Racklog: Prolog-Style Logic Programming

Version 9.0.0.1

Dorai Sitaram

October 20, 2025

Adapted from Schelog by Dorai Sitaram for Racket by Dorai Sitaram, John Clements, and Jay McCarthy.

```
(require racklog)
```
package: `racklog`

Racklog is an *embedding* of Prolog-style logic programming in Racket. "Embedding" means you don't lose Racket: You can use Prolog-style and conventional Racket code fragments alongside each other. Racklog contains the full repertoire of Prolog features, including meta-logical and second-order ("set") predicates, leaving out only those features that could more easily and more efficiently be done with Racket subexpressions.

The Racklog implementation uses the approach to logic programming for Scheme described in Felleisen [mf:prolog] and Haynes [logick]. In contrast to earlier Lisp simulations of Prolog [campbell], which used explicit continuation arguments to store failure (backtrack) information, the Felleisen and Haynes model uses the implicit reified continuations of Scheme. In Racket these are provided by the operator `call-with-current-continuation` (aka `call/cc`). This allows Racklog to be an *embedding*, ie, logic programming is not built as a new language on top of Racket, but is used alongside Racket's other features. Both styles of programming may be mixed to any extent that a project needs.

The Racklog user does not need to know about the implementation mechanism or about `call/cc` and continuations to get on with the business of doing logic programming with Racklog.

This text is a gentle introduction to Racklog syntax and programming. It assumes a working knowledge of Racket and an awareness of, if not actual programming experience with, Prolog. If you need assistance for Prolog, you may consult [bratko, ok:prolog, aop] or many other excellent books and online documents available.

Contents

1 Simple Goals and Queries

Racklog objects are the same as Racket objects. However, there are two subsets of these objects that are of special interest to Racklog: *goals* and *predicates*. We will first look at some simple goals. §2 "Predicates" will introduce predicates and ways of making complex goals using predicates.

A goal is an object whose truth or falsity we can check. A goal that turns out to be true is said to succeed. A goal that turns out to be false is said to fail.

Two simple goals that are provided in Racklog are:

```
%true
%fail
```

The goal `%true` succeeds. The goal `%fail` always fails.

(The names of all Racklog primitive objects start with `%`. This is to avoid clashes with the names of conventional Racket objects of related meaning. User-created objects in Racklog are not required to follow this convention.)

A Racklog user can *query* a goal by wrapping it in a `%which`-form.

```
(%which () %true)
```

evaluates to `()`, indicating success, whereas:

```
(%which () %fail)
```

evaluates to `#f`, indicating failure.

Note 1: The second subexpression of the `%which`-form is the empty list `()`. Later (§2.3 "Solving Goals"), we will see `%which`es with other lists as the second subform.

Henceforth, we will use the notation:

```
> E
'F
```

to say that `E` *evaluates to* `F`. Thus,

```
> (%which () %true)
'()
```

2 Predicates

More interesting goals are created by applying a special kind of Racklog object called a *predicate* (or *relation*) to other Racklog objects. Racklog comes with some primitive predicates, such as the arithmetic operators `%=:=` and `%<`, standing for arithmetic "equal" and "less than" respectively. For example, the following are some goals involving these predicates:

```
> (%which () (%=:= 1 1))
'()
> (%which () (%< 1 2))
'()
> (%which () (%=:= 1 2))
#f
> (%which () (%< 1 1))
#f
```

Other arithmetic predicates are `%>` ("greater than"), `%<=` ("less than or equal"), `%>=` ("greater than or equal"), and `%=/=` ("not equal").

Racklog predicates are not to be confused with conventional Racket predicates (such as `<` and `=`). Racklog predicates, when applied to arguments, produce goals that may either succeed or fail. Racket predicates, when applied to arguments, yield a boolean value. Henceforth, we will use the term "predicate" to mean Racklog predicates. Conventional predicates will be explicitly called "Racket predicates".

2.1 Predicates Introducing Facts

Users can create their own predicates using the Racklog form `%rel`. For example, let's define the predicate `%knows`:

```
(define %knows
  (%rel ()
    [('Odysseus 'TeX)]
    [('Odysseus 'Racket)]
    [('Odysseus 'Prolog)]
    [('Odysseus 'Penelope)]
    [('Penelope 'TeX)]
    [('Penelope 'Prolog)]
    [('Penelope 'Odysseus)]
    [('Telemachus 'TeX)]
    [('Telemachus 'calculus)]))
```

The expression has the expected meaning. Each *clause* in the `%rel` establishes a *fact*: Odysseus knows TeX, Telemachus knows calculus, &c. In general, if we apply the predicate

to the arguments in any one of its clauses, we will get a successful goal. Thus, since `%knows` has a clause that reads `[('Odysseus 'TeX)]`, the goal `(%knows 'Odysseus 'TeX)` will be true.

We can now get answers for the following types of queries:

```
> (%which ()
    (%knows 'Odysseus 'TeX))
'()
> (%which ()
    (%knows 'Telemachus 'Racket))
#f
```

2.2 Predicates with Rules

Predicates can be more complicated than the above bald recitation of facts. The predicate clauses can be *rules*, eg,

```
(define %computer-literate
  (%rel (person)
    [(person)
      (%knows person 'TeX)
      (%knows person 'Racket)]
    [(person)
      (%knows person 'TeX)
      (%knows person 'Prolog)]))
```

This defines the predicate `%computer-literate` in terms of the predicate `%knows`. In effect, a person is defined as computer-literate if they know TeX and Racket, *or* TeX and Prolog.

Note that this use of `%rel` employs a local *logic variable* called `person`. In general, a `%rel`-expression can have a list of symbols as its second subform. These name new logic variables that can be used within the body of the `%rel`.

The following query can now be answered:

```
> (%which ()
    (%computer-literate 'Penelope))
'()
```

Since Penelope knows TeX and Prolog, she is computer-literate.

2.3 Solving Goals

The above queries are yes/no questions. Racklog programming allows more: We can formulate a goal with *uninstantiated* logic variables and then ask the querying process to provide, if possible, values for these variables that cause the goal to succeed. For instance, the query:

```
> (%which (what)
    (%knows 'Odysseus what))
'((what . TeX))
```

asks for an instantiation of the logic variable `what` that satisfies the goal `(%knows 'Odysseus what)`. In other words, we are asking, "What does Odysseus know?"

Note that this use of `%which` — like `%rel` in the definition of `%computer-literate` — uses a local logic variable, `what`. In general, the second subform of `%which` can be a list of local logic variables. The `%which`-query returns an answer that is a list of bindings, one for each logic variable mentioned in its second subform. Thus,

```
> (%which (what)
    (%knows 'Odysseus what))
'((what . TeX))
```

But that is not all that wily Odysseus knows. Racklog provides a zero-argument procedure ("thunk") called `%more` that *retries* the goal in the last `%which`-query for a different solution.

```
> (%more)
'((what . Racket))
```

We can keep pumping for more solutions:

```
> (%more)
'((what . Prolog))
> (%more)
'((what . Penelope))
> (%more)
#f
```

The final `#f` shows that there are no more solutions. This is because there are no more clauses in the `%knows` predicate that list Odysseus as knowing anything else.

2.4 Asserting Extra Clauses

We can add more clauses to a predicate after it has already been defined with a `%rel`. Racklog provides the `%assert!` form for this purpose. Eg,

```
(%assert! %knows ()
  [('Odysseus 'archery)])
```

tacks on a new clause at the end of the existing clauses of the `%knows` predicate. Now, the query:

```
> (%which (what)
    (%knows 'Odysseus what))
'((what . TeX))
```

gives TeX, Racket, Prolog, and Penelope, as before, but a subsequent `(%more)` yields a new result:

```
> (%more)
'((what . archery))
```

The Racklog form `%assert-after!` is similar to `%assert!` but adds clauses *before* any of the current clauses.

Both `%assert!` and `%assert-after!` assume that the variable they are adding to already names a predicate (presumably defined using `%rel`). In order to allow defining a predicate entirely through `%assert!`s, Racklog provides an empty predicate value `%empty-rel`. `%empty-rel` takes any number of arguments and always fails. A typical use of the `%empty-rel` and `%assert!` combination:

```
(define %parent %empty-rel)

(%assert! %parent ()
  [('Laertes 'Odysseus)])

(%assert! %parent ()
  [('Odysseus 'Telemachus)]
  [('Penelope 'Telemachus)])
```

(Racklog does not provide a predicate for *retracting* assertions, since we can keep track of older versions of predicates using conventional Racket features (`let` and `set!`).)

2.5 Local Variables

The local logic variables of `%rel`- and `%which`-expressions are in reality introduced by the Racklog syntactic form called `%let`. (`%rel` and `%which` are macros written using `%let`.)

`%let` introduces new lexically scoped logic variables. Supposing, instead of

```
> (%which (what)
    (%knows 'Odysseus what))
'((what . TeX))
```

we had asked

```
> (%let (what)
    (%which ()
      (%knows 'Odysseus what)))
'()
```

This query, too, succeeds five times, since Odysseus knows five things. However, `%which` emits bindings only for the local variables that *it* introduces. Thus, this query emits `()` five times before `(%more)` finally returns `#f`.

However, note that, like conventional Racket `let`, the body forms of `%let` are evaluated in turn with the last form in tail position. Thus, to combine multiple goals involving a new logical variable introduced by `%let`, it is necessary to wrap the body in an `%and` form.

For example:

```
> (%which (d)
    (%let (a p)
      (%and (%= p (cons 1 2))
            (%= p (cons a d)))))
'((d . 2))
```

whereas

```
> (%which (d)
    (%let (a p)
      (%= p (cons 1 2)) ; This goal is ignored
      (%= p (cons a d))))
'((d . _))
```

3 Using Conventional Racket Expressions in Racklog

The arguments of Racklog predicates can be any Racket objects. In particular, composite structures such as lists, vectors, strings, hash tables, etc can be used, as also Racket expressions using the full array of Racket's construction and decomposition operators. For instance, consider the following goal:

```
(%member x '(1 2 3))
```

Here, `%member` is a predicate, `x` is a logic variable, and `'(1 2 3)` is a structure. Given a suitably intuitive definition for `%member`, the above goal succeeds for `x` = `1`, `2`, and `3`.

Now to defining predicates like `%member`:

```
(define %member
  (%rel (x y xs)
    [(x (cons x xs))]
    [(x (cons y xs))
      (%member x xs)]))
```

Ie, `%member` is defined with three local variables: `x`, `y`, `xs`. It has two clauses, identifying the two ways of determining membership.

The first clause of `%member` states a fact: For any `x`, `x` is a member of a list whose head is also `x`.

The second clause of `%member` is a rule: `x` is a member of a list if we can show that it is a member of the *tail* of that list. In other words, the original `%member` goal is translated into a *sub*goal, which is also a `%member` goal.

Note that the variable `y` in the definition of `%member` occurs only once in the second clause. As such, it doesn't need you to make the effort of naming it. (Names help only in matching a second occurrence to a first.) Racklog lets you use the expression `(_)` to denote an anonymous variable. (Ie, `_` is a thunk that generates a fresh anonymous variable at each call.) The predicate `%member` can be rewritten as

```
(define %member
  (%rel (x xs)
    [(x (cons x (_)))]
    [(x (cons (_) xs))
      (%member x xs)]))
```

3.1 Constructors

We can use constructors — Racket procedures for creating structures — to simulate data types in Racklog. For instance, let's define a natural-number data-type where `0` denotes zero, and `(succ x)` denotes the natural number whose immediate predecessor is `x`. The constructor `succ` can be defined in Racket as:

```
(define succ
  (lambda (x)
    (vector 'succ x)))
```

Addition and multiplication can be defined as:

```
(define %add
  (%rel (x y z)
    [(0 y y)]
    [((succ x) y (succ z))
      (%add x y z)]))

(define %times
  (%rel (x y z z1)
    [(0 y 0)]
    [((succ x) y z)
     (%times x y z1)
     (%add y z1 z)]))
```

We can do a lot of arithmetic with this in place. For instance, the factorial predicate looks like:

```
(define %factorial
  (%rel (x y y1)
    [(0 (succ 0))]
    [((succ x) y)
      (%factorial x y1)
      (%times (succ x) y1 y)]))
```

3.2 `%is`

The above is a very inefficient way to do arithmetic, especially when the underlying language Racket offers excellent arithmetic facilities (including a comprehensive number "tower" and exact rational arithmetic). One problem with using Racket calculations directly in Racklog clauses is that the expressions used may contain logic variables that need to be dereferenced. Racklog provides the predicate `%is` that takes care of this. The goal

```
(%is X E)
```

unifies *X* with the value of *E* considered as a Racket expression. *E* can have logic variables, but usually they should at least be bound, as unbound variables may not be palatable values to the Racket operators used in *E*.

We can now directly use the numbers of Racket to write a more efficient `%factorial` predicate:

```
(define %factorial
  (%rel (x y x1 y1)
    [(0 1)]
    [(x y) (%is x1 (- x 1))
           (%factorial x1 y1)
           (%is y (* y1 x))]))
```

A price that this efficiency comes with is that we can use `%factorial` only with its first argument already instantiated. In many cases, this is not an unreasonable constraint. In fact, given this limitation, there is nothing to prevent us from using Racket's factorial directly:

```
(define %factorial
  (%rel (x y)
    [(x y)
     (%is y (racket-factorial
              x))]))
```

or better yet, "in-line" any calls to `%factorial` with `%is`-expressions calling `racket-factorial`, where the latter is defined in the usual manner:

```
(define racket-factorial
  (lambda (n)
    (if (= n 0) 1
        (* n (racket-factorial
               (- n 1))))))
```

3.3 Lexical Scoping

One can use Racket's lexical scoping to enhance predicate definition. Here is a list-reversal predicate defined using a hidden auxiliary predicate:

```
(define %reverse
  (letrec
    ([revaux
```

```
    (%rel (x y z w)
      [('() y y)]
      [((cons x y) z w)
        (revaux y
          (cons x z) w)])])
(%rel (x y)
  [(x y) (revaux x '() y)]))))
```

`(revaux X Y Z)` uses `Y` as an accumulator for reversing `X` into `Z`. (`Y` starts out as `()`. Each head of `X` is `cons`ed on to `Y`. Finally, when `X` has wound down to `()`, `Y` contains the reversed list and can be returned as `Z`.)

`revaux` is used purely as a helper predicate for `%reverse`, and so it can be concealed within a lexical contour. We use `letrec` instead of `let` because `revaux` is a recursive procedure.

3.4 Type Predicates

Racklog provides a couple of predicates that let the user probe the type of objects.

The goal

```
(%constant X)
```

succeeds if `X` is an *atomic* object.

The predicate `%compound`, the negation of `%constant`, checks if its argument is not an atomic object.

The above are merely the logic-programming equivalents of corresponding Racket predicates. Users can use the predicate `%is` and Racket predicates to write more type checks in Racklog. Thus, to test if `X` is a string, the following goal could be used:

```
(%is #t (string? X))
```

User-defined Racket predicates, in addition to primitive Racket predicates, can be thus imported.

4 Backtracking

It is helpful to go into the following evaluation (§2.2 “Predicates with Rules”) in a little detail:

```
(%which ()
  (%computer-literate 'Penelope))
```

The starting goal is:

```
G0 = (%computer-literate Penelope)
```

(I’ve taken out the quote because `Penelope` is the result of evaluating `'Penelope`.)

Racklog tries to match this with the head of the first clause of `%computer-literate`. It succeeds, generating a binding `[person . Penelope]`.

But this means it now has two new goals — *subgoals* — to solve. These are the goals in the body of the matching clause, with the logic variables substituted by their instantiations:

```
G1 = (%knows Penelope TeX)
G2 = (%knows Penelope Racket)
```

For `G1`, Racklog attempts matches with the clauses of `%knows`, and succeeds at the fifth try. (There are no subgoals in this case, because the bodies of these “fact” clauses are empty, in contrast to the “rule” clauses of `%computer-literate`.) Racklog then tries to solve `G2` against the clauses of `%knows`, and since there is no clause stating that Penelope knows Racket, it fails.

All is not lost though. Racklog now *backtracks* to the goal that was solved just before, viz., `G1`. It *retries* `G1`, ie, tries to solve it in a different way. This entails searching down the previously unconsidered `%knows` clauses for `G1`, ie, the sixth onwards. Obviously, Racklog fails again, because the fact that Penelope knows TeX occurs only once.

Racklog now backtracks to the goal before `G1`, ie, `G0`. We abandon the current successful match with the first clause-head of `%computer-literate`, and try the next clause-head. Racklog succeeds, again producing a binding `[person . Penelope]`, and two new subgoals:

```
G3 = (%knows Penelope TeX)
G4 = (%knows Penelope Prolog)
```

It is now easy to trace that Racklog finds both `G3` and `G4` to be true. Since both of `G0`’s subgoals are true, `G0` is itself considered true. And this is what Racklog reports. The interested reader can now trace why the following query has a different denouement:

```
> (%which ()
    (%computer-literate 'Telemachus))
#f
```

5 Unification

When we say that a goal matches with a clause-head, we mean that the predicate and argument positions line up. Before making this comparison, Racklog dereferences all already bound logic variables. The resulting structures are then compared to see if they are recursively identical. Thus, `1` unifies with `1`, and `(list 1 2)` with `'(1 2)`; but `1` and `2` do not unify, and neither do `'(1 2)` and `'(1 3)`.

In general, there could be quite a few uninstantiated logic variables in the compared objects. Unification will then endeavor to find the most natural way of binding these variables so that we arrive at structurally identical objects. Thus, `(list x 1)`, where `x` is an unbound logic variable, unifies with `'(0 1)`, producing the binding `[x 0]`.

Unification is thus a goal, and Racklog makes the unification predicate available to the user as `%=`. Eg,

```
> (%which (x)
    (%= (list x 1) '(0 1)))
'((x . 0))
```

Racklog also provides the predicate `%/=`, the *negation* of `%=`. `(%/= X Y)` succeeds if and only if `X` does *not* unify with `Y`.

Unification goals constitute the basic subgoals that all Racklog goals devolve to. A goal succeeds because all the eventual unification subgoals that it decomposes to in at least one of its subgoal-branching succeeded. It fails because every possible subgoal-branching was thwarted by the failure of a crucial unification subgoal.

Going back to the example in §4 "Backtracking", the goal `(%computer-literate 'Penelope)` succeeds because (a) it unified with `(%computer-literate person)`; and then (b) with the binding `[person . Penelope]` in place, `(%knows person 'TeX)` unified with `(%knows 'Penelope 'TeX)` and `(%knows person 'Prolog)` unified with `(%knows 'Penelope 'Prolog)`.

In contrast, the goal `(%computer-literate 'Telemachus)` fails because, with `[person . Telemachus]`, the subgoals `(%knows person 'Racket)` and `(%knows person 'Prolog)` have no facts they can unify with.

5.1 The Occurs Check

A robust unification algorithm uses the *occurs check*, which ensures that a logic variable isn't bound to a structure that contains itself. Not performing the check can cause the unification to go into an infinite loop in some cases. On the other hand, performing the occurs check greatly increases the time taken by unification, even in cases that wouldn't require the check.

Racklog uses the global parameter `use-occurs-check?` to decide whether to use the occurs check. By default, this variable is `#f`, ie, Racklog disables the occurs check. To enable the check,

```
(use-occurs-check? #t)
```

6 Conjunctions and Disjunctions

Goals may be combined using the forms %and and %or to form compound goals. (For %not, see §8.2 "Negation as Failure".) Eg,

```
> (%which (x)
    (%and (%member x '(1 2 3))
          (%< x 3)))
'((x . 1))
```

gives solutions for x that satisfy both the argument goals of the %and. Ie, x should both be a member of '(1 2 3) *and* be less than 3. Typing (%more) gives another solution:

```
> (%more)
'((x . 2))
> (%more)
#f
```

There are no more solutions, because [x 3] satisfies the first but not the second goal.

Similarly, the query

```
> (%which (x)
    (%or (%member x '(1 2 3))
         (%member x '(3 4 5))))
'((x . 1))
```

lists all x that are members of either list.

```
> (%more)
'((x . 2))
> (%more)
'((x . 3))
> (%more)
'((x . 3))
> (%more)
'((x . 4))
> (%more)
'((x . 5))
```

(Yes, ([x 3]) is listed twice.)

We can rewrite the predicate %computer-literate from §2.2 "Predicates with Rules" using %and and %or:

```
(define %computer-literate
  (%rel (person)
    [(person)
     (%or
       (%and (%knows person
               'TeX)
             (%knows person
               'Racket))
       (%and (%knows person
               'TeX)
             (%knows person
               'Prolog)))]))
```

Or, more succinctly:

```
(define %computer-literate
  (%rel (person)
    [(person)
      (%and (%knows person
              'TeX)
        (%or (%knows person
               'Racket)
             (%knows person
               'Prolog)))]))
```

We can even dispense with the `%rel` altogether:

```
(define %computer-literate
  (lambda (person)
    (%and (%knows person
            'TeX)
      (%or (%knows person
             'Racket)
        (%knows person
          'Prolog)))))
```

This last looks like a conventional Racket predicate definition, and is arguably the most readable format for a Racket programmer.

7 Manipulating Racklog Variables

Racklog provides special predicates for probing logic variables, without risking their getting bound.

7.1 Checking for Variables

The goal

```
(%== X Y)
```

succeeds if `X` and `Y` are *identical* objects. This is not quite the unification predicate `%=`, for `%==` doesn't touch unbound objects the way `%=` does. Eg, `%==` will not equate an unbound logic variable with a bound one, nor will it equate two unbound logic variables unless they are the *same* variable.

The predicate `%/==` is the negation of `%==`.

The goal

```
(%var X)
```

succeeds if `X` isn't completely bound — ie, it has at least one unbound logic variable in its innards.

The predicate `%nonvar` is the negation of `%var`.

7.2 Preserving Variables

Racklog lets the user protect a term with variables from unification by allowing that term to be treated as a (completely) bound object. The predicates provided for this purpose are `%freeze`, `%melt`, `%melt-new`, and `%copy`.

The goal

```
(%freeze S F)
```

unifies `F` to the frozen version of `S`. Any lack of bindings in `S` are preserved no matter how much you toss `F` about. Frozen variables will only unify with themselves. For example, in the following query, even though `x` and `y` are unbound, their frozen counterparts `a` and `b` will not unify with each other:

```
> (%which (x y frozen a b)
    (%freeze (list x y) (list a b))
    (%= a b))
#f
```

However, we can unify a frozen variable with itself:

```
> (%which (x frozen y z)
    (%freeze (list x x) (list y z))
    (%= y z))
'((x . _) (frozen . _) (y . #<frozen>) (z . #<frozen>))
```

The goal

```
(%melt F S)
```

retrieves the object frozen in *F* into *S*; the variables in *S* will be identical to those in the original object. For example:

```
> (%which (x frozen melted)
    (%freeze (list x x) frozen)
    (%melt frozen melted)
    (%= melted (list 2 2)))
'((x . 2) (frozen #<frozen> #<frozen>) (melted 2 2))
```

The goal

```
(%melt-new F S)
```

is similar to %melt, except that when *S* is made, the unbound variables in *F* are replaced by brand-new unbound variables, respecting the existing equivalences between them, for example:

```
> (%which (x y frozen a b c)
    (%freeze (list x x y) frozen)
    (%melt-new frozen (list a b c))
    (%= a 'red)
    (%= c 'green))
'((x . _)
  (y . _)
  (frozen #<frozen> #<frozen> #<frozen>)
  (a . red)
  (b . red)
  (c . green))
```

The goal

```
(%copy S C)
```

is an abbreviation for `(%freeze S F)` followed by `(%melt-new F C)`.

8 The Cut (!)

The cut (called !) is a special goal that is used to prune backtracking options. Like the %true goal, the cut goal too succeeds, when accosted by the Racklog subgoaling engine. However, when a further subgoal down the line fails, and time comes to retry the cut goal, Racklog will refuse to try alternate clauses for the predicate in whose definition the cut occurs. In other words, the cut causes Racklog to commit to all the decisions made from the time that the predicate was selected to match a subgoal till the time the cut was satisfied.

For example, consider again the %factorial predicate, as defined in §3.2 "%is":

```
(define %factorial
  (%rel (x y x1 y1)
    [(0 1)]
    [(x y) (%is x1 (- x 1))
           (%factorial x1 y1)
           (%is y (* y1 x))]))
```

Clearly,

```
> (%which ()
    (%factorial 0 1))
'()
> (%which (n)
    (%factorial 0 n))
'((n . 1))
```

But what if we asked for (%more) for either query? Backtracking will try the second clause of %factorial, and sure enough the clause-head unifies, producing binding [x . 0]. We now get three subgoals. Solving the first, we get [x1 . -1], and then we have to solve (%factorial -1 y1). It is easy to see there is no end to this, as we fruitlessly try to get the factorials of numbers that get more and more negative.

If we placed a cut at the first clause:

```
...
[(0 1) !]
...
```

the attempt to find more solutions for (%factorial 0 1) is nipped in the bud.

Calling %factorial with a *negative* number would still cause an infinite loop. To take care of that problem as well, we use another cut:

```
(define %factorial
```

```
  (%rel (x y x1 y1)
    [(0 1) !]
    [(x y) (%< x 0) ! %fail]
    [(x y) (%is x1 (- x 1))
           (%factorial x1 y1)
           (%is y (* y1 x))]))
```

```
> (%which ()
    (%factorial 0 1))
'()
> (%more)
#f
> (%which ()
    (%factorial -1 1))
#f
```

Using *raw* cuts as above can get very confusing. For this reason, it is advisable to use it hidden away in well-understood abstractions. Two such common abstractions are the conditional and negation.

8.1 Conditional Goals

An "if ... then ... else ..." predicate can be defined as follows

```
(define %if-then-else
  (%rel (p q r)
    [(p q r) p ! q]
    [(p q r) r]))
```

(Note that for the first time we have predicate arguments that are themselves goals.)

Consider the goal

```
G0 = (%if-then-else Gbool Gthen Gelse)
```

We first unify `G0` with the first clause-head, giving `[p . Gbool]`, `[q . Gthen]`, `[r . Gelse]`. `Gbool` can now either succeed or fail.

Case 1: If `Gbool` fails, backtracking will cause the `G0` to unify with the second clause-head. `r` is bound to `Gelse`, and so `Gelse` is tried, as expected.

Case 2: If `Gbool` succeeds, the cut commits to this clause of the `%if-then-else`. We now try `Gthen`. If `Gthen` should now fail — or even if we simply retry for more solutions —

we are guaranteed that the second clause-head will not be tried. If it were not for the cut, `G0` would attempt to unify with the second clause-head, which will of course succeed, and `Gelse` *will* be tried.

8.2 Negation as Failure

Another common abstraction using the cut is *negation*. The negation of goal `G` is defined as `(%not G)`, where the predicate `%not` is defined as follows:

```
(define %not
  (%rel (g)
    [(g) g ! %fail]
    [(g) %true]))
```

Thus, `g`'s negation is deemed a failure if `g` succeeds, and a success if `g` fails. This is of course confusing goal failure with falsity. In some cases, this view of negation is actually helpful.

9 Set Predicates

The goal

```
(%bag-of X G Bag)
```

unifies with *Bag* the list of all instantiations of *X* for which *G* succeeds. Thus, the following query asks for all the things known — ie, the collection of things such that someone knows them:

```
> (%which (things-known)
    (%let (someone x)
      (%bag-of x (%knows someone x)
        things-known)))
'((things-known
   TeX
   Racket
   Prolog
   Penelope
   TeX
   Prolog
   Odysseus
   TeX
   calculus
   archery))
```

This is the only solution for this goal:

```
> (%more)
#f
```

Note that some things — eg, TeX — are enumerated more than once. This is because more than one person knows TeX. To remove duplicates, use the predicate `%set-of` instead of `%bag-of`:

```
> (%which (things-known)
    (%let (someone x)
      (%set-of x (%knows someone x)
        things-known)))
'((things-known TeX Racket Prolog Penelope Odysseus calculus
archery))
```

In the above, the free variable *someone* in the `%knows`-goal is used as if it were existentially quantified. In contrast, Prolog's versions of `%bag-of` and `%set-of` fix it for each solution of

the set-predicate goal. We can do it too with some additional syntax that identifies the free variable. Eg,

```
> (%which (someone things-known)
    (%let (x)
      (%bag-of x
        (%free-vars (someone)
          (%knows someone x))
        things-known)))
'((someone . Odysseus) (things-known TeX Racket Prolog Penelope))
```

The bag of things known by *one* someone is returned. That someone is Odysseus. The query can be retried for more solutions, each listing the things known by a different someone:

```
> (%more)
'((someone . Penelope) (things-known TeX Prolog Odysseus))
> (%more)
'((someone . Telemachus) (things-known TeX calculus))
> (%more)
'((someone . Odysseus) (things-known archery))
> (%more)
#f
```

Racklog also provides two variants of these set predicates, viz., `%bag-of-1` and `%set-of-1`. These act like `%bag-of` and `%set-of` but fail if the resulting bag or set is empty.

10 Higher-order Predicates

Logic variables which contain predicates may be used as the operator in predicate expressions:

```
> (%which ()
    (%let (p)
      (%and (%= p %knows)
            (p 'Odysseus 'TeX))))
'()
```

First the logic variable `p` is unified with the predicate `%knows`. In the expression

```
(p 'Odysseus 'TeX)
```

`p` is replaced by its value `%knows` to become

```
(%knows 'Odysseus 'TeX)
```

which succeeds.

This allows us to reason about predicates themselves. For example:

```
> (%which (p)
    (%member p (list %knows %parent))
    (p 'Odysseus 'Penelope))
'((p . #<procedure:relation>))
> (%more)
#f
```

Here we test which of the predicates `%knows` and `%parent` succeed when given the arguments `'Odysseus` and `'Penelope`.

The goal `(%knows 'Odysseus 'Penelope)` succeeds, but `(%parent 'Odysseus 'Penelope)` fails. Hence the only possible value for `p` is `%knows`.

However, logic variables used as a predicate must be instantiated. Since the set of defined predicates is not enumerable by Racklog, an unbound query will fail:

```
> (%which (p) (p 'Odysseus 'Penelope))
#f
```

We can define a higher-order predicate which tests for unary predicates that succeed with `'Odysseus` as their argument:

```
(define (%odyssean p)
  (p 'Odysseus))
```

For example:

```
> (%which () (%odyssean %computer-literate))
'()
```

This succeeds because `(%computer-literate 'Odysseus)` succeeds.

```
> (%which () (%odyssean %compound))
#f
```

This fails because `(%compound 'Odysseus)` fails.

This also works if the predicate argument is a logic variable:

```
> (%which (p)
    (%member p (list %computer-literate %compound))
    (%odyssean p))
'((p . #<procedure:%computer-literate>))
```

Compare this with the example above.

Racklog also provides two predicates for defining relations involving arbitrary predicates.

10.1 %apply

The `%apply` predicate is analogous to convential Racket `apply`.

The goal

```
(%apply P L)
```

succeeds if `L` is a list with elements `E`, ..., and if `P` is a predicate that accepts as many arguments as there are `E`s, and if the goal `(P E ...)` succeeds. For example:

```
> (%which () (%apply %knows '(Odysseus TeX)))
'()
```

In this case, the goal

```
(%apply %knows '(Odysseus TeX))
```

is equivalent to

```
(%knows 'Odysseus 'TeX)
```

The list argument to %apply must be sufficiently instantiated to determine its length. The following goals succeed:

```
> (%which () (%apply %knows (list 'Odysseus 'TeX)))
'()
> (%which (X) (%apply %knows (list X 'TeX)))
'((X . Odysseus))
```

but it is not possible to use %apply with a list of unknown length:

```
> (%which (X Y) (%apply %knows (cons X Y)))
#f
```

10.2 %andmap

The %andmap predicate is analogous to convential Racket andmap.

The goal

```
(%andmap P L ...+)
```

succeeds if all the Ls are lists of equal length, and the goal (P E ...) succeeds for each set of elements E, ... of the Ls. For example:

```
> (%which () (%andmap %knows '(Odysseus Penelope) '(TeX Prolog)))
'()
```

In this case, the goal

```
(%andmap %knows '(Odysseus Penelope) '(TeX Prolog))
```

is equivalent to

```
(%and (%knows 'Odysseus 'TeX)
      (%knows 'Penelope 'Prolog))
```

11 Racklog Module Language

`#lang racklog` package: `racklog`

This module language accepts the syntax of Datalog (except clauses need not be safe) and compiles each predicate to a relation.

The accepted syntax is available in the §1 “Datalog Module Language” documentation.

12 Glossary of Racklog Primitives

12.1 Racket Predicates

```
(logic-var? x) → boolean?
  x : any/c
```

Identifies a logic variable.

```
(atomic-struct? x) → boolean?
  x : any/c
```

Identifies structures that the `(current-inspector)` cannot inspect.

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

Identifies atomic values that may appear in Racklog programs. Equivalent to the contract `(or/c boolean? number? string? bytes? char? symbol? regexp? pregexp? byte-regexp? byte-pregexp? keyword? null? procedure? void? set? atomic-struct?)`.

```
(compound-struct? x) → boolean?
  x : any/c
```

Identifies structures that the `(current-inspector)` can inspect.

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

Identifies compound values that may appear in Racklog programs. Equivalent to the contract `(or/c pair? vector? mpair? box? hash? compound-struct?)`.

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

Identifies values that may appear in Racklog programs. Essentially either an `atom?`, `logic-var?`, or `compound?` that contains `unifiable?`s.

```
(answer-value? x) → boolean?
  x : any/c
```

Identifies values that may appear in `answer?`. Essentially `unifiable?`s that do not contain `logic-var?`s.

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

Identifies answers returned by %more and %which. Equivalent to the contract (or/c false/c (listof (cons/c symbol? answer-value?))).

```
goal/c : contract?
```

A contract for goals.

12.2 User Interface

```
(%which (V ...) G ...)

  V : identifier?
  G : goal/c
```

Returns an answer? of the variables V, ..., that satisfies all of G, ... If G, ..., cannot be satisfied, returns #f. Calling the thunk %more produces more instantiations, if available.

```
(%more) → answer?
```

The thunk %more produces more instantiations of the variables in the most recent %which-form that satisfy the goals in that %which-form. If no more solutions can be found, %more returns #f.

```
(%find-all (V ...) G ...)

  V : identifier?
  G : goal/c
```

Like (list (%which (V ...) G ...) (%more) ...) with as many (%more)s as there are answers. (This will not terminate if there are an infinite number of answers.)

12.3 Relations

```
(%rel (V ...) clause ...)

clause = [(E ...) G ...]
```

```
  V : identifier?
  E : expression?
  G : goal/c
```

Returns a predicate function. Each clause `C` signifies that the goal created by applying the predicate object to anything that matches `(E ...)` is deemed to succeed if all the goals `G`, ..., can, in their turn, be shown to succeed. The variables `V`, ..., are local logic variables for `clause`, ...

```
(%empty-rel E ...) → goal/c
  E : unifiable?
```

The goal `(%empty-rel E ...)` always fails. The *value* `%empty-rel` is used as a starting value for predicates that can later be enhanced with `%assert!` and `%assert-after!`.

```
(%assert! Pname (V ...) clause ...)

  Pname : identifier?
  V : identifier?
```

Adds the clauses `clauses`, ..., to the *end* of the predicate that is the value of the Racket variable `Pname`. The variables `V`, ..., are local logic variables for `clause`, ...

```
(%assert-after! Pname (V ...) clause ...)

  Pname : identifier?
  V : identifier?
```

Like `%assert!`, but adds the new clauses to the *front* of the existing predicate.

12.4 Racklog Variables

```
(_) → logic-var?
```

A thunk that produces a new logic variable. Can be used in situations where we want a logic variable but don't want to name it. (`%let`, in contrast, introduces new lexical names for the logic variables it creates.)

```
(%let (V ...) expr ...)

  V : identifier?
```

Introduces `V`, ..., as lexically scoped logic variables to be used in `expr`, ...

12.5 Cut

```
(%cut-delimiter . any)
```

Introduces a cut point. See §8 “The Cut (`!`)”.

```
!
```

The cut goal, see §8 “The Cut (`!`)”.

May only be used syntactically inside `%cut-delimiter` or `%rel`.

12.6 Racklog Operators

```
%fail : goal/c
```

The goal `%fail` always fails.

```
%true : goal/c
```

The goal `%true` succeeds. Fails on retry.

```
(%repeat) → goal/c
```

The goal `(%repeat)` always succeeds (even on retries). Useful for failure-driven loops.

```
(%and G ...)

  G : goal/c
```

The goal `(%and G ...)` succeeds if all the goals `G`, ..., succeed.

```
(%or G ...)

  G : goal/c
```

The goal `(%or G ...)` succeeds if one of `G`, ..., tried in that order, succeeds.

```
(%not G) → goal/c
  G : goal/c
```

The goal `(%not G)` succeeds if `G` fails.

```
(%if-then-else G1 G2 G3) → goal/c
  G1 : goal/c
  G2 : goal/c
  G3 : goal/c
```

The goal (%if-then-else *G1* *G2* *G3*) tries *G1* first: if it succeeds, tries *G2*; if not, tries *G3*.

12.7 Unification

```
(%= E1 E2) → goal/c
  E1 : unifiable?
  E2 : unifiable?
```

The goal (%= *E1* *E2*) succeeds if *E1* can be unified with *E2*. Any resulting bindings for logic variables are kept.

```
(%/= E1 E2) → goal/c
  E1 : unifiable?
  E2 : unifiable?
```

%/= is the negation of %=. The goal (%/= *E1* *E2*) succeeds if *E1* can not be unified with *E2*.

```
(%== E1 E2) → goal/c
  E1 : unifiable?
  E2 : unifiable?
```

The goal (%== *E1* *E2*) succeeds if *E1* is *identical* to *E2*. They should be structurally equal. If containing logic variables, they should have the same variables in the same position. Unlike a %=-call, this goal will not bind any logic variables.

```
(%/== E1 E2) → goal/c
  E1 : unifiable?
  E2 : unifiable?
```

%/== is the negation of %==. The goal (%/== *E1* *E2*) succeeds if *E1* and *E2* are not identical.

```
(%is E1 E2)
```

The goal (%is *E1* *E2*) unifies with *E1* the result of evaluating *E2* as a Racket expression. *E2* may contain logic variables, which are dereferenced automatically. Fails if *E2* contains unbound logic variables.

```
(use-occurs-check?) → boolean?
(use-occurs-check? on?) → void?
  on? : boolean?
```

If this is false (the default), Racklog's unification will not use the occurs check. If it is true, the occurs check is enabled.

12.8 Numeric Predicates

```
(%< E1 E2) → goal/c
  E1 : unifiable?
  E2 : unifiable?
```

The goal `(%< E1 E2)` succeeds if *E1* and *E2* are bound to numbers and *E1* is less than *E2*.

```
(%<= E1 E2) → goal/c
  E1 : unifiable?
  E2 : unifiable?
```

The goal `(%<= E1 E2)` succeeds if *E1* and *E2* are bound to numbers and *E1* is less than or equal to *E2*.

```
(%=/= E1 E2) → goal/c
  E1 : unifiable?
  E2 : unifiable?
```

The goal `(%=/= E1 E2)` succeeds if *E1* and *E2* are bound to numbers and *E1* is not equal to *E2*.

```
(%=:= E1 E2) → goal/c
  E1 : unifiable?
  E2 : unifiable?
```

The goal `(%=:= E1 E2)` succeeds if *E1* and *E2* are bound to numbers and *E1* is equal to *E2*.

```
(%> E1 E2) → goal/c
  E1 : unifiable?
  E2 : unifiable?
```

The goal `(%> E1 E2)` succeeds if *E1* and *E2* are bound to numbers and *E1* is greater than *E2*.

```
(%>= E1 E2) → goal/c
  E1 : unifiable?
  E2 : unifiable?
```

The goal `(%>= E1 E2)` succeeds if *E1* and *E2* are bound to numbers and *E1* is greater than or equal to *E2*.

12.9 List Predicates

```
(%append E1 E2 E3) → goal/c
  E1 : unifiable?
  E2 : unifiable?
  E3 : unifiable?
```

The goal `(%append E1 E2 E3)` succeeds if *E3* is unifiable with the list obtained by appending *E1* and *E2*.

```
(%member E1 E2) → goal/c
  E1 : unifiable?
  E2 : unifiable?
```

The goal `(%member E1 E2)` succeeds if *E1* is a member of the list in *E2*.

12.10 Set Predicates

```
(%set-of E1 G E2) → goal/c
  E1 : unifiable?
  G : goal/c
  E2 : unifiable?
```

The goal `(%set-of E1 G E2)` unifies with *E2* the *set* of all the instantiations of *E1* for which goal *G* succeeds.

```
(%set-of-1 E1 G E2) → goal/c
  E1 : unifiable?
  G : goal/c
  E2 : unifiable?
```

Similar to `%set-of`, but fails if the set is empty.

```
(%bag-of E1 G E2) → goal/c
  E1 : unifiable?
  G : goal/c
  E2 : unifiable?
```

The goal `(%bag-of E1 G E2)` unifies with `E2` the *bag* (multiset) of all the instantiations of `E1` for which goal `G` succeeds.

```
(%bag-of-1 E1 G E2) → goal/c
  E1 : unifiable?
  G : goal/c
  E2 : unifiable?
```

Similar to `%bag-of`, but fails if the bag is empty.

```
(%free-vars (V ...) G)

  V : identifier?
  G : goal/c
```

Identifies the occurrences of the variables `V`, ..., in goal `G` as free. It is used to avoid existential quantification in calls to set predicates (`%bag-of`, `%set-of`, &c.).

12.11 Racklog Predicates

```
(%compound E) → goal/c
  E : unifiable?
```

The goal `(%compound E)` succeeds if `E` is a compound value.

```
(%constant E) → goal/c
  E : unifiable?
```

The goal `(%constant E)` succeeds if `E` is an atomic value.

```
(%var E) → goal/c
  E : unifiable?
```

The goal `(%var E)` succeeds if `E` is not completely instantiated, ie, it has at least one unbound variable in it.

```
(%nonvar E) → goal/c
  E : unifiable?
```

`%nonvar` is the negation of `%var`. The goal `(%nonvar E)` succeeds if `E` is completely instantiated, ie, it has no unbound variable in it.

12.12 Higher-order Predicates

```
(%apply P L) → goal/c
  P : unifiable?
  L : unifiable?
```

The goal `(%apply P L)` succeeds if `L` is a list with elements `E`, ..., and if `P` is a predicate accepting as many arguments as there are `E`s, and if the goal `(P E ...)` succeeds.

The goal will fail if `L` is not sufficiently instantiated to determine its length.

For example, the goal

```
(%apply %= (list 1 X))
```

is equivalent to

```
(%= 1 X)
```

which succeeds if `X` can be unified with `1`.

```
(%andmap P L ...+) → goal/c
  P : unifiable?
  L : unifiable?
```

The goal `(%andmap P L ...)` succeeds if all the values `L`, ..., are lists of equal length, and if the goal `(P E ...)` succeeds for each set of values `E`, ..., taken in turn from each of the lists `L`, ...

As an example, in particular the goal `(%andmap %<= '(1 2 3) '(4 5 6))` is equivalent to

```
(%and (%<= 1 4)
      (%<= 2 5)
      (%<= 3 6))
```

12.13 Racklog Variable Manipulation

```
(%freeze S F) → goal/c
  S : unifiable?
  F : unifiable?
```

The goal `(%freeze S F)` unifies with `F` a new frozen version of the structure in `S`. Freezing implies that all the unbound variables are preserved. `F` can henceforth be used as *bound* object with no fear of its variables getting bound by unification.

```
(%melt F S) → goal/c
  F : unifiable?
  S : unifiable?
```

The goal `(%melt F S)` unifies `S` with the thawed (original) form of the frozen structure in `F`.

```
(%melt-new F S) → goal/c
  F : unifiable?
  S : unifiable?
```

The goal `(%melt-new F S)` unifies `S` with a thawed *copy* of the frozen structure in `F`. This means new logic variables are used for unbound logic variables in `F`.

```
(%copy F S) → goal/c
  F : unifiable?
  S : unifiable?
```

The goal `(%copy F S)` unifies with `S` a copy of the frozen structure in `F`.

Bibliography

[aop] Leon Sterling and Ehud Shapiro, *The Art of Prolog, 2nd Edition*, MIT Press, 1994. `http://mitpress.mit.edu/books/art-prolog`

[bratko] Ivan Bratko, *Prolog Programming for Artificial Intelligence*, Addison-Wesley, 1986.

[campbell] J A Campbell (editor), *Implementations of Prolog*, Ellis Horwood, 1984.

[ok:prolog] Richard A O'Keefe, *The Craft of Prolog*, MIT Press, 1990. `http://mitpress.mit.edu/books/craft-prolog`

[logick] Christopher T Haynes, "Logic continuations," J Logic Program, vol 4, 157–176, 1987.

[mf:prolog] Matthias Felleisen, "Transliterating Prolog into Scheme," Indiana U Comp Sci Dept Tech Report #182, 1985.