Login

Assignment 13: Further Logic Programming

> Mathematics is concerned only with the enumeration and comparison of relations.

Preliminaries

This week's assignment is a continuation of our work with miniKanren. So you should need to `(require "mk.rkt")` and `(require "numbers.rkt")`, as these files contain miniKanren and the relational arithmetic system. You should be able to test your solutions with the attached a13-student-tests.rkt

Make sure they pass the included tests, but as usual you should also write your own.

NOTE: Remember that, in general, the order of answers in a miniKanren program is irrelevant. Its not mandatory that your answers appear in the same order as below, but do try and check that your relations don't produce incorrect answers when run with fresh logic variables.

NOTE: Also recall that, while generally you want your recursions last, if some of your calls are serious, non-tail calls you might have to change the ordering.

Assignment

1. `listo`

As a warm-up, first, write the `listo` miniKanren relation. This is intended to operate similarly to Racket's `list?`

```
> (run 1 q (listo '(a b c d e)))
(_0)

> (run 1 q (listo '(a b c d . e)))
()

> (run 4 q (listo q))
(() (_0) (_0 _1) (_0 _1 _2))

> (run 4 q (listo `(a b ,q)))
(_0)
```

For the next two questions, we get a chance to play with the miniKanren relational arithmetic system.

2. `facto` Build a relation that holds between two backward-binary numbers whenever the second is the factorial of the first. Use the backwards-binary arithmetic primitives found in `numbers.rkt`, and described in chapters 7 & 8 of *The Reasoned Schemer*.

```
> (run 1 q (facto  q '(0 0 0 1 1 1 1)))
((1 0 1))

> (run 1 q (facto (build-num 5) q))
((0 0 0 1 1 1 1))

> (run 6 q (fresh (n1 n2) (facto n1 n2) (== `(,n1 ,n2) q)))
((() (1))
 ((1) (1))
 ((0 1) (0 1))
 ((1 1) (0 1 1))
 ((0 0 1) (0 0 0 1 1))
 ((1 0 1) (0 0 0 1 1 1 1)))
```

3. `fibs` Plain old fibonacci has had its day. Let's do something more fun. And in the process, learn a technique sometimes of use when building miniKanren relations. Here, we're going to take a function `fibs` which takes one input and returns two values, and make a three-place relation `fibso`. Check out the definition of `fibs`:

```
> (define fibs
    (lambda (n)
      (cond
        ((eqv? n 0) (values 1 1))
        (else
         (let ((n- (- n 1)))
           (let-values (((u v) (fibs n-)))
             (let ((u+v (+ u v)))
               (values v u+v)))))))))
> (fibs 0)
1
1
> (fibs 1)
1
2
> (fibs 2)
2
3
> (fibs 3)
3
5
```

We use `values`, which allows a Racket function to return multiple output values. This function returns the nth and n+1th fibonacci number. Since `fibs` returns two values, we need to use `let-values` when let binding the recursive call. Here, the first answer is bound to `u` and the second is bound to `v`. When writing `fibso` you will need to use two "output variables" in your relation, rather than just one like we usually do.

```
> (run 4 q
    (fresh (n o1 o2)
      (== q `(,n ,o1 ,o2))
      (fibso n o1 o2)))
((() (1) (1))
 ((1) (1) (0 1))
 ((0 1) (0 1) (1 1))
 ((1 1) (1 1) (1 0 1)))
> (run 1 q
    (fresh (n o1)
      (== q `(,n ,o1))
      (fibso n o1 (build-num 5))))
```

```
(((1 1) (1 1)))
> (run 1 q
    (fresh (n o2)
      (== q `(,n ,o2))
      (fibso n (build-num 5) o2)))
(((0 0 1) (0 0 0 1)))
```

4. fo-lavo

In class, Dan demonstrated a relational interpreter written in miniKanren which generates quines. What we have here is a whole different way of making it “run backwards”. You might have seen the interpreter `fo-lav` on Assignment 3. Drop `1bus`, `?orez`, `fi`, and `*`, and instead add `etouq` and `tsil`. Your job is to write a relation `fo-lavo`, which is the transformation of this provided interpreter into miniKanren. You should be able to use your Racket implementation of `fo-lav` and the `val-ofo` from class to guide you a good deal of the way. Your relation should be able to produce `fo-lav` quines.

```
;; testing etouq
> (run 1 q (fo-lavo '(etouq etouq) '() '() q))
(etouq)
;; testing tsil
> (run 1 q (fo-lavo '((cat etouq) tsil) '() '() q))
((cat))
> (run 1 q (fo-lavo '((cat etouq) . tsil) '() '() q))
()
> (run 1 q (fo-lavo '((dog etouq) (cat etouq) tsil) '() '() q))
((dog cat))
;; fo-lav quines
> (run 1 (q) (fo-lavo q '() '() q))
((((((_0 (etouq etouq) tsil) _0 tsil) (_0) adbmal)
     etouq)
    (((_0 (etouq etouq) tsil) _0 tsil) (_0) adbmal)))
```

Cool side note. When used in combination with the "val-ofo" code from class, you should be able to build programs in one language which, on given input, generate programs in the other language.

5. Color Middle Earth

If you take a look at a world map, it's colored in by countries. Notice, too that no two adjacent countries are given the same color. The four-color theorem [http://en.wikipedia.org/wiki/Four_color_theorem], that four distinct colors suffice to color any planar graph, was first conjectured in 1852. It stood as an open problem for more than 100 years, until it was proven with the aid of a computer in 1976. The initial proof, by contradiction and by cases for 1,936 cases, raised philosophical questions into what it means to be a proof.

Here, we're not asking you to prove the theorem. We're simply asking you to color a particular graph. Consider the following map of Tolkien's Middle Earth.

As the map is somewhat ambiguous, consider the simplified list representation below, which makes clear which realms (nodes) are adjacent (connected via an edge) to which. For each list, the car of the list is considered to be adjacent to all the realms in the cdr.

```
> (define middle-earth
    '((lindon eriador forodwaith)
      (forodwaith lindon rhovanion eriador)
      (eriador lindon forodwaith rhovanion enedwaith)
      (rhovanion forodwaith eriador enedwaith rohan rhun)
      (enedwaith eriador rhovanion rohan gondor)
      (rohan enedwaith rhovanion rhun gondor mordor)
      (gondor enedwaith rohan mordor)
      (rhun rohan rhovanion khand mordor)
      (mordor gondor rohan rhun khand harad)
      (khand mordor rhun harad)
      (harad mordor khand)))
```

You may find you don't need this list `middle-earth` per se, just the information it contains. Write a program `color-middle-earth`, that takes a list of colors, and uses that as a part of a miniKanren relation to associate a list of colors with an alist containing pairs (node . color). Notice that we aren't asking you to solve the general case, merely this particular graph. Also notice we only ask for one answer. Also, its not necessary to generate this exact coloring: any valid coloring will suffice. Moreover, your solution isn't required to “run backwards”. The trick to this problem is to just tell miniKanren what needs to be the case. Declare what you want, and miniKanren will color your graph.

```
> (color-middle-earth '(red orange purple black))
(((lindon . red) (forodwaith . orange) (eriador . purple) (rhovanion . red)
   (enedwaith . orange) (rohan . purple) (gondor . red)
   (rhun . orange) (mordor . black) (khand . red)
   (harad . orange)))
```

Two hints.

1. Use `absento` and the recursive relation `membero`. You will have to define `membero`, you can get it out of the book.
2. Consider the following quote:

I used Prolog in a comparative languages course. The biggest program we did was a map-coloring one (color a map with only four colors so that no bordering items have the same color, given a mapping of things that border each other). I say biggest because we were given the most time with it. I started out like most people in my class trying to hack the language into letting me code a stinking algorithm to color a stinking map. Then I wrote a test function to check if the map was colored and, in a flash of prolog, realized that that was really all I needed to code.

Brainteaser

Omitted.

Just Dessert

In my implementation of `color-middle-earth`, we've hard-coded it to color a particular graph. That's somewhat unfortunate. We could write a relation `color-middle-eartho` that holds between a list of colors, a list of `(node . neighbors)` pairs, where `neighbors` is a list of neighbors, and an association list of `(node . color)`, that represents the graph coloring. Which is a cool thing to do!

But instead, let's do something else. Write a **macro** `color-graph` that takes a name and a list of pairs `(node . neighbors)` and have it expand into the definition of a function that will take a list of colors and generate their coloring. Another way of saying it is that your macro should expand into the definition that is your solution to the `color-middle-earth` problem.

```
> (color-grapho another-color-middle-earth
                ((lindon eriador forodwaith)
                 (forodwaith lindon rhovanion eriador)
                 (eriador lindon forodwaith rhovanion enedwaith)
                 (rhovanion forodwaith eriador enedwaith rohan rhun)
                 (enedwaith eriador rhovanion rohan gondor)
                 (rohan enedwaith rhovanion rhun gondor mordor)
                 (gondor enedwaith rohan mordor)
                 (rhun rohan rhovanion khand mordor)
                 (mordor gondor rohan rhun khand harad)
                 (khand mordor rhun harad)
                 (harad mordor khand)))
> (another-color-middle-earth '(red orange purple black))
(((harad . orange) (khand . red) (mordor . purple) (rhun . orange)
   (gondor . red) (rohan . black) (enedwaith . orange)
   (rhovanion . red) (eriador . purple) (forodwaith . orange)
   (lindon . red)))
```