miniKanren Puzzles

William E. Byrd

An Unreasonable Schemer Production

Book 02 of Will's ShovelWare Book Pack
Version 1.0 Sunday, October 26, 2025

To my wonderful parents, Ellen Byrd and Dr. Edward Byrd.

About Will's ShovelWare Book Pack

In an experiment to see if I can train myself to finish writing a book on my own—instead of just poking at it for a few weeks before abandoning it and starting another—I have publically committed to writing and releasing seventeen (17) books in one year, releasing one book every three weeks. I made the original announcement on Will Radio Part XXVIII[1] on September 15, 2025, with the experiment to end on September 6, 2026, when the 17th book is due.

Good or bad, short or long, my goal is to finish and release a book every 21 days, like clockwork.

The idea for this experiment came from my kiloTube Video challenge from a couple years ago. I had been wanting to make more videos, but was suffering from perfectionism—I would record 70 or 80 takes of a video before deleting it in disgust. So I publically announced that I was going to make 1,024 (or 2^{10}) videos in a single year (one kiloTube worth), in an attempt to force myself to just make video after video, with the idea that I'd eventually learn the mechanics, and learn my own style for making videos. Although I didn't come close to making 1,024 videos that year (partly because making videos involved making noise, which caused some social issues), the practice of making an average of 3 videos a day for weeks quickly got me over my perfectionism block. I now have no trouble sitting down and making a video in one take, editing it, and uploading it in a single sitting.

My hope is that after finishing and releasing a dozen or so books, I'll start to relax out a little, and will be over the fear of releasing something less than perfect. There's a go adage, "Lose your first hundred games as quickly as possible." I'm trying to write and release my first dozen (or so) terrible books as quickly as possible. With 17 books to write, I might write a few decent ones, as well.

I am allowing myself to update books I've already released, so long as I continue releasing a new book every 3 weeks.

This is the second of the 17 books, due on Sunday, October 26, 2025, Anywhere on Earth (or whenever I go to sleep!). (Originally I was going to release the books on Monday nights, but Sundays work better.) The next book, book 03, will be due Sunday, November 16, 2025.

[1] `https://www.youtube.com/watch?v=d6_cB-jtxYI`

Contents

Preface

This book is a collection of programming puzzles in the miniKanren programming language. Inspired by graded chess problems, the intent is to present puzzles of increasing difficulty that I hope you will find fun and stimulating, and will increase your understanding and enjoyment of miniKanren programming.

I wrote this book for readers who are familiar with miniKanren, and who have written and run at least a few miniKanren programs. If you have read the first or second edition of *The Reasoned Schemer*[2], or have implemented a miniKanren or microKanren[3], or have read my PhD dissertation[4], you should be ready for at least the easier puzzles. The first chapter of this book is a brief overview of the version of miniKanren used in these puzzles, since there are now many dialects of miniKanren, in multiple host languages.

The puzzles have all been tested in the faster-miniKanren[5] implementation of miniKanren, running under Chez Scheme[6] 10.2.0.

Acknowledgements

My mother, Ellen Byrd, suggested that this book be about programming puzzles, during a conversation we were having with my father, Dr. Edward Byrd, about chess puzzles. My father said that my mom's suggestion was "wonderful," and that I might present the puzzles in order of increasing difficulty, similar to a graded book of chess puzzles. I agree that her suggestion was wonderful, as was his, and as are my parents. I greatly appreciate the encouragement

[2] *The Reasoned Schemer*, MIT Press, 2005, and *The Reasoned Schemer*, Second Edition, MIT Press, 2018

[3] `https://github.com/jasonhemann/microKanren`

[4] `https://github.com/webyrd/dissertation-single-spaced`

[5] `https://github.com/michaelballantyne/faster-miniKanren`

[6] `https://cisco.github.io/ChezScheme/`

of my parents, brother, and sister to write books, to finish books, release books into the world. I love you all.

Nada Amin looked at an early draft, and encouraged me to focus more on the concepts, and to make the book tell a story. I hope I succeeded, at least partially. Thank you, Nada!

I thank Matt Might and the rest of the Hugh Kaul Precision Medicine Institute at the University of Alabama at Birmingham for their support and encouragement over the years.

I've learned a great deal about miniKanren programming from hundreds of budding and expert miniKanren programmers over the past 20 years. I am especially grateful to my many students, mentees, collaborators, and co-authors. Thank you all!

Will Byrd
Mt. Pleasant, South Carolina
October, 2025

Chapter 1

The dialect of miniKanren used in this book

This book uses the faster-miniKanren implementation[1] of miniKanren, running in Chez Scheme[2] 10.2.0.

Here are the parts of the faster-miniKanren dialect we will use.

1.1 `run n` (Query interface)

The `run` form acts as the interface between miniKanren and the Scheme host language.

Syntax:

`(run` *`ne`* `(`*`x x*`* `...)` *`ge ge*`* `...)`

where,

`ne` is a Scheme expression that evaluates to a non-negative integer, indicating the maximum number of answers to be retuned

`(`*`x x*`* `...)` is a list of one or more distinct identifiers that serve as fresh *query variables*

`ge ge*` `...` is a sequence of one or more *goal expressions*—that is, Scheme expressions that evaluate to miniKanren goals

Example:

[1] `https://github.com/michaelballantyne/faster-miniKanren`
[2] `https://cisco.github.io/ChezScheme/`

```
(run 2 (a b c)
  (== '(cat fish) a)
  (conde
    ((== c b))
    (succeed)
    ((== a b))))
=>
(((cat fish) _.0 _.0)
 ((cat fish) _.0 _.1))
```

Example:

```
(run (- (* 2 3) (+ 3 3)) (a b c)
  (== '(cat fish) a))
=>
()
```

Example:

```
(run (- (* 2 3) (+ 2 3)) (a b c)
  (== '(cat fish) a))
=>
(((cat fish) _.0 _.1))
```

1.2 `run*` (Query interface)

Like `run`, the `run*` form acts as the interface between miniKanren and the Scheme host language.

Syntax:

`(run* (`*x x* ...*`)` *ge ge* ...*`)`

where,

`(`*x x* ...*`)` is a list of one or more distinct identifiers that serve as fresh *query variables*

ge ge ...* is a sequence of one or more *goal expressions*—that is, Scheme expressions that evaluate to miniKanren goals

Example:

```
(run* (a b c)
  (== '(cat fish) a)
  (conde
    ((== c b))
    (succeed)
    ((== a b))))
=>
(((cat fish) _.0 _.0)
 ((cat fish) _.0 _.1)
 ((cat fish) (cat fish) _.0))
```

1.3 `succeed` (A goal that succeeds)

The `succeed` goal succeeds when run, and is equivalent to `(== #f #f)`.

Syntax:

```
succeed
```

Example:

```
(run* (q) succeed)
=>
(_.0)
```

1.4 `fail` (A goal that fails)

The `fail` goal fails when run, and is equivalent to `(== #f #t)`.

Syntax:

```
fail
```

Example:

```
(run* (q) fail)
=>
()
```

1.5 `fresh` (Introduces fresh variables, conjunction of goals)

The `fresh` form introduces fresh variables, and performs a conjunction of goals.

Syntax:

`(fresh (`*x*`* ...) `*ge ge*`* ...)`

where,

`(`*x*`* ...)` is a list of zero or more distinct identifiers that serve as fresh variables

ge ge`* ...` is a sequence of one or more *goal expressions*—that is, Scheme expressions that evaluate to miniKanren goals

Example:

```
(run* (q)
  (fresh (x y z)
    (== (list x y z) q)
    (== 'cat x)
    (== (list x y) z)))
=>
((cat _.0 (cat _.0)))
```

Example:

```
(run* (q)
  (fresh (q)
    (== 5 q))
  (fresh (q)
    (== 6 q)))
=>
(_.0)
```

1.6 `conde` (Disjunction of conjunction(s) of goals)

The `conde` form performs a disjunction over its clauses; each clause, in turn, performs a conjunction of its goals.

Syntax:

`(conde (`*ge1 ge1* ...*`) (`*ge2 ge2* ...*`) ...)`

where,

`(`*ge1 ge1* ...*`)` is a list of one or more *goal expressions*—that is, Scheme expressions that evaluate to miniKanren goals

`(`*ge2 ge2* ...*`)` `...` is a sequence of zero or more lists of goal expressions

Example:

```
(run* (x y)
  (conde
    ((== 'cat x))
    ((== 'dog y) (== y x) (== 'fish x))
    ((== 'bear y) (== 'mouse x))))
=>
((cat _.0)
 (mouse bear))
```

1.7 == (Equality constraint)

The `==` procedure is a goal constructor that takes two expressions—which much evaluate to legal miniKanren terms—and returns a goal that, when run, attempts to unify those terms, either succeeding (and potentially extending the substitution), or failing.

Syntax:

`(==` *te1 te2*`)`

where,

te1 **and** *te2* are *term expressions*—that is, Scheme expressions that evaluate to miniKanren terms

Example:

```
(run* (x y z) (== (list y z) x) (== 'cat y))
=>
(((cat _.0) cat _.0))
```

1.8 =/= (Disequality constraint)

The `=/=` procedure is a goal constructor that takes two expressions—which much evaluate to legal miniKanren terms—and returns a goal that, when run, attempts to unify those terms. If the terms do not unify, the disequality constraint cannot be violated, and the goal succeeds. If the terms unify, without extending the substitution, the terms are already identical, which means the disequality constraint is violated, and the goal fails. If the terms unify, extending the substitution, the terms are not currently identical, but could become so in the future, so a (potentially simplified) disequality constraint is retained in the constraint store.

Syntax:

`(=/= te1 te2)`

where,

te1* and *te2 are *term expressions*—that is, Scheme expressions that evaluate to miniKanren terms

Example:

```
(run* (x y)
  (== x y)
  (=/= y x))
=>
()
```

Example:

```
(run* (x y)
  (fresh (a b c d)
    (== (list a b) x)
    (=/= x y)
    (== 5 a)
    (== (list c d) y)
    (== 5 c)))
=>
((((5 _.0) (5 _.1))) (=/= ((_.0 _.1))))
```

1.9 `absento` (Absence constraint)

The `absento` procedure is a goal constructor that takes two expressions—which much evaluate to legal miniKanren terms—and returns a goal that, when run, fails if the first term is identical to, or appears anywhere within, the second term. Otherwise, the goal succeeds.

Syntax:

`(absento` *`te1 te2`*`)`

where,

`te1` **and** ***`te2`*** are *term expressions*—that is, Scheme expressions that evaluate to miniKanren terms

Example:

```
(run* (q)
  (absento 'cat 'cat))
=>
()
```

Example:

```
(run* (q)
  (absento q q))
=>
()
```

Example:

```
(run* (q)
  (absento q (list q)))
=>
()
```

Example:

```
(run* (q)
  (absento (list q) q))
=>
(_.0)
```

Example:

```
(run* (x y)
  (absento x y))
=>
(((_.0 _.1) (absento (_.0 _.1))))
```

Example:

```
(run* (x y)
  (== 'dog x)
  (absento x y))
=>
(((dog _.0) (absento (dog _.0))))
```

Example:

```
(run* (x y)
  (fresh (a d)
    (== (cons a d) y)
    (== 'fish d))
  (== 'dog x)
  (absento x y))
=>
(((dog (_.0 . fish)) (absento (dog _.0))))
```

1.10 `numbero`, `symbolo`, and `stringo` (Type constraints)

The `numbero`, `symbolo`, and `stringo` procedures are goal constructors that each take a single term expression—which much evaluate to a legal miniKanren term—and returns a goal that, when run, succeeds if the term is a fresh variable, or is a ground Scheme value of the appropriate type. Otherwise, the goal fails.

Syntax:

```
(numbero te)

(symbolo te)

(stringo te)
```

where,

`te` is a *term expression*—that is, a Scheme expression that evaluate to a miniKanren term

Example:

```
(run* (x y z)
  (numbero x)
  (symbolo y)
  (stringo z))
=>
(((_.0 _.1 _.2) (num _.0) (str _.2) (sym _.1)))
```

Example:

```
(run* (x y z)
  (== 5 x)
  (numbero x)
  (== "Hello!" z)
  (symbolo y)
  (== 'cat y)
  (stringo z))
=>
((5 cat "Hello!"))
```

Example:

```
(run* (x y z)
  (numbero x)
  (conde
    ((== 'dog x))
    ((== "Bye!" y))
    ((== 5 z)))
  (symbolo y)
  (stringo z))
=>
()
```

Chapter 2

Fill in the blanks

For each problem, fill in the blanks (___) according to the instructions.

Puzzle 2.1 Fill in the blanks of this `run*` expression

```
(run* (___) (___ ___ ___))
```

using the following identifiers

```
==
5
x
```

to make the `run*` expression produce the answer:

```
(5)
```

You may use a single identifier in multiple blanks.

Puzzle 2.2 Fill in the blanks of this `run*` expression

```
(run* (___)
  (conde
    ((___ (quote ___) q))
    ((___ (___ (___) (___ (quote ___) ___)) q))
    ((___ (quote (___)) (quote ((quote ___) ___)) q))))
```

using the following identifiers

```
==
appendo
cat
run*
q
```

to make the `run*` expression produce the answer:

```
(run* (q) (== 'cat q))
```

You may use a single identifier in multiple blanks.

The `appendo` helper relation is defined as:

```
(define (appendo l s ls)
  (conde
    ((== '() l) (== s ls))
    ((fresh (a d res)
       (== (cons a d) l)
       (== (cons a res) ls)
       (appendo d s res)))))
```

Chapter 3

Diverge or terminate?

Standard `appendo`

Given this definition of `appendo`, determine if each query will *diverge* (loop forever) or *terminate* (finish running in finite time).

```
(define appendo
  (lambda (l s ls)
    (conde
      ((== '() l) (== s ls))
      ((fresh (a d res)
         (== (cons a d) l)
         (== (cons a res) ls)
         (appendo d s res))))))
```

Puzzle 3.1.1 Will this query *diverge* or *terminate*?

```
(run* (x) (appendo '(a b c) '(d e) '(a b c d e)))
```

Puzzle 3.1.2 Will this query *diverge* or *terminate*?

```
(run* (x) (appendo `(,x b c) '(d e) '(a b c d e)))
```

Puzzle 3.1.3 Will this query *diverge* or *terminate*?

```
(run 6 (x y) (appendo x y '(a b c d e)))
```

Puzzle 3.1.4 Will this query *diverge* or *terminate*?

```
(run 7 (x y) (appendo x y '(a b c d e)))
```

Puzzle 3.1.5 Will this query *diverge* or *terminate*?

```
(run* (x y) (appendo x y '(a b c d e)))
```

Reordered `appendo`

Given this definition of `appendo`, determine if each query will *diverge* (loop forever) or *terminate* (finish running in finite time).

```
(define appendo
  (lambda (l s ls)
    (conde
      ((fresh (a d res)
         (appendo d s res)
         (== (cons a d) l)
         (== (cons a res) ls)))
      ((== s ls) (== '() l)))))
```

Puzzle 3.2.1 Will this query *diverge* or *terminate*?

```
(run* (x) (appendo '(a b c) '(d e) '(a b c d e)))
```

Puzzle 3.2.2 Will this query *diverge* or *terminate*?

```
(run* (x) (appendo `(,x b c) '(d e) '(a b c d e)))
```

Puzzle 3.2.3 Will this query *diverge* or *terminate*?

```
(run 6 (x y) (appendo x y '(a b c d e)))
```

Puzzle 3.2.4 Will this query *diverge* or *terminate*?

```
(run 7 (x y) (appendo x y '(a b c d e)))
```

Puzzle 3.2.5 Will this query *diverge* or *terminate*?

```
(run* (x y) (appendo x y '(a b c d e)))
```

Chapter 4

Reorder the conjuncts and disjuncts

Reorder the conjuncts and disjuncts in the following definitions to work well with miniKanren's complete interleaving search.

Puzzle 4.1 Reorder the conjuncts and disjuncts

```
(defrel (parseo expr)
  (conde
    ((numbero expr))
    ((== #t expr))
    ((== #f expr))
    ((fresh (e)
       (parseo e)
       (== `(zero? e) expr)))
    ((fresh (e1 e2)
       (== `(+ ,e1 ,e2) expr)
       (parseo e1)
       (parseo e2)))
    ((fresh (e1 e2)
       (== `(* ,e1 ,e2) expr)
       (parseo e1)
       (parseo e2)))))
```

Chapter 5

Spot the errors

Identify all the errors in this definition of `eval-expro`, and write the correct definition:

```
(define eval-expro
  (lambda (expr env val)
    (conde
      (conde
        (conde
          ((numbero expr) (== expr val))
          ((== (== #f expr) val))
          ((== (== #t expr) val)))
        ((== `(quote ,datum) expr) (== datum val))
        ((fresh (e1 e2 v1 v2)
           (== `(cons ,e1 ,e2) expr)
           (== `(pair ,v1 ,v2) val)
           (eval-expro e1 env v1)
           (eval-expro e2 env v2)))
        ((fresh (e pr v1 v2)
           (== `(car ,e) expr)
           (== (car (cdr pr)) val)
           (== `(pair ,v1 ,v2) pr)
           (eval-expro e env pr))))))
```

Chapter 6

Simplify the program

For each problem, simplify the definition of the relation.

Puzzle 6.1 Simplify the definition of `bazo`

```
(define bazo
  (lambda (a b c)
    (fresh (x y)
      (fresh (y x)
        (== y x)
        (fresh (z)
          (== z y)
          (conde
            ((== z a) (== (list x) b)))
          (fresh (w)
            (== c w)
            (== w x))))
      (== x 5)
      (== y 6))))
```

Puzzle 6.2 Simplify the definition of `quxo`

```
(define quxo
  (lambda (a b c)
    (fresh (x y)
      (== y x)
      (fresh (y x)
        (== y x)
        (fresh (z)
          (== z y)
          (conde
            ((== z a) (== (list x) b)))
          (fresh (w)
            (== c w)
            (== w x))))
      (== x 5)
      (== y 6))))
```

Chapter 7

Which clauses overlap?

Which pairs of `conde` clauses overlap in the following definitions?

Puzzle 7.1 Which pairs of `conde` clauses overlap?

```
(define bazo
  (lambda (x y)
    (conde
      ((fresh (ne)
         (== (cons 'num (cons ne '())) x)
         (== ne y)))
      ((fresh (var body)
         (== (list 'lam var body) x)
         (== body y)))
      ((fresh (e1 e2)
         (== (cons '+ (cons e1 (cons e2 '()))) x)
         (== (cons e1 e2) y)))
      ((fresh (a b c d)
         (== `(,a ,b ,c . ,d) x)
         (== (list b c d) y))))))
```

Chapter 8

Make non-overlapping

For each problem, rewrite the definition to make overlapping `conde` clauses no longer overlap.

Puzzle 8.1 Rewrite `lookupo` so that clauses no longer overlap

```
(defrel (lookupo x env val)
  (fresh (y v env^)
    (symbolo x)
    (symbolo y)
    (== `((,y . ,v) . ,env^) env)
    (conde
      ((== x y) (== v val))
      ((lookupo x env^ val)))))
```

Chapter 9

Replace unnecessary tags

Using tagged lists or pairs to represent expressions or values is a standard, useful technique when writing evaluators, type inferencers, parsers, theorem provers, static analyzers, and any other program that acts as an interpreter over data. When writing a relation that operates over Scheme expressions, this tagging of expressions can get in the way, making expressions more verbose, harder to read and interpret (especially if there is a fresh variable in place of a tag), and prevents using the same expression in both a Scheme implementation and in the relation. For this reason, it is helpful to know how to remove unnecessary tagging.

For each problem, rewrite the definition to remove unnecessary tagging from the expressions, using type constraints, helper relations, or other techniques as necessary to preserve non-overlapping between clauses.

Puzzle 9.1 Replace unnecessary tagging

```
(define parseo
  (lambda (expr)
    (conde
      ((fresh (n)
         (== `(num ,n) expr)))
      ((fresh (b)
         (== `(bool ,b) expr)))
      ((fresh (e)
         (== `(zero? ,e) expr)
```

```
      (parseo e)))
    ((fresh (e)
       (== `(not ,e) expr)
       (parseo e)))
    ((fresh (e1 e2)
       (== `(+ ,e1 ,e2) expr)
       (parseo e1)
       (parseo e2))))))
```

Puzzle 9.2 Replace unnecessary tagging

```
(defrel (eval-expro expr env val)
  (conde
    ((fresh (b)
       (== `(bool ,b) expr)
       (== expr val)))
    ((fresh (x e)
       (== `(lambda (,x) ,e) expr)
       (== `(closure ,x ,e ,env) val)))
    ((fresh (x)
       (== `(var ,x) expr)
       (lookupo x env val)))
    ((fresh (e b)
       (== `(not ,e) expr)
       (not-in-envo 'not env)
       (conde
         ((== #t b)
          (== #f val))
         ((== #f b)
          (== #t val)))
       (eval-expro e env b)))
    ((fresh (e1 e2 x e env^ v)
       (== `(app ,e1 ,e2) expr)
       (eval-expro e1 env `(closure ,x ,e ,env^))
       (eval-expro e2 env v)
       (eval-expro e `((,x . ,v) . ,env^) val)))))
```

Chapter 10

Fully tag the expressions

For each problem, change the representation of expressions from a minimally-tagged representation to a fully tagged expression representation, using neither type constraints nor `not-in-envo`.

Puzzle 10.1 Fully tag the expressions in `!-o`

```
(defrel (not-in-envo x env)
  (fresh ()
    (symbolo x)
    (conde
      ((== '() env))
      ((fresh (y v env^)
         (symbolo y)
         (== `((,y . ,v) . ,env^) env)
         (=/= x y)
         (not-in-envo x env^))))))

(define (!-o expr env type)
  (conde
    ((numbero expr) (== 'num type))
    ((symbolo expr) (lookupo expr env type))
    ((fresh (e t)
       (== `(car ,e) expr)
       (not-in-envo 'car env)
       (!-o e env `(pair ,type ,t))))
    ((fresh (e t)
```

```
   (== `(cdr ,e) expr)
   (not-in-envo 'cdr env)
   (!-o e env `(pair ,t ,type))))
((fresh (e1 e2 t1 t2)
   (== `(cons ,e1 ,e2) expr)
   (== `(pair ,t1 ,t2) type)
   (not-in-envo 'cons env)
   (!-o e1 env t1)
   (!-o e2 env t2)))
((fresh (t t^)
   (symbolo x)
   (== `(lambda (,x) ,e) expr)
   (== `(-> ,t ,t^) type)
   (not-in-envo 'lambda env)
   (!-o e `((,x . ,t) . ,env) t^)))
((fresh (e1 e2 t)
   (== `(,e1 ,e2) expr)
   (!-o e1 env `(-> ,t ,type))
   (!-o e2 env t)))))
```

Chapter 11

Equivalent, or not?

For each of these problems, determine if the two expressions or definitions are equivalent.

Puzzle 11.1 Are these definitions equivalent?

```
(define baro
  (lambda (a b c)
    (== a c)))
```

```
(define baro
  (lambda (x y z)
    (== x z)))
```

Puzzle 11.2 Are these `run*` expressions equivalent?

```
(run* (a b c) (== a c))
```

```
(run* (c b a) (== c a))
```

Puzzle 11.3 Are these definitions equivalent?

```
(defrel (quuxo s t)
  (fresh (a s)
    (== 'cat a)
    (== 'dog s)
    (fresh (b)
      (== 'mouse b)
      (== a t))))
```

```
(defrel (quuxo s t)
  (fresh (x y)
    (== 'cat x)
    (== 'dog s)
    (fresh (z)
      (== 'mouse z)
      (== x t))))
```

Puzzle 11.4 Are these definitions equivalent?

```
(define (fooo a b c)
  (fresh (x y z)
    (== (list y z) x)
    (== a x)
    (== c z)))
```

```
(define (fooo a b c)
  (fresh (x)
    (fresh (y)
      (== (list y z) x)
      (fresh (z)
        (== a x)
        (== c z)))))
```

Chapter 12

Which answers are subsumed?

For each problem, which shows the result of a `run*` query, identify which answers are subsumed by other answers.

Puzzle 12.1 Which answers are subsumed by others?

```
(run* (q) (baro q))
=>
(_.0
 (_.0 (num _.0))
 (sub1 _.0)
 (+ _.0 _.1)
 (* _.0 _.1)
 (_.0 _.1 . _.2))
```

Chapter 13

Ground the answers

These problems involve individual answers selected from the result of this `run` query to the `eval-expro` relation:

```
(run 10 (expr) (fresh (val) (eval-expro expr '() val)))
```

For each problem, ground the answer by replacing each reified fresh variable with a legal Scheme value, consistent with any associated constraints.

Puzzle 13.1 Ground the answer

```
(_.0 (num _.0))
```

Puzzle 13.2 Ground the answer

```
((cons (lambda (_.0) _.1) _.2)
 (num _.2)
 (sym _.0))
```

Puzzle 13.3 Ground the answer

```
(((lambda (_.0) _.0) _.1)
 (num _.1)
 (sym _.0))
```

Puzzle 13.4 Ground the answer

```
(((lambda (_.0) (lambda (_.1) _.2)) _.3)
 (=/= ((_.0 lambda)))
 (num _.3)
 (sym _.0 _.1))
```

Answers

Chapter 2 Answers: Fill in the blanks

Puzzle 2.1 Fill in the blanks of this `run*` expression
The filled in `run*` expression:

```
(run* (x) (== 5 x))
```

or

```
(run* (x) (== x 5))
```

Puzzle 2.2 Fill in the blanks of this `run*` expression
The filled in `run*` expression:

```
(run* (q)
  (conde
    ((== (quote run*) q))
    ((== (run* (q) (== (quote q) q)) q))
    ((appendo (quote (==)) (quote ((quote cat) q)) q))))
```

Chapter 3 Answers: Diverge or terminate?

Standard `appendo`

Puzzle 3.1.1 Terminate
Puzzle 3.1.2 Terminate
Puzzle 3.1.3 Terminate
Puzzle 3.1.4 Terminate
Puzzle 3.1.5 Terminate

Reordered `appendo`

Puzzle 3.2.1 Terminate
Puzzle 3.2.2 Terminate
Puzzle 3.2.3 Terminate
Puzzle 3.2.4 Diverge
Puzzle 3.2.5 Diverge

Chapter 4 Answers: Reorder the conjuncts and disjuncts

Puzzle 4.1 Reorder the conjuncts and disjuncts

A reasonable ordering is:

```
(defrel (parseo expr)
  (conde
    ((numbero expr))
    ((== #t expr))
    ((== #f expr))
    ((fresh (e)
       (parseo e)
       (== `(zero? e) expr)))
    ((fresh (e1 e2)
       (== `(+ ,e1 ,e2) expr)
       (parseo e1)
       (parseo e2)))
    ((fresh (e1 e2)
       (== `(* ,e1 ,e2) expr)
       (parseo e1)
       (parseo e2)))))
```

The `conde` clauses for the `+` and `*` cases could be swapped, as could the order of the recursive calls within each of those clauses.

Chapter 5 Answers: Spot the errors

There are four errors in the definition of `eval-expo`:

1. Calls to `==` appear as arguments to `==` calls for the `#f` and `#t` clauses
2. The nested `conde` clause should be wrapped in parentheses
3. There is a missing `fresh` that is needed to introduce the variable `datum` in the `quote` clause
4. Scheme's `cdr` procedure is called on fresh variable `pr` in the expression `(car (cdr pr))`

The corrected code is:

```
(define eval-expro
  (lambda (expr env val)
    (conde
      ((conde
         ((numbero expr) (== expr val))
         ((== #f expr) (== expr val))
         ((== #t expr) (== expr val))))
      ((fresh (datum)
         (== `(quote ,datum) expr)
         (== datum val)))
      ((fresh (e1 e2 v1 v2)
         (== `(cons ,e1 ,e2) expr)
         (== `(pair ,v1 ,v2) val)
         (eval-expro e1 env v1)
         (eval-expro e2 env v2)))
      ((fresh (e pr v1 v2)
         (== `(car ,e) expr)
         (== v1 val)
         (== `(pair ,v1 ,v2) pr)
         (eval-expro e env pr))))))
```

which can be simplified to:

```
(define eval-expro
  (lambda (expr env val)
```

```
(conde
  ((numbero expr) (== expr val))
  ((== #f expr) (== expr val))
  ((== #t expr) (== expr val))
  ((== `(quote ,val) expr))
  ((fresh (e1 e2 v1 v2)
     (== `(cons ,e1 ,e2) expr)
     (== `(pair ,v1 ,v2) val)
     (eval-expro e1 env v1)
     (eval-expro e2 env v2)))
  ((fresh (e v2)
     (== `(car ,e) expr)
     (eval-expro e env `(pair ,val ,v2)))))))
```

Chapter 6 Answers: Simplify the program

Puzzle 6.1 Simplify the definition of `bazo`
We will simplify the definition of `bazo` one step at a time, testing the relation after each simplification to ensure we do not change the answers returned. Here is the original definition of `bazo`:

```
(define bazo
  (lambda (a b c)
    (fresh (x y)
      (fresh (y x)
        (== y x)
        (fresh (z)
          (== z y)
          (conde
            ((== z a) (== (list x) b)))
          (fresh (w)
            (== c w)
            (== w x))))
      (== x 5)
      (== y 6))))

(run* (a b c) (bazo a b c))
=>
((_.0 (_.0) _.0))
```

First, we can remove the outer `(fresh (x y) ... (== x 5) (== y 6))` since both variables `x` and `y` are shadowed by the inner `(fresh (y x) ...)`, and since both `(== x 5)` and `(== y 6)` will succeed once (since `x` and `y` are fresh), and since neither call to `==` affects the arguments to `bazo` (`a`, `b`, and `c`).

After the first simplification step, the definition of `bazo` is:

```
(define bazo
  (lambda (a b c)
    (fresh (y x)
      (== y x)
      (fresh (z)
        (== z y)
        (conde
          ((== z a) (== (list x) b)))
```

```
        (fresh (w)
          (== c w)
          (== w x))))))

(run* (a b c) (bazo a b c))
=>
((_.0 (_.0) _.0))
```

The `(fresh (w) (== c w) (== w x))` associates the argument `c` with the local fresh variable `w`, which is then associated with `x`. The variable `w` is not used in any other way. Therefore this entire `fresh` expression can be replaced with the direct association of `c` with `x`, `(== c x)`:

```
(define bazo
  (lambda (a b c)
    (fresh (y x)
      (== y x)
      (fresh (z)
        (== z y)
        (conde
          ((== z a) (== (list x) b)))
        (== c x)))))

(run* (a b c) (bazo a b c))
=>
((_.0 (_.0) _.0))
```

Another simplication comes from noticing that the `conde` expression has only a single clause, and therefore can be replaced by a `(fresh () ...)` containing the goal expressions from that clause:

```
(define bazo
  (lambda (a b c)
    (fresh (y x)
      (== y x)
      (fresh (z)
        (== z y)
        (fresh ()
          (== z a)
          (== (list x) b))
        (== c x)))))
```

```
(run* (a b c) (bazo a b c))
=>
((_.0 (_.0) _.0))
```

Since the `(fresh () ...)` we just introduced is already inside of a conjunction (the `(fresh (z) ...)`), we can remove the `(fresh () ...)` and splice the goal expressions into the surrounding `(fresh (z) ...)`:

```
(define bazo
  (lambda (a b c)
    (fresh (y x)
      (== y x)
      (fresh (z)
        (== z y)
        (== z a)
        (== (list x) b)
        (== c x)))))

(run* (a b c) (bazo a b c))
=>
((_.0 (_.0) _.0))
```

Since `y` and `x` are associated with each either, we can remove `x` from the `fresh`, and replace all occurrences of `x` with `y`:

```
(define bazo
  (lambda (a b c)
    (fresh (y)
      (fresh (z)
        (== z y)
        (== z a)
        (== (list y) b)
        (== c y)))))

(run* (a b c) (bazo a b c))
=>
((_.0 (_.0) _.0))
```

(Of course, we have to be careful of lexical scope and potential shadowing of variables when we perform this transformation.)

Similarly, since `y` and `z` are associated with each other (and since neither variable is shadowed), the `(fresh (z) ...)` can be

removed, with the enclosed goal expressions inlined into the outer `(fresh (y) ...)`, and all occurrences of `z` replaced by `y`:

```
(define bazo
  (lambda (a b c)
    (fresh (y)
      (== y a)
      (== (list y) b)
      (== c y))))

(run* (a b c) (bazo a b c))
=>
((_.0 (_.0) _.0))
```

Once again, since `y` and `a` are associated with each other, we can remove `y` from the list of variables introduced by the `fresh` form, and replace all occurrences of `y` with `a`:

```
(define bazo
  (lambda (a b c)
    (fresh ()
      (== (list a) b)
      (== c a))))

(run* (a b c) (bazo a b c))
=>
((_.0 (_.0) _.0))
```

Puzzle 6.2 Simplify the definition of `quxo`
Here is the original definition of `quxo`

```
(define quxo
  (lambda (a b c)
    (fresh (x y)
      (== y x)
      (fresh (y x)
        (== y x)
        (fresh (z)
          (== z y)
          (conde
            ((== z a) (== (list x) b)))
          (fresh (w)
```

```
            (== c w)
            (== w x))))
      (== x 5)
      (== y 6))))

(run* (a b c) (quxo a b c))
=>
()
```

The definition of `quxo` is identical to that of `bazo`, with a single exception (other than the name change): `(== y x)` has been added just after the outermost `fresh`:

```
(fresh (x y)
  (== y x)
  ...
  (== x 5)
  (== y 6))
```

Due to the addition of `(== y x)`, we can no longer safely remove the outer `fresh` and the `==` calls to the arguments introduced by that `fresh`. In fact, the chain `(== y x)`, `(== x 5)`, and `(== y 6)` is guaranteed to fail, since the chain is equivalent to `(== 5 6)`. Therefore, the entire `quxo` relation is guaranteed to fail. Which means we can simplify the entire definition of `quxo` to:

```
(define quxo
  (lambda (a b c)
    fail))

(run* (a b c) (quxo a b c))
=>
()
```

Chapter 7 Answers: Which clauses overlap?

Puzzle 7.1 Which pairs of `conde` clauses overlap?

The second and fourth clauses overlap, as do the third and fouth clauses. Both the second and third clauses succeed when `x` can be associated with a list of length 3, in which the first element is a symbol representing a tag (`lam` or `+`, respectively). The fourth clause overlaps with both of those patterns.

Although the fourth clause overlaps with both the second and third clauses, the second and third clauses do not overlap with each other due to the separate tags `lam` and `+`. (Indeed, this is the main reasons for tagging expressions and values.)

We can demonstrate the overlapping behavior with these two `run*` expressions, both of which produce two answers:

```
(run* (x y)
  (fresh (var body)
    (== `(lam ,var ,body) x)
    (bazo x y)))
=>
(((lam _.0 _.1) _.1)
 ((lam _.0 _.1) (_.0 _.1 ())))
```

and:

```
(run* (x y)
  (fresh (e1 e2)
    (== `(+ ,e1 ,e2) x)
    (bazo x y)))
=>
(((+ _.0 _.1) (_.0 . _.1))
 ((+ _.0 _.1) (_.0 _.1 ())))
```

The overlapping can also be seen by passing `bazo` fresh variables as its arguments:

```
(run* (x y) (bazo x y))
=>
(((num _.0) _.0)
 ((lam _.0 _.1) _.1)
 ((+ _.0 _.1) (_.0 . _.1))
 ((_.0 _.1 _.2 . _.3) (_.1 _.2 _.3)))
```

Chapter 8 Answers: Make non-overlapping

Puzzle 8.1 Rewrite `lookupo` so that clauses no longer overlap

The `conde` clauses in `lookupo` overlap because the recursive call will be tried even if `x` and `y` are the same. This overlap can result in multiple answers if the environment `env` includes shadowed variables, such as `((w . 5) (z . 3) (w . 6))` in which the variable `w` is shadowed, with `5` being the current value for `w`. Unfortunately, the original definition of `lookupo` produces both `5` and `6` as values bound to `w` in that environment:

```
(run* (val) (lookupo 'w '((w . 5) (z . 3) (w . 6)) val))
=>
(5 6)
```

`lookupo` can be made non-overlapping by guarding the recursive clause with the disequality `(=/= x y)`:

```
(defrel (lookupo x env val)
  (fresh (y v env^)
    (symbolo x)
    (symbolo y)
    (== `((,y . ,v) . ,env^) env)
    (conde
      ((== x y) (== v val))
      ((=/= x y) (lookupo x env^ val)))))
```

which we can verify by re-running the previous `run*` query:

```
(run* (val) (lookupo 'w '((w . 5) (z . 3) (w . 6)) val))
=>
(5)
```

Chapter 9 Answers: Replace unnecessary tags

Puzzle 9.1 Replace unnecessary tagging

Unnecessary `num` and `bool` tags can be removed from `parseo`, replacing the `num` tag with a use of the `numbero` type constraint, and replacing the `bool` tag with explicit `#f` and `#t` Boolean constants:

```
(define parseo
  (lambda (expr)
    (conde
      ((numbero expr))
      ((== #f expr))
      ((== #t expr))
      ((fresh (e)
         (== `(zero? ,e) expr)
         (parseo e)))
      ((fresh (e)
         (== `(not ,e) expr)
         (parseo e)))
      ((fresh (e1 e2)
         (== `(+ ,e1 ,e2) expr)
         (parseo e1)
         (parseo e2))))))
```

As a result of this untagging, the same expressions can be used both in Scheme and by the `parseo` relation:

```
(let ((my-expression '(not (zero? (+ 3 4)))))
  (list
    (eval my-expression)
    (run* (q) (parseo my-expression))))
=>
(#t (_.0))
```

Puzzle 9.2 Replace unnecessary tagging

The tricky case in `eval-expro` is removing the `app` tag for the application case. Removing the `app` tag results in the application clause overlapping with the `not` clause, which is why we add the `not-in-envo` call to the `not` clause.

```
(defrel (not-in-envo x env)
```

```
  (fresh ()
    (symbolo x)
    (conde
      ((== '() env))
      ((fresh (y v env^)
         (symbolo y)
         (== `((,y . ,v) . ,env^) env)
         (=/= x y)
         (not-in-envo x env^))))))

(defrel (eval-expro expr env val)
  (conde
    ((== #f expr) (== expr val))
    ((== #t expr) (== expr val))
    ((fresh (x e)
       (symbolo x)
       (== `(lambda (,x) ,e) expr)
       (== `(closure ,x ,e ,env) val)
       (not-in-envo 'lambda env)))
    ((symbolo expr) (lookupo expr env val))
    ((fresh (e b)
       (== `(not ,e) expr)
       (conde
         ((== #t b)
          (== #f val))
         ((== #f b)
          (== #t val)))
       (not-in-envo 'not env)
       (eval-expro e env b)))
    ((fresh (e1 e2 x e env^ v)
       (== `(,e1 ,e2) expr)
       (eval-expro e1 env `(closure ,x ,e ,env^))
       (eval-expro e2 env v)
       (eval-expro e `((,x . ,v) . ,env^) val)))))
```

Chapter 10 Answers: Fully tag the expressions

Puzzle 10.1 Fully tag the expressions in `!-o`

This fully tagged expression representation for `!-o` uses a `num` tag for numbers (not to be confused with the `num` type), a `var` tag for variables, and an `app` tag for procedure applications. Since application is explicitly tagged, there is no longer a need for the `not-in-envo` relation.

```
(define (!-o expr env type)
  (conde
    ((fresh (n)
       (== `(num ,n) expr)
       (== 'num type)))
    ((fresh (x)
       (== `(var ,x) expr)
       (lookupo x env type)))
    ((fresh (e t)
       (== `(car ,e) expr)
       (!-o e env `(pair ,type ,t))))
    ((fresh (e t)
       (== `(cdr ,e) expr)
       (!-o e env `(pair ,t ,type))))
    ((fresh (e1 e2 t1 t2)
       (== `(cons ,e1 ,e2) expr)
       (== `(pair ,t1 ,t2) type)
       (!-o e1 env t1)
       (!-o e2 env t2)))
    ((fresh (t t^)
       (symbolo x)
       (== `(lambda (,x) ,e) expr)
       (== `(-> ,t ,t^) type)
       (!-o e `((,x . ,t) . ,env) t^)))
    ((fresh (e1 e2 t)
       (== `(app ,e1 ,e2) expr)
       (!-o e1 env `(-> ,t ,type))
       (!-o e2 env t)))))
```

Chapter 11 Answers: Equivalent, or not?

Puzzle 11.1 Are these definitions equivalent?

The two definitions of `baro` are equivalent, with only a consistent rewriting of variables names needed to transform one definition into the other. (In technical jargon, the two definitions are *α-equivalent*.)

Puzzle 11.2 Are these `run*` expressions equivalent?

The two `run*` expressions are equivalent, with only a consistent rewriting of variables names needed to transform one definition into the other. (In technical jargon, the two expressions are *α-equivalent*.)

Puzzle 11.3 Are these definitions equivalent?

The two definitions of `quuxo` are *not* equivalent. There is no consistent renaming of variables that will produce one definition from the other; in the first definition of `quuxo`, the argument `s` is shadowed by `s` introduced by the outer `fresh` expression, while in the second definition of `quuxo` the argument `s` is not shadowed. Therefore, `(== 'dog s)` has behaves differently in the two definitions. (In technical jargon, the two definitions are not *α-equivalent*.)

Puzzle 11.4 Are these definitions equivalent?

The two definitions of `fooo` are not equivalent. The `(== (list y z) x)` in the second definition appears outside of the `(fresh (z) ...)` expression; therefore, the `z` in `(list y z)` is unbound, which will result in an error if the second definition of `fooo` is run.

Chapter 12 Answers: Which answers are subsumed?

Puzzle 12.1 Which answers are subsumed by others?

The last five answers are subsumed by the first answer, `_.0`. The final answer, `(_.0 _.1 . _.2)`, subsumes `(sub1 _.0)`, `(+ _.0 _.1)`, and `(* _.0 _.1)`.

Here is the definition of the `baro` relation that produced the six answers:

```
(define (baro x)
  (conde
    (succeed)
    ((numbero x))
    ((fresh (e)
       (== `(sub1 ,e) x)))
    ((fresh (e1 e2)
       (== `(+ ,e1 ,e2) x)))
    ((fresh (e1 e2)
       (== `(* ,e1 ,e2) x)))
    ((fresh (op e e*)
       (== `(,op ,e . ,e*) x)))))
```

Chapter 13 Answers: Ground the answers

The answers to be grounded were selected from this `run` query:

```
(run 10 (expr) (fresh (val) (eval-expro expr '() val)))
=>
((_.0 (num _.0))
 ((lambda (_.0) _.1) (sym _.0))
 ((cons _.0 _.1) (num _.0 _.1))
 (((lambda (_.0) _.1) _.2) (num _.1 _.2) (sym _.0))
 ((cons _.0 (lambda (_.1) _.2)) (num _.0) (sym _.1))
 ((cons (lambda (_.0) _.1) _.2) (num _.2) (sym _.0))
 (((lambda (_.0) _.0) _.1) (num _.1) (sym _.0))
 ((cons _.0 (cons _.1 _.2)) (num _.0 _.1 _.2))
 (((lambda (_.0) (lambda (_.1) _.2)) _.3)
  (=/= ((_.0 lambda)))
  (num _.3)
  (sym _.0 _.1))
 (((lambda (_.0) _.1) (lambda (_.2) _.3))
  (num _.1)
  (sym _.0 _.2)))
```

using this definition of `eval-expro`:

```
(define (lookupo x env val)
  (fresh (y v env^)
    (symbolo x)
    (symbolo y)
    (== `((,y . ,v) . ,env^) env)
    (conde
      ((== x y) (== v val))
      ((=/= x y) (lookupo x env^ val)))))

(define (not-in-envo x env)
  (fresh ()
    (symbolo x)
    (conde
      ((== '() env))
      ((fresh (y v env^)
         (symbolo y)
         (== `((,y . ,v) . ,env^) env)
```

```
          (=/= x y)
          (not-in-envo x env^))))))

(define (eval-expro expr env val)
  (conde
    ((numbero expr) (== expr val))
    ((fresh (x e)
       (symbolo x)
       (== `(lambda (,x) ,e) expr)
       (== `(closure ,x ,e ,env) val)
       (not-in-envo 'lambda env)))
    ((symbolo expr) (lookupo expr env val))
      ((fresh (e1 e2 v1 v2)
         (== `(cons ,e1 ,e2) expr)
         (== `(,v1 . ,v2) val)
         (eval-expro e1 env v1)
         (eval-expro e2 env v2)))
    ((fresh (e1 e2 x e env^ v)
       (== `(,e1 ,e2) expr)
       (eval-expro e1 env `(closure ,x ,e ,env^))
       (eval-expro e2 env v)
       (eval-expro e `((,x . ,v) . ,env^) val)))))
```

Puzzle 13.1 Ground the answer

Any Scheme integer is a legal value, such as `0`, `42`, or `-1378273284`.

Puzzle 13.2 Ground the answer

`_.0` can be replaced with any legal Scheme symbol, and `_.2` with any Scheme integer. `_.1` can be replaced with any value in the subset of Scheme interpreted by `eval-expro`. Two legal ground answers are:

```
(cons (lambda (z) 42) 17)
```

and:

```
(cons (lambda (lambda) lambda) 0)
```

Puzzle 13.3 Ground the answer

Two legal ground answers are:

```
((lambda (cat) cat) -3)
```

and:

```
((lambda (cons) cons) 19)
```

Puzzle 13.4 Ground the answer

Two legal ground answers are:

```
((lambda (x) (lambda (y) 3)) 4)
```

and:

```
((lambda (cons) (lambda (cons) cons)) 82)
```

However, this is *not* a legal ground answer:

```
((lambda (lambda) (lambda (x) x)) 42)
```

since it violates the side condition `(=/= ((_.0 lambda)))`.