C311/B521/A596 Programming Languages

Home   Policies   Instructors   Resources   Schedule   Assignments   B621

Login

Assignment 10: Introduction to Logic Programming

> Success is the ability to go from one failure to another with no loss of enthusiasm.
>
> —*Winston Churchill*

Preliminaries

This week we've begun our discussion of relational programming using miniKanren. Before you start the assignment, make sure you have `mk.rkt` `mk.scm` and `numbers.rkt` from the archive `c311-sup.zip`. Go ahead and put the contents in your C311 directory (that you linked with `raco`) and you should just be able just require them as follows:

```
(require C311/mk)
(require C311/numbers)
```

To familiarize yourself with miniKanren, you should review relevant sections of the *The Reasoned Schemer*, class code posted to OnCourse, and your own notes from class.

Assignment

This assignment is composed of two parts, plus a brainteaser. To test your assignment, you can used the attached `a10-student-tests.rkt`

Part I

Write the answers to the questions found in the `a10.rkt` file provided to you. **For each problem, explain `how` miniKanren arrived at the answer.** You will be graded on the quality and completeness of your explanation; a full explanation will require several sentences.

Part II

Here are the Racket procedures `assoc`, and `reverse`, `stutter`:

```
(define assoc
  (lambda (x ls)
    (let-pair ((a . d) ls)
      (let-pair ((aa . da) a)
        (cond
          ((equal? aa x) a)
          ((not (equal? aa x)) (assoc x d)))))))

(define reverse
  (lambda (ls)
    (cond
      ((equal? ls '()) '())
      ((not (equal? ls '()))
       (let-pair ((a . d) ls)
         (let ((res (reverse d)))
           (append res `(,a))))))))

(define stutter
  (lambda (ls)
    (cond
      ((null? ls) '())
      (else
        (let-pair ((a . d) ls)
          (let ((res (stutter d)))
            `(,a ,a . ,res)))))))
```

Take `assoc`, `reverse`, and `stutter`, and translate them into the equivalent miniKanren relations (`assoco`, and `reverseo`, `stuttero`) and put them at the end of your `a10.rkt` file.

The below tests are a guide. Your relations might not pass these tests, as implementing goals in a different order may cause the stream to return results in a different order. So it is possible that your code is correct though the tests fail.

Remember that you may need to rearrange some of the goals in your relations to ensure termination (and therefore to pass the tests). In general, recursive calls should come at the end of a sequence of goals, while explicit or implicit calls to `==` should come at the beginning.

Here are the tests.

```
> (require "a10.rkt")

> (run 1 (q) (stuttero q '(1 1 2 2 3 3)))
((1 2 3))

> (run* (q) (stuttero q '(1 1 2 2 3 3)))
((1 2 3))

> (run 1 (q) (fresh (a b c d) (== q `(,a ,b ,c ,d)) (stuttero a `(1 ,b ,c 2 3 ,d))))
(((1 2 3) 1 2 3))

> (run 1 (q) (fresh (a b c d) (== q `(,a ,b ,c ,d)) (stuttero `(,b 1) `(,c . ,d))))
((_.0 _.1 _.1 (_.1 1 1)))

> (run 1 (q) (fresh (e f g) (== q `(,e ,f ,g)) (stuttero `(,e . ,f) g)))
((_.0 () (_.0 _.0)))

> (run 2 (q) (fresh (e f g) (== q `(,e ,f ,g)) (stuttero `(,e . ,f) g)))
((_.0 () (_.0 _.0)) (_.0 (_.1) (_.0 _.0 _.1 _.1)))

> (run* (q) (assoco 'x '() q))
()

> (run* (q) (assoco 'x '((x . 5)) q))
((x . 5))

> (run* (q) (assoco 'x '((y . 6) (x . 5)) q))
((x . 5))

> (run* (q) (assoco 'x '((x . 6) (x . 5)) q))
((x . 6))

> (run* (q) (assoco 'x '((x . 5)) '(x . 5)))
(_.0)

> (run* (q) (assoco 'x '((x . 6) (x . 5)) '(x . 6)))
(_.0)

> (run* (q) (assoco 'x '((x . 6) (x . 5)) '(x . 5)))
()

> (run* (q) (assoco q '((x . 6) (x . 5)) '(x . 5)))
()

> (run* (q) (assoco 'x '((x . 6) . ,q) '(x . 6)))
(_.0)

> (run 5 (q) (assoco 'x q '(x . 5)))
(((x . 5) . _.0)
 ((((_.0 . _.1) (x . 5) . _.2) (=/= ((_.0 x))))
 ((((_.0 . _.1) (_.2 . _.3) (x . 5) . _.4) (=/= ((_.0 x)) ((_.2 x))))
 ((((_.0 . _.1) (_.2 . _.3) (_.4 . _.5) (x . 5) . _.6)
   (=/= ((_.0 x)) ((_.2 x)) ((_.4 x))))
 ((((_.0 . _.1) (_.2 . _.3) (_.4 . _.5) (_.6 . _.7) (x . 5) . _.8)
   (=/= ((_.0 x)) ((_.2 x)) ((_.4 x)) ((_.6 x)))))

> (run 5 (q) (fresh (x y z)
                (assoco x y z)
                (== `(,x ,y ,z) q)))
((_.0 ((_.0 . _.1) . _.2) (_.0 . _.1))
 ((_.0 ((_.1 . _.2) (_.0 . _.3) . _.4) (_.0 . _.3)) (=/= ((_.0 _.1))))
 ((_.0 ((_.1 . _.2) (_.3 . _.4) (_.0 . _.5) . _.6) (_.0 . _.5))
   (=/= ((_.0 _.1)) ((_.0 _.3))))
 ((_.0 ((_.1 . _.2) (_.3 . _.4) (_.5 . _.6) (_.0 . _.7) . _.8) (_.0 . _.7))
   (=/= ((_.0 _.1)) ((_.0 _.3)) ((_.0 _.5))))
 ((_.0
    ((_.1 . _.2) (_.3 . _.4) (_.5 . _.6) (_.7 . _.8) (_.0 . _.9) . _.10)
    (_.0 . _.9))
   (=/= ((_.0 _.1)) ((_.0 _.3)) ((_.0 _.5)) ((_.0 _.7)))))

> (run* (q) (reverseo '() q))
(())

> (run* (q) (reverseo '(a) q))
((a))

> (run* (q) (reverseo '(a b c d) q))
((d c b a))

> (run* (q) (fresh (x) (reverseo `(a b ,x c d) q)))
```

```
((d c _.0 b a))

> (run* (x) (reverseo `(a b ,x d) '(d c b a)))
(c)

> (run* (x) (reverseo `(a b c d) `(d . ,x)))
((c b a))

> (run* (q) (fresh (x) (reverseo `(a b c d) `(d . (,q . ,x)))))
(c)

> (run 10 (q) (fresh (x y) (reverseo x y) (== `(,x ,y) q)))
((() ()) ((_.0) (_.0)) ((_.0 _.1) (_.1 _.0))
 ((_.0 _.1 _.2) (_.2 _.1 _.0))
 ((_.0 _.1 _.2 _.3) (_.3 _.2 _.1 _.0))
 ((_.0 _.1 _.2 _.3 _.4) (_.4 _.3 _.2 _.1 _.0))
 ((_.0 _.1 _.2 _.3 _.4 _.5) (_.5 _.4 _.3 _.2 _.1 _.0))
 ((_.0 _.1 _.2 _.3 _.4 _.5 _.6)
  (_.6 _.5 _.4 _.3 _.2 _.1 _.0))
 ((_.0 _.1 _.2 _.3 _.4 _.5 _.6 _.7)
  (_.7 _.6 _.5 _.4 _.3 _.2 _.1 _.0))
 ((_.0 _.1 _.2 _.3 _.4 _.5 _.6 _.7 _.8)
  (_.8 _.7 _.6 _.5 _.4 _.3 _.2 _.1 _.0)))
```

Brainteaser

Chapter 7 of *The Reasoned Schemer* introduces a way to handle integers in relational programming. These are provided in the `numbers` suite.

Write `lengtho`, which associates its output with the length of a list as a binary number in reverse order (if you use the provided helpers, this is the normal representation of numbers). Here are some tests:

```
> (require C311/numbers)

> (run 1 (q) (lengtho '() q))
(())

> (run 1 (q) (lengtho '(a b) q))
((0 1))

> (run 1 (q) (lengtho '(a b c) q))
((1 1))

> (run 1 (q) (lengtho '(a b c d e f g) q))
((1 1 1))

> (run 1 (q) (lengtho q (build-num 0)))
(())

> (run 1 (q) (lengtho q (build-num 5)))
((_.0 _.1 _.2 _.3 _.4))

> (run 10 (q) (fresh (x y) (lengtho x y) (== `(,x ,y) q)))
((() ()) ((_.0) (1)) ((_.0 _.1) (0 1)) ((_.0 _.1 _.2) (1 1))
 ((_.0 _.1 _.2 _.3) (0 0 1)) ((_.0 _.1 _.2 _.3 _.4) (1 0 1))
 ((_.0 _.1 _.2 _.3 _.4 _.5) (0 1 1))
 ((_.0 _.1 _.2 _.3 _.4 _.5 _.6) (1 1 1))
 ((_.0 _.1 _.2 _.3 _.4 _.5 _.6 _.7) (0 0 0 1))
 ((_.0 _.1 _.2 _.3 _.4 _.5 _.6 _.7 _.8) (1 0 0 1)))
```

lp-a2.txt · Last modified: 2014/11/06 20:46 by jhemann