object?`
 kind : `object?`

10.1.25 `prop-defined`

Determines if the object defines or inherits the property propname, according to the flags value. If flags is not specified, a default value of prop-def-any is used.

Syntax:

(@ **prop-defined** *objname propname* [*flag*])
 objname : `object?`
 propname : `symbol?`
 flag : `(or/c` **prop-def-any**
```
                    prop-def-directly
                    prop-def-inherits
                    prop-def-get-kind)
```

The valid flags values are:

Flag	**Function Returns**
prop-def-any	#t if the object defines or inherits the property; Otherwise #f
prop-def-directly	#t only if the object directly-defines the property; if it inherits the property from a kind, the function r
prop-def-inherits	#t only if the object inherits the property from a kind; if it defines the property directly, or doesn't def
prop-def-get-kind	the kind from which the property is inherited, or this object if the object defines the property directly.

10.1.26 `prop-inherited`

Determines if the object inherits the property prop. target is the "original target object," which is the object on which the method was originally invoked. definer is the "defining object," which is the object defining the method which will b inheriting the kind implementation.

Syntax:

(@ **prop-inherited** *objname*
  propname
  target
  definer
  [*flag*])
  objname : `object?`
  propname : `symbol?`
  taget : `object?`
  definer : `object?`
  flag : `(or/c` **prop-def-any**
```
                    prop-def-get-kind)
```

The return value depends on the value of the flags argument:

Flag	**Function Returns**
prop-def-any	#t if the object defines or inherits the property; Otherwise #f
prop-def-get-kind	the kind from which the property is inherited, or this object if the object defines the property directly.

This method is most useful for determining if the currently active method will invoke an inherited version of the method if it uses the inherited operator; this is done by passing targetprop for the prop parameter, targetobj for the target parameter, and defining-obj for the definer parameter. When a kind is designed as a "mix-in" (which means that the kind is designed to be used with multiple inheritance as one of several base kinds, and adds some isolated functionality that is "mixed" with the functionality of the other base kinds), it sometimes useful to be able to check to see if the method is inherited from any other base kind involved in multiple inheritance. This method allows the caller to determine exactly what inherited will do.

10.1.27 `prop-type`

Returns the datatype of the given property of the given object, or undefined if the object does not define or inherit the property. This function does not evaluate the property, but merely determines its type. The return value is one of the symbols returned by examining the `struct->vector` of the value.

Syntax

(@ **prop-type** *objname* *prop*)
objname : `object?`
prop : `symbol?`

10.1.28 `set-kinds-list!`

Sets the `kinds` property of the object to the specified list of objects.

Syntax:

(@ **set-kinds-list!** *objname* *kinds*)
objname : `object?`
kinds : `(listof object?)`

10.1.29 `set-method!`

Assigns the procedure proc as a method of the object, using the property prop.

Syntax:

(@ **set-method!** *objname* *propname* *proc* `[#:contract]`)
objname : `object?`
propname : `symbol?`
proc : `procedure?`
`#:contract` : `contract?`

If a contract is provided then the proc is bound to it.

10.1.30 `show`

Displays the defined definition of self object.

Syntax:

```
(@ show objname [#:flag
                   #sign
                   #:precision
                   #:notation
                   #:format-exponent])]
 objname : object?
 #:flag     : (or/c #f +kinds +mods +all 'chs)
 #:sign     : (or/c #f  '+ '++ 'parens
                             (let ([ind (or/c string? (list/c string?
string?))])
                               (list/c ind ind ind)))
 #:precision : (or/c exact-nonnegative-integer?
                                  (list/c '= exact-nonnegative-integer?))
 #:notation : (or/c 'positional 'exponential
                                (-> rational? (or/c 'positional 'ex-
ponential)))
 format-exponent : (or/c #f string? (-> exact-integer? string?))
```

Flag options are:

- +kinds - displays the object and its directly-defined kinds.
- +mods - displays all mods in self kind-order list.
- +all - displays all kinds in self kind-order list.
- chs - displays requirements and characteristics.

The *sign*, *precision*, *notation*, and *format-exponent* optional arguments control formating for inexact number property values, based on the ~*r* formatting rules.

10.1.31 `template-object?`

Returns true if this object is a template-object; false otherwise. See Templates for template-object requirements.

10.1.32 `template-objreqs`

If the object is a template it returns a list of missing objreqs that must be supplied by any object deriving from this object. If the object is not a template, then an empty list is returned.

10.1.33 ε`-props-assert`

Objects that implement ε`-props` flagged objects must NOT define the ε-props object properties.

10.1.34 ι-`props-assert`

Enforces the interfaces this object `implements`. If the object does not implement a property indicated by one of its interfaces an error is thrown. By default this is only enforced when the object is constructed. The `no-assert` flag can be used to bypass this object's assertion though it will still affect objects derived from this object.

10.1.35 μ-`props-assert!`

Enforces the μ-`props` interfaces this object `implements`. Adds the interface properties to this object when it doesn't directly-define or inherit them. The order of interfaces takes precedence.

10.2 Object Functions and Macros

The following is an alphabetical listing of object functions and macros.

10.2.1 aux

A macro that binds property namess to their values for the directly-defined propertiees of `self` during a method call.

Syntax:

```
(aux var [var ...])
 var      = varname
                   | (varname value)
 varname : symbol?
```

Examples:

```
> (% Rectangle
     (width 3)
     (height 2)
     (area () (-> real?)
           (aux width height)
           (+ width height)))
'Rectangle
> (@ area Rectangle)
5
```

It's important to remember that the *varname* is bound to the object's associated directly-defined property value, and to `undefined` if it does not exist. If the property is a method the method procedure is returned, not the application of the method.

10.2.2 `debug`

A macro that produces diagnostic displays associated with object messaging.

10.2.3 `debug!`

A macro for setting the `debug` boolean.

Syntax:

(**debug!** *value*)
value : `boolean?`

10.2.4 `default-kind`

A macro for returning the default kind(s) that objects inherit when they are not defined with a `kinds` property.

10.2.5 `default-kind!`

A macro for setting the value of `default-kind`.

Syntax:

(**default-kind!** *kind* [*kind* ...])
kind : `object?`

10.2.6 `defining-obj`

A pseudo-variable. Provides access at run-time to the current method definer. This is the object that actually defines the method currently executing; in most cases, this is the object that defined the current method code in the source code of the program.

10.2.7 `delegated`

A function for passing control to another object's property and `kind-order`.

10.2.8 directly-defined-properties

A pseudo-variable. Provides access at run-time to the `self` object's directly-defined properties.

10.2.9 `immutable-object?`

A function for determining whether an object is immutable. Returns `(and (object?` *value*`) (@ flags?` *value* `immutable))`

Syntax:

(**immutable-object?** *value*))

Immutable objects prohibit property modification through `!`.

10.2.10 `inherited`

A function for passing control down the object's `kind-order`.

10.2.11 `invokee`

A pseudo-variable. Provides a pointer to the currently executing function.

10.2.12 `invokee-args`

A pseudo-variable. Provides a pointer to the currently executing function arguments.

10.2.13 `invokee-arity`

A pseudo-variable. Contains a normalized arity giving information about the number of by-position arguments accepted by `invokee`.

10.2.14 `kind-order`

A pseudo-variable. Provides a reference to the collection of objects that form the inheritance sequence.

10.2.15 `mutable-object?`

A function for determining whether an object is mutable. Returns `(not (immutable-object?` *value*`))`

Syntax:

`(`**mutable-object?** *value*`))`

Objects are mutable by default.

10.2.16 `object`

A macro for defining objects (See section Object Defining).

10.2.17 `objects`

A macro for the objects database.

10.2.18 `objects-copy`

A function for filtering the objects database. Returns a hash copy of objects filtering on kinds. Results are as follows:

filter-not?	**kinds**	**contains**
#f	o	o and inherited kinds
#t	o	All objects except o, o modifies, and inheriting o
#f	empty	Empty database
#t	empty	All objects currently in database

10.2.19 `object-names`

A macro for object names registered in the objects database.

10.2.20 `object->list`

A function that converts an object into a list consisting of the object name followed by property value pairs.

Examples:

```
> (% A
    (p0 () (-> any) #t)
    (p1 'foo)
    (p2 (a b c)))
'A
> (object->list A)
'(A (kinds Object) (p0 . #<procedure:...nspiracy/object.rkt:260:42>)
(p1 . foo) (p2 a b c))
```

10.2.21 `object?`

A function for determining whether a value is an object. Returns #t if the value is a symbol registered in the objects database; otherwise #f.

Syntax:

(**object?** *value*

10.2.22 `Object?`

A function used to identify the Object symbol. Returns #t if the value is `'Object`; otherwise #f.

Syntax:

(**Object?** *value*

10.2.23 `objreq`

A struct identifying 'object requirements'.

Syntax

(**objreq** *type objname propname value proc?*)

type :

objname	: `object?`
propname	: `symbol?`
value	: `any/c`
proc?	: `boolean?`

This is a representation of an individual object property requirement. The type determines which interface type should be asserted.

10.2.24 `self`

A pseudo-variable. Provides a reference to the object whose method was originally invoked to reach the current method. Because of inheritance, this is not necessarily the object where the current method is actually defined.

10.2.25 `target-obj`

A pseudo-variable. Provides access at run-time to the original target object of the current method. This is the object that was specified in the method call that reached the current method.

10.2.26 `target-prop`

A pseudo-variable. Provides access at run-time to the current target property, which is the property that was invoked to reach the current method. This complements `self`, which gives the object whose property was invoked.

10.2.27 `this`

A pseudo-variable. Equivalent to `defining-obj`.

10.2.28 `true?`

A function specific to Conspiracy truth and falsehood. Returns #t when *value* is neither #f or `undefined`.

Syntax

(**true?** *value*)

10.2.29 `undefined`

A symbol representing 'undefined' values.

10.2.30 `undefined?`

A function for determining whether a value is the `undefined` datatype. Returns #t if the value is *undefined*; otherwise #f.

Syntax:

(**undefined?** *value*)

value : `any/c`

10.2.31 `without-objects`

A macro for parameterizing the current-objects, current-default-kind, and current-debug, restoring their values once the context of the macro has terminated.

Filter objects by kinds and their inherited objects.

10.2.32 `with-objects`

A macro for parameterizing the current-objects, current-default-kind, and current-debug, restoring their values once the context of the macro has terminated.

Filter-not objects by kinds and their modified and inheriting objects.

10.2.33 `!`

A function for setting the property value of an object.

10.2.34 `%`

A macro for registering objects.

10.2.35 `%*`

A macro for registering ’anonymous’ objects.

10.2.36 `%+`

A macro for registering modifed objects in the database.

10.2.37 `%-`

A macro for unregistering objects from the database.

10.2.38 `%=`

A macro for registering the replacement of an object in the database.

10.2.39 `?`

A function for retrieving the property value of an object.

10.2.40 `@`

A function for applying the property value of an object to the specified arguments.

10.2.41 `τ`

A macro for registering ’anonymous’ objects in a ’template’ fashioned syntax.

10.2.42 `#%top`

A macro wrapping unbound identifiers in quotes.

10.2.43 `τ`

A macro for registering ’anonymous’ objects in a ’template’ fashioned syntax.

11 Characteristics Object

```
(require conspiracy/object
         (submod conspiracy/object Characteristics)
```

Characteristics are those properties of an object that make it interesting. Conspiracy defines this as a relationship among 3 objects:

- An ι-props object defining properties required for those characteristics.
- A μ-props object defining the characteristic properties.
- A characteristics-template object.

11.1 `characteristics-template?`

This method returns true when:

- The object derives from Characteristics.
- The object implements a μ-props object.
- The object is a template-object.

11.2 The Characteristics Relationship

Even when we meet the requirements of Characteristics, there's no guarantee that the μ-props properties will be satisfied by the values supplied when deriving from the characteristics-template. They may not reference the characteristics-template's ι-proops properties, or they may not be satisfied by them.

But to give an example of how characteristics are designed to work, imagine a circle template that produces objects calculating diameter, circumference, and area, given the readius of the circle.

Since *radius* is our required template value, we define an ι-props object for it:

```
>(% ι-Circle (radius real?))
'ι-Circle
```

No `flags` property is needed for defining an ι-props object. By convention we prefix the objet name with "ι-" though this has no effect on the object definition. In addition we require the `radius` to be a *real* number.

Next we translate the following formulas into μ-props properties:

diameter = 2 * radius circumference = 2 * pi * radius area = pi * radius^2

into:

```
>(% μ-Circle (flags immutable μ-props)
   (diameter () (-> real?)
             (aux radius)
             (* 2 radius))
   (circumference () (-> real?)
                  (aux radius)
                  (* 2 pi radius))
```

```
    (area () (-> real?)
           (aux radius)
           (* pi radius radius)))
'μ-Circle
```

Notice that these properties take no argumentts, and get the `radius` value from `self`. We previx the object name with "μ-", again by convention. But this object must specify μ`-props` in the `flags` property to distinguish it from ι`-props` and ε`-props` implementations.

Also, we are using the `aux` macro to bind `radius` to the directly-defined property value of `self`. Since these μ-props methods will be copied to our *Characteristics* derived object, the `construct` call will be to the object being constructed at that point.

Next we define the *characteristis-template*:

```
>(% Circle (flags immutable no-assert)
    (implements ι-Circle μ-Circle)
    (kinds Characteristics))
'Circle
```

11.3 `characteristics`

Syntax:

(@ **characteristics** *objname* [#:prop+val?])
objname : object?
#:prop+val? : (or/c **#f** #t)

This method returns:

- By default, or when #:prop+val? #f, a list of 2 elements: The first is a list is of the object's missing ι`-props` propnames. The second is a list of the object's μ`-props` propnames.
- When #:prop+val? #t, the sublists consist of the porpname / value pairs for those elements.

Conspiracy says that a *characteristics-template* has no characteristics. By this it means that the template is merely a recipe for an object that does have characteristics. So the value of a propname/value pair is undefined for *characteristics-templates*.

Characteristics-templates produce immutable anonymous objects, creating directly-defined properties corresponding to the μ-props properties, for their derivations, excpet that the property values are the result of the property invocation.

```
> (τ Circle radius: 10)
'obj#63998617
> (τ Circle radius: 10)
'obj#71842977
> (@ characteristics obj#71842977)
'((radius) (area circumference diameter))
> (@ characteristics obj#71842977 #:prop+val? #t)
'(((radius . 10)) ((area . 314.1592653589793) (circumference . 62.83185307179586)
(diameter . 20)))
```

11.4 `show` and characteristics

Objects that have characteristics produce additional `show` information.

```
> (@ show obj#71842977)
(% obj#71842977
        (area 314.1592653589793)
        (circumference 62.83185307179586)
        (diameter 20)
        (flags (immutable))
        (kinds (Circle))
        (radius 10))

 • obj#71842977 characteristics requirements:
        (radius 10)
 • obj#71842977 characteristics:
        (area 314.1592653589793)
        (circumference 62.83185307179586)
        (diameter 20) (diameter 20)
```

Objects that are not anonymous and mutable created with the *characteristics-template* are different in that their μ-`props` properties are not directly-defined with property values that are the result of the property invocation.

The directly-defined ι-`props` properties of the mutable Characteristics object derivation can be changed, with a corresponding change in the reporting of characteristics.

```
> (% C (kinds Circle) (radius 20))
'C
> (@ show C)
(% C
        (kinds (Circle))
        (radius 20))

 • C characteristics requirements:
        (radius 20)
 • C characteristics:
        (area 1256.6370614359173)
        (circumference 125.66370614359172)
        (diameter 40)
> (! radius C 10)
> (@ show C)
(% C
        (kinds (Circle))
        (radius 10))

 • C characteristics requirements:
        (radius 10)
 • C characteristics:
        (area 314.1592653589793)
        (circumference 62.83185307179586)
        (diameter 20)
```

12 Nothing Object

The Nothing submodule of Conspiracy object implements a parent, sibling, child relationship.

```
> (require conspiracy/object
           (submod conspiracy/object Nothing))
'Object
'ε-Nothing
'Nothing
```

12.1 Containment

This module specializes in a mechanism for representing an 'object tree'. The relationship among objects in the tree is independent of their relationships through inheritance or delegation. To help illustrate what we mean by an object tree we'll make use of the following classic diagram:

```
meadow
   |
mailbox - player - torch
    |                |
  note             battery
```

In the *meadow* there is a *mailbox*, a *player*, and a *torch*. Inside the *mailbox* is a *note*. Inside the *torch* is a *battery*.

One way to model these relationships would be with a set of pointers: *location* and *contents*. The *location* would point to a single object. The *contents* would be a list of objects.

The `Nothing` Library implements a different model using `parent`, `sibling`, `child` relationships that place the object within the object tree.

12.1.1 `parent`

In the above diagram we would say that *meadow* is the parent of *mailbox*, *player*, and *torch*. The *mailbox* is the `parent` of the *note*. The *torch* is the `parent` of the *battery*. Incidentally, the *meadow* has a `parent`, which is the `Nothing` object itself.

12.1.2 `sibling`

The `sibling` of the *mailbox* is the *player*, but the relationship isn't reciprocal, it flows from left to right. The `sibling` of the *player* is the *torch*. Finally, the *torch* has no `sibling`, or as the model would say, the `sibling` of *torch* is `Nothing`.

12.1.3 `child`

While the objects in the *meadow* can be spoken of as `children` of the meadow, there is only one `child` of *meadow*, the *mailbox*.

12.2 Defining Objects Derived From `Nothing`

Object's derived from `Nothing` have a special `construct` property. This checks the object's directly-defined properties for `parent`, `sibling`, and `child` and adds any that are missing, assigning them the value of `Nothing`. It then moves the object into the object tree.

12.3 Defining The Object Tree

An object derived from `Nothing` `implements` ε-`Nothing`, which has 'existential exclusion properties' `sibling` and `child`. This means that these objects cannot have those properties in their definition (the library will supply them).

The following describes how to define an object tree:

- Define each object with the appropriate `parent` property.
- Do not define `sibling` or `child` properties for the objects.
- Define sibling objects in right to left order.

Going back to our *meadow* object tree:

```
> (% meadow)
'meadow
> (% torch   (parent meadow))
'torch
> (% battery (parent torch))
'battery
> (% player (parent meadow))
'player
> (% mailbox (parent meadow))
'mailbox
> (% note (parent mailbox))
'note
> (@ show meadow)
(% meadow
        (child mailbox)
        (kinds (Nothing))
        (parent Nothing)
        (sibling Nothing))
> (@ show mailbox)
(% mailbox
        (child note)
        (kinds (Nothing))
        (parent meadow)
        (sibling player))
> (@ show note)
(% note
        (child Nothing)
        (kinds (Nothing))
        (parent mailbox)
        (sibling Nothing))
> (@ show player)
(% player
        (child Nothing)
        (kinds (Nothing))
        (parent meadow)
        (sibling torch))
> (@ show torch)
(% torch
        (child battery)
        (kinds (Nothing))
        (parent meadow)
        (sibling Nothing))
> (@ show battery)
(% battery
        (child Nothing)
        (kinds (Nothing))
        (parent torch)
        (sibling Nothing))
```

12.4 `Nothing` Properties

The following documents the properties of `Nothing`.

12.4.1 `in?`

Returns #t if *objname* is the `parent` of `self`.

Syntax:

(**in?** *objname*) `-> boolean?`
objname `: object?`

12.4.2 `is-in?`

Determines if `self` is in the sub-tree whose parent is *objname*, and if so returns #t; otherwise returns #f.

Syntax:

(**is-in?** *objname*) `-> boolean?`
objname `: object?`

12.4.3 `move-to`

Moves `self` to *objname* in the object-tree.

Syntax:

(**move-to** *objname*) `->|`

objname `: object?`

In our *meadow* object tree, suppose the player 'takes' the battery from the torch:

```
> (@ move-to battery player)
> (@ show player)
(% player
        (child battery)
        (kinds (Nothing))
        (parent meadow)
        (sibling torch))
> (@ show torch)
(% torch
        (child Nothing)
        (kinds (Nothing))
        (parent meadow)
        (sibling Nothing))
> (@ show battery)
(% battery
        (child Nothing)
        (kinds (Nothing))
        (parent player)
        (sibling Nothing))
```

And now suppose we move the torch to the mailbox.

```
> (@ move-to torch mailbox)
> (@ show mailbox)
(% mailbox
        (child torch)
        (kinds (Nothing))
        (parent meadow)
        (sibling player))
> (@ show note)
(% note
        (child Nothing)
        (kinds (Nothing))
        (parent mailbox)
        (sibling Nothing))
> (@ show player)
(% player
        (child battery)
        (kinds (Nothing))
        (parent meadow)
        (sibling Nothing))
```

```
> (@ show torch)
(% torch
        (child Nothing)
        (kinds (Nothing))
        (parent mailbox)
        (sibling note))
> (@ show battery)
(% battery
        (child Nothing)
        (kinds (Nothing))
        (parent player)
        (sibling Nothing))
```

The examples above show that the 'eldest' sibling is the last in the chain. The `move-to` always places the object as the `child` of the new `parent` and pushes the previous `child` down the chain.

12.4.4 `parents`

Accumulates a list by applying *fn*, if provided, to each successive `parent` of `self` until `Nothing` is reached. If *fn* is #f then *acc* is the list of successive *parents* of the original object.

Syntax:

(**parents** [*fn*] [*acc*]) `-> list?`
fn `: (or/c #f procedure?)`
acc `: list?`

12.4.5 `siblings`

Accumulates a list by applying *fn*, if provided, to each successive `sibling` of `self` until `Nothing` is reached. If *fn* is #f then *acc* is the list of successive *siblings* of the original object.

Syntax:

(**siblings** [*fn*] [*acc*]) `-> list?`
fn `: (or/c #f procedure?)`
acc `: list?`

12.4.6 `children`

Accumulates a list by applying *fn*, if provided, to each successive `child` of `self` until `Nothing` is reached. If *fn* is #f then *acc* is the list of successive *children* of the original object.

Syntax:

(**children** [*fn*] [*acc*]) `-> list?`
fn `: (or/c #f procedure?)`
acc `: list?`

12.4.7 `contents`

Returns a list of objects that are the *contents* of this object. If the object has no contents, returns an empty list.

Syntax:
(**contents**) `-> list?`

12.5 `Nothing` Functions and Macros

The following documents the functions and macros of `Nothing`.

12.5.1 `object-loop`

Iterates over all objects. When *pred* is supplied the object is considered for selection only when the procedure returns (not/c (or/c #f undefined?)). When *acc* is supplied the body statements are only called once with the list of accumulated objects; otherwise body statements are called once for each object satisfying *proc*.

Syntax:
(**object-loop** (*var* [*pred* [*acc*]] `body0 body ...) -> any/c`
pred `: procedure?`
acc `: identifier?`

Note: This macro simulates the *objectloop* of Graham Nelson's Inform language.

Examples:

```
> (% meadow)
'meadow
> (% torch (parent meadow))
'torch
> (% battery (parent torch))
'battery
> (% player (parent meadow))
'player
> (% mailbox (parent meadow))
'mailbox
> (% note (parent mailbox))
'note
(object-loop (o) (printf "~%~a" o))
Object
battery
meadow
torch
mailbox
player
note
ε-Nothing
Nothing
(object-loop (o #t acc) acc)
```

```
'(% battery meadow torch mailbox player note ε-Nothing Nothing)
(object-loop (o (@ of-kind? o Nothing) acc) acc)
'(battery meadow torch mailbox player note Nothing)
(@ move-to torch player)
(object-loop (o (@ is-in? o player) acc) acc)
'(battery torch)
```

Of course, it should be noted that Racket *for loops* can offer more flexible and powerful alternatives.

13 Rule Object

The Rule submodule of Conspiracy object implements a conditional control flow datatype.

```
> (require conspiracy/object
           (submod conspiracy/object Rule))
'Object
'Rule
```

13.1 Executing Rules

Syntax:

```
(@ exec objname [arg ...] [#:when-undefined])
  objname          : object?
  arg              : any/c
  #:when-undefined : any/c
```

When a *rule* receives an `exec` message the following sequence occurs:

- The rule's `pred` property is applied with the same *arg*s supplied to `exec`.
- If `pred` returns #f, rule processing terminates and #f is returned.
- The rule's `proc` property is applied with the same *arg*s supplied to `exec`.
- The rule's `rulebook` property is processed, with each rule of the rulebook repeating the above process.
- The rule's `value` property is updated with the result of its execution.

Rules can be a little hard to wrap your head around. But a simple example involving the nesting of rulebooks should help illustrate their function.

Examples:

```
> (% r0 (proc () (-> any) (printf "self=~a~%" self))
      (rulebook (r1 r2 r3)))
'r0
> (% r1 (proc () (-> any) (printf "self=~a~%" self)))
'r1
> (% r2 (proc () (-> any) (printf "self=~a~%" self))
     (rulebook (r21 r22 r23)))
'r2
> (% r3 (proc () (-> any) (printf "self=~a~%" self)))
'r3
> (% r21 (proc () (-> any) (printf "self=~a~%" self)))
'r21
> (% r22 (proc () (-> any) (printf "self=~a~%" self)))
'r22
> (% r23 (proc () (-> any) (printf "self=~a~%" self)))
'r23
> (@ exec r0)
self=r0
self=r1
self=r2
self=r21
self=r22
self=r23
self=r3
> (@ value r22)
> (@ value r2)
> (@ value r0)
```

In the above example *r0* executes, and then executes the rules in its `rulebook`, which in turn execute the rules in their *rulebooks*. It's a depth-first tree traversal beginning with *r0*.

13.1.1 Short-circuiting with `raise`

syntax:
(
(**raise** *value*)
  value : (not/c exn?)

Strictly speaking, `raise` can be used with any value. We're interested in non-exception values because *Rule* will trap those values and perform special processing with them.

In particular, when you `raise` a non-exception when executing a rule that rule traps the value and stores it in its `value` property, just as it would the result of normal rule processing.

But the processing will terminate at that point, with control going back up the `rule-chain` to what we might consider the root of the call tree, where it is then returned as a normal result. The consequence is that no further rule processing is performed.

Examples:

```
> (% r0 (proc () (-> any) (printf "self=~a~%" self))
     (rulebook (r1 r2 r3)))
'r0
> (% r1 (proc () (-> any) (printf "self=~a~%" self)))
```

```
'r1
> (% r2 (proc () (-> any) (printf "self=~a~%" self))
     (rulebook (r21 r22 r23)))
'r2
> (% r3 (proc () (-> any) (printf "self=~a~%" self)))
'r3
> (% r21 (proc () (-> any) (printf "self=~a~%" self)))
'r21
> (% r22 (proc () (-> any) (printf "self=~a~%" self) (raise self)))
'r22
> (% r23 (proc () (-> any) (printf "self=~a~%" self)))
'r23
> (@ exec r0)
self=r0
self=r1
self=r2
self=r21
self=r22
> (@ value r22)
'r22
> (@ value r2)
'r22
> (@ value r0)
'r22
```

The above examples demonstrates one way to 'short-circuit' rule processing. The next section demonstrates the other.

13.1.2 Short-circuiting with `raise-rule-value`

Syntax:
(**raise-rule-value** *value* [*objname*])
value : `any/c`
objname : `(`λ `(v) (member v (cdr rule-chain)))`

Raises the value to be caught at any point in `(cdr rule-chain)`. By default this would be the first of that list, or if `empty`, then `(car rule-chain)`.

Examples:

```
> (% r0 (proc () (-> any) (printf "self=~a~%" self))
     (rulebook (r1 r2 r3)))
'r0
> (% r1 (proc () (-> any) (printf "self=~a~%" self)))
'r1
> (% r2 (proc () (-> any) (printf "self=~a~%" self))
    (rulebook (r21 r22 r23)))
'r2
> (% r3 (proc () (-> any) (printf "self=~a~%" self)))
'r3
```

```
> (% r21 (proc () (-> any) (printf "self=~a~%" self)))
'r21
> (% r22 (proc () (-> any) (printf "self=~a~%" self) (raise-rule-value
self)))
'r22
> (% r23 (proc () (-> any) (printf "self=~a~%" self)))
'r23
> (@ exec r0)
self=r0
self=r1
self=r2
self=r21
self=r22
self=r3
> (@ value r22)
'r22
> (@ value r2)
'r22
> (@ value r0)
```

Here `raise-rule-value` 'short-circuits' the processing of the `rulebook` in which it is referenced. So *r0* `rulebook` processing continues normally.

Another example:

```
> (% r0 (proc () (-> any) (printf "self=~a~%" self))
     (rulebook (r1 r2 r3)))
'r0
> (% r1 (proc () (-> any) (printf "self=~a~%" self)))
'r1
> (% r2 (proc () (-> any) (printf "self=~a~%" self))
     (rulebook (r21 r22 r23)))
'r2
> (% r3 (proc () (-> any) (printf "self=~a~%" self)))
'r3
> (% r21 (proc () (-> any) (printf "self=~a~%" self)))
'r21
> (% r22 (proc () (-> any) (printf "self=~a~%" self) (raise-rule-value
self r0)))
'r22
> (% r23 (proc () (-> any) (printf "self=~a~%" self)))
'r23
> (@ exec r0)
self=r0
self=r1
self=r2
self=r21
self=r22
'r22
```

```
> (@ value r22)
'r22
> (@ value r2)
'r22
> (@ value r0)
'r22
```

13.2 Rule Properties

The following is an alphabetical listing of properties directly-defined by `Rule`.

13.2.1 `active?`

Boolean indicator signifying whether the rule is active or inactive.

13.2.2 `exec`

The method used to *execute* a rule. This method invokes `pred` first to determine if the rule is prepared to be processed. If it is, then `proc` is invoked, followed by the processing of the rules in the rule's `rulebook`.

Syntax:

```
(@ exec objname arg ... [#:when-undefined])
   objname         : object?
   arg             : any/c
   #:when-undefined : any/c
```

13.2.3 `pred`

Method deciding whether to invoke the rule's `proc` and `rulebook` processing.

Sample Syntax:

```
(pred args (->* () #:rest (listof any/c) any) (? active? self))
```

13.2.4 `proc`

Method representing the rule's 'rule'.

Sample Syntax:

```
(proc args (->* () #:rest (listof any/c) any) body0 body ...)
```

13.2.5 `rulebook`

A list of rules to be *executed* after the rule's `proc` property.

13.2.6 `value`

The result of `exec` processing for the rule.

13.3 Rule Functions and Macros

The following is an alphabetical listing of `Rule` functions and macros.

13.3.1 `raise-rule-value`

Raises the value up the `rule-chain`. Will be handled by the optional *objname*, which defaults to `(first (rest rule-chain))` or to `(first rule-chain` when `(rest rule-chain` is empty.

Syntax:

`(`**raise-rule-value** *value* `[`*objname*`])`

value : `any/c`

objname : `object?`

13.3.2 `rule-chain`

A macro that represents the execution chain of rules from newest to oldest.