The Reasoned Schemer

The Reasoned Schemer

Daniel P. Friedman
William E. Byrd
Oleg Kiselyov

Drawings by Duane Bibby

The MIT Press
Cambridge, Massachusetts
London, England

MIT Press books may be purchased at special quantity discounts for business or sales promotional use. For information, please e-mail special_sales@mitpress.mit.edu or write to Special Sales Department, The MIT Press, 55 Hayward Street, Cambridge, Mass. 02142.

This book was set in Computer Modern by the authors using LaTeX. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Friedman, Daniel P.
The reasoned schemer / Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov; drawings by Duane Bibby.
p. cm.
Includes index.
ISBN 0-262-56214-6 (pbk. : alk. paper)
1. Scheme (Computer program language) I. Byrd, William E. II. Kiselyov, Oleg. III. Title.
QA76.73.S34F76 2005
005.13′3—dc22

2005051092

10 9 8 7 6 5 4 3 2 1

To Mary, Sarah, Rachel, Shannon and Rob,
and to the memory of Brian.

To Mom, Dad, Brian, Mary, and Renzhong.

((**Contents**)

Preface

The goal of this book is to show the beauty of relational programming. We believe that it is natural to extend functional programming to relational programming. We demonstrate this by extending Scheme with a few new constructs, thereby combining the benefits of both styles. This extension also captures the essence of Prolog, the most well-known logic programming language.

Our main assumption is that you understand the first eight chapters of *The Little Schemer*[1]. The only true requirement, however, is that you understand functions as values. That is, a function can be both an argument to and the value of a function call. Furthermore, you should know that functions remember the context in which they were created. And that's it—we assume no further knowledge of mathematics or logic. Readers of the appendix **Connecting the Wires**, however, must also have a rudimentary knowledge of Scheme macros at the level of **let**, **and**, and **cond**.

In order to do relational programming, we need only two constants: #s and #u, and only three operators: $\equiv$, **fresh**, and **cond**^{e}. These are introduced in the first chapter and are the only operators used until chapter 6. The additional operators we introduce are variants of these three. In order to keep this extension simple, we mimicked existing Scheme syntax. Thus, #s and #u are reminiscent of the Boolean constants: #t and #f; **fresh** expressions resemble **lambda** expressions; and **cond**^{e} expressions are syntactically like **cond** expressions.

We use a few notational conventions throughout the text—primarily changes in font for different classes of symbols. Lexical variables are in *italics*, forms are in **boldface**, data are in sans serif, and lists are wrapped by boldfaced parentheses '**()**'. A relation, a function that returns a goal as its value, ends its name with a superscript 'o' (e.g., car^o and $null^o$). We also use a superscript with our interface to Scheme, **run**, which is fully explained in the first chapter. We have taken certain liberties with punctuation to increase clarity, such as frequently omitting a question mark when a question ends with a special symbol. We do this to avoid confusion with function names that might end with a question mark.

In chapters 7 and 8 we define arithmetic operators as relations. The $+^o$ relation can not only add but also subtract; $*^o$ can not only multiply but also factor numbers; and log^o can not only find the logarithm given a number and a base but also find the base given a logarithm and a number. Just as we can define the subtraction relation from the addition relation, we can define the exponentiation relation from the logarithm relation.

In general, given $(*^o\ x\ y\ z)$ we can specify what we know about these numbers (their values, whether they are odd or even, etc.) and ask $*^o$ to find the unspecified values. We don't specify *how to* accomplish the task; rather, we describe what we want in the result.

[1]Friedman, Daniel P., and Matthias Felleisen. *The Little Schemer, fourth ed.* MIT Press, 1996.

This book would not have been possible without earlier work on implementing and using logic systems with Matthias Felleisen, Anurag Mendhekar, Jon Rossie, Michael Levin, Steve Ganz, and Venkatesh Choppella. Steve showed how to partition Prolog's named relations into unnamed functions, while Venkatesh helped characterize the types in this early logic system. We thank them for their effort during this developmental stage.

There are many others we wish to thank. Mitch Wand struggled through an early draft and spent several days in Bloomington clarifying the semantics of the language, which led to the elimination of superfluous language forms. We also appreciate Kent Dybvig's and Yevgeniy Makarov's comments on the first few chapters of an early draft and Amr Sabry's Haskell implementation of the language.

We gratefully acknowledge Abdulaziz Ghuloum's insistence that we remove some abstract material from the introductory chapter. In addition, Aziz's suggestions significantly clarified the **run** interface. Also incredibly helpful were the detailed criticisms of Chung-chieh Shan, Erik Hilsdale, John Small, Ronald Garcia, Phill Wolf, and Jos Koot. We are especially grateful to Chung-chieh for **Connecting the Wires** so masterfully in the final implementation.

We thank David Mack and Kyle Blocher for teaching this material to students in our undergraduate programming languages course and for making observations that led to many improvements to this book. We also thank those students who not only learned from the material but helped us to clarify its presentation.

There are several people we wish to thank for contributions not directly related to the ideas in the book. We would be remiss if we did not acknowledge Dorai Sitaram's incredibly clever Scheme typesetting program, SLaTeX. We are grateful for Matthias Felleisen's typesetting macros (created for *The Little Schemer*), and for Oscar Waddell's implementation of a tool that selectively expands Scheme macros. Also, we thank Shriram Krishnamurthi for reminding us of a promise we made that the food would be vegetarian in the next *little* book. Finally, we thank Bob Prior, our editor, for his encouragement and enthusiasm for this effort.

Food appears in examples throughout the book for two reasons. First, food is easier to visualize than abstract symbols; we hope the food imagery helps you to better understand the examples and concepts. Second, we want to provide a little distraction. We know how frustrating the subject matter can be, thus these culinary diversions are for whetting your appetite. As such, we hope that thinking about food will cause you to stop reading and have a bite.

You are now ready to start. Good luck! We hope you enjoy the book.

Bon appétit!

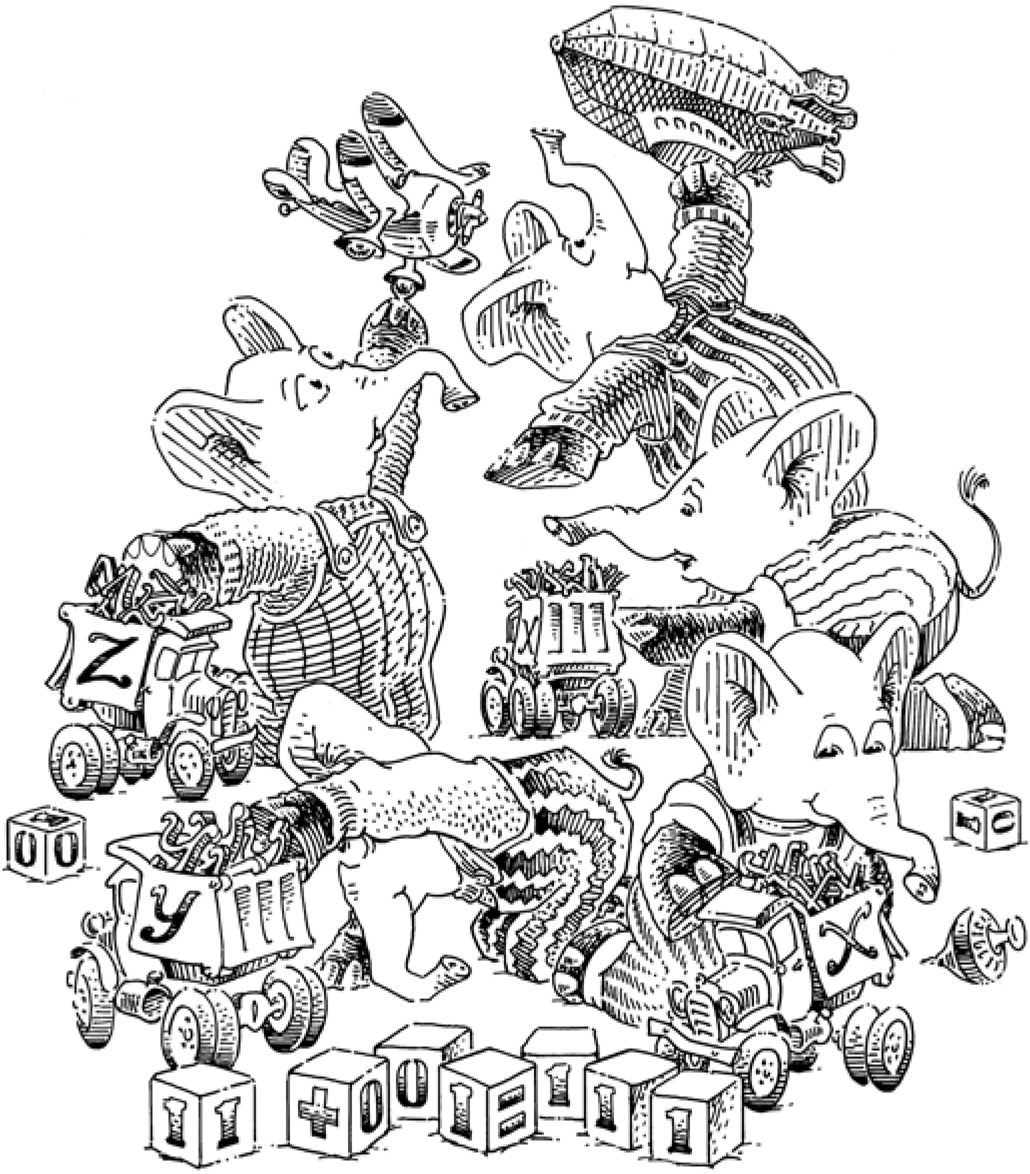
Daniel P. Friedman
William E. Byrd
Bloomington, Indiana

Oleg Kiselyov
Monterey, California

The Reasoned Schemer

1.

Playthings



Welcome.	1	It is good to be here.
Have you read *The Little Schemer*?† † Or *The Little LISPer*.	2	`#f`.
Are you sure you haven't read *The Little Schemer*?	3	Well. . .
Do you know about **Lambda the Ultimate**?	4	`#t`.
Are you sure you have read that much of *The Little Schemer*?	5	Absolutely.† † If you are familiar with recursion and know that functions are values, you may continue anyway.
What is `#s`† † `#s` is written `succeed`.	6	It is a *goal* that succeeds.
What is the name of `#s`	7	*succeed*, because it succeeds.
What is `#u`† † `#u` is written `fail`.	8	It is a goal that fails; it is unsuccessful.

What is the name of `#u`	9	*fail*, because it fails.
What is the value of[†] (**run*** (q) `#u`) † This expression is written (**run** `#f` (q) `#u`).	10	**()**, since `#u` fails, and because the expression[†] (**run*** (q) g ...) has the value **()** if any goal in g ... fails. † This expression is written (**run** `#f` (q) g ...).
What is the value of[†] (**run*** (q) ($\equiv$ `#t` q)) † ($\equiv$ v w) is read "*unify v with w*" and $\equiv$ is written `==`.	11	**(#t)**, because `#t` is *associated with* q if ($\equiv$ `#t` q) succeeds.
What is the value of (**run*** (q) `#u` ($\equiv$ `#t` q))	12	**()**, because the expression (**run*** (q) g ... ($\equiv$ `#t` q)) has the value **()** if the goals g ... fail.
What value is associated with q in (**run*** (q) `#s` ($\equiv$ `#t` q))	13	`#t` (a Boolean[†] value), because the expression (**run*** (q) g ... ($\equiv$ `#t` q)) associates `#t` with q if the goals g ... and ($\equiv$ `#t` q) succeed. † Thank you George Boole (1815–1864).

Then, what is the *value of*

```
(run* (q)
  #s
  (≡ #t q))
```

14 (#t),
because #s succeeds.

What value is associated with r in[†]

```
(run* (r)
  #s
  (≡ corn r))
```

[†] corn is written as the expression (**quote** corn).

15 corn[†],
because r is associated with corn when (≡ corn r) succeeds.

[†] It should be clear from context that corn is a value; it is not an expression. The phrase *the value associated with* corresponds to the phrase *the value of*, but where the outer parentheses have been removed. This is our convention for avoiding meaningless parentheses.

What is the value of

```
(run* (r)
  #s
  (≡ corn r))
```

16 (corn),
because r is associated with corn when (≡ corn r) succeeds.

What is the value of

```
(run* (r)
  #u
  (≡ corn r))
```

17 (),
because #u fails.

What is the value of

```
(run* (q)
  #s
  (≡ #f q))
```

18 (#f),
because #s succeeds and because **run***
returns a nonempty list if its goals succeed.

Does

(≡ #f x)

succeed?

19 It depends on the value of x.

Does

```
(let ((x #t))
  (≡ #f x))†
```

succeed?

[20] No,
since #f is not equal to #t.

† This **let** expression is the same as

```
((lambda (x) (≡ #f x)) #t).
```

We say that **let** *binds* x to #t and evaluates the body (≡ #f x) using that binding.

Does

```
(let ((x #f))
  (≡ #f x))
```

succeed?

[21] Yes,
since #f is equal to #f.

What is the value of

```
(run* (x)
  (let ((x #f))
    (≡ #t x)))
```

[22] **()**,
since #t is not equal to #f.

What value is associated with q in

```
(run* (q)
  (fresh (x)
    (≡ #t x)
    (≡ #t q)))
```

[23] #t,
because '(**fresh** (x ...) g ...)' binds *fresh* variables to x ... and succeeds if the goals g ... succeed. (≡ v x) succeeds when x is fresh.

When is a variable fresh?

[24] When it has no association.

Is x the only variable that starts out fresh in

```
(run* (q)
  (fresh (x)
    (≡ #t x)
    (≡ #t q)))
```

[25] No,
since q also starts out fresh.

The Law of Fresh

If x is fresh, then $(\equiv v\ x)$ succeeds and associates x with v.

What value is associated with q in

```
(run* (q)
  (fresh (x)
    (≡ x #t)
    (≡ #t q)))
```

26 #t,
because the order of arguments to ≡ does not matter.

What value is associated with q in

```
(run* (q)
  (fresh (x)
    (≡ x #t)
    (≡ q #t)))
```

27 #t,
because the order of arguments to ≡ does not matter.

The Law of ≡

$(\equiv v\ w)$ is the same as $(\equiv w\ v)$.

What value is associated with x in

```
(run* (x)
  #s)
```

28 $\_{0}$,
a symbol representing a fresh variable.†

† This symbol is `_.0`, and is created using (*reify-name* `0`). See the definition of *reify-name* in frame 52 of chapter 9 (i.e., 9:52).

What is the value of

```
(run* (x)
  (let ((x #f))
    (fresh (x)
      (≡ #t x))))
```

29 $(\_{0})$,
since the x in (≡ #t x) is the one introduced by the **fresh** expression; it is neither the x introduced in the **run** expression nor the x introduced in the **lambda** expression.

What value is associated with r in

```
(run* (r)
  (fresh (x y)
    (≡ (cons x (cons y ()†)) r)))
```

30 $(\_{0}\ \_{1})$.
For each different fresh variable there is a symbol with an underscore followed by a numeric subscript. This entity is not a variable but rather is a way of showing that the variable was fresh.† We say that such a variable has been *reified*.

† () is (**quote** ()).

† Thank you, Thoralf Albert Skolem (1887–1963).

What value is associated with s in

```
(run* (s)
  (fresh (t u)
    (≡ (cons t (cons u ())) s)))
```

31 $(\_{0}\ \_{1})$.
The expressions in this and the previous frame differ only in the names of the lexical variables. Therefore the values are the same.

What value is associated with r in

```
(run* (r)
  (fresh (x)
    (let ((y x))
      (fresh (x)
        (≡ (cons y (cons x (cons y ()))) r)))))
```

32 $(\_{0}\ \_{1}\ \_{0})$.
Within the inner **fresh**, x and y are different variables, and since they are still fresh, they get different reified names.

What value is associated with r in

```
(run* (r)
  (fresh (x)
    (let ((y x))
      (fresh (x)
        (≡ (cons x (cons y (cons x ()))) r)))))
```

33 $(\_{0}\ \_{1}\ \_{0})$.
x and y are different variables, and since they are still fresh, they get different reified names. Reifying r's value reifies the fresh variables in the order in which they appear in the list.

What is the value of
(**run*** (q)
  ($\equiv$ `#f` q)
  ($\equiv$ `#t` q))
	34	**()**.
		The first goal ($\equiv$ `#f` q) succeeds, associating `#f` with q; `#t` cannot then be associated with q, since q is no longer fresh.

What is the value of
(**run*** (q)
  ($\equiv$ `#f` q)
  ($\equiv$ `#f` q))
	35	**(`#f`)**.
		In order for the **run** to succeed, both ($\equiv$ `#f` q) and ($\equiv$ `#f` q) must succeed. The first goal succeeds while associating `#f` with the fresh variable q. The second goal succeeds because although q is no longer fresh, `#f` is already associated with it.

What value is associated with q in
(**run*** (q)
  (**let** ((x q))
    ($\equiv$ `#t` x)))
	36	`#t`,
		because q and x are the same.

What value is associated with r in
(**run*** (r)
  (**fresh** (x)
    ($\equiv$ x r)†))
	37	$\__0$,
		because r starts out fresh and then r gets whatever association that x gets, but both x and r remain fresh. When one variable is associated with another, we say they *co-refer* or *share*.

What value is associated with q in
(**run*** (q)
  (**fresh** (x)
    ($\equiv$ `#t` x)
    ($\equiv$ x q)))
	38	`#t`,
		because q starts out fresh and then q gets x's association.

What value is associated with q in
(**run*** (q)
  (**fresh** (x)
    ($\equiv$ x q)
    ($\equiv$ `#t` x)))
	39	`#t`,
		because the first goal ensures that whatever association x gets, q also gets.

Are q and x different variables in

```
(run* (q)
  (fresh (x)
    (≡ #t x)
    (≡ x q)))
```

40 Yes, they are different because both

```
(run* (q)
  (fresh (x)
    (≡ (eq? x q) q)))
```

and

```
(run* (q)
  (let ((x q))
    (fresh (q)
      (≡ (eq? x q) x))))
```

associate #f with q. Every variable introduced by **fresh** (or **run**) is different from every other variable introduced by **fresh** (or **run**).†

† Thank you, Jacques Herbrand (1908–1931).

What is the value of

```
(cond
  (#f #t)
  (else #f))
```

41 #f,
because the *question* of the first **cond** line is #f, so the value of the **cond** expression is determined by the *answer* in the second **cond** line.

Which #f is the value?

42 The one in the (**else** #f) **cond** line.

Does

```
(cond
  (#f #s)
  (else #u))
```

succeed?

43 No,
it fails because the answer of the second **cond** line is #u.

Does

```
(conde
  (#u #s)
  (else #u))
```

succeed?†

44 No,
because the question of the first **cond**^{e} line is the goal #u.

† **cond**^{e} is written `conde` and is pronounced “con-dee”. **cond**^{e} is the default control mechanism of Prolog. See William F. Clocksin. *Clause and Effect*. Springer, 1997.

Does

```
(conde
  (#u #u)
  (else #s))
```

succeed?

45 Yes,
because the question of the first **cond**^{e} line is the goal #u, so **cond**^{e} tries the second line.

Does

```
(conde
  (#s #s)
  (else #u))
```

succeed?

46 Yes,
because the question of the first **cond**^{e} line is the goal #s, so **cond**^{e} tries the answer of the first line.

What is the value of

```
(run* (x)
  (conde
    ((≡ olive x) #s)
    ((≡ oil x) #s)
    (else #u)))
```

47 (olive oil),
because (≡ olive x) succeeds; therefore, the answer is #s. The #s preserves the association of x to olive. To get the second value, we *pretend* that (≡ olive x) fails; this imagined failure *refreshes* x. Then (≡ oil x) succeeds. The #s preserves the association of x to oil. We then pretend that (≡ oil x) fails, which once again refreshes x. Since no more goals succeed, we are done.

The Law of cond^e

To get more values from cond^e, pretend that the successful cond^e line has failed, refreshing all variables that got an association from that line.

What does the "*e*" stand for in **cond**^{e}

48 It stands for *every line*, since every line can succeed.

What is the value of†

```
(run¹ (x)
  (condᵉ
    ((≡ olive x) #s)
    ((≡ oil x) #s)
    (else #u)))
```

† This expression is written (**run** 1 (x) ...).

49 (olive),
because (≡ olive x) succeeds and because **run¹** produces at most *one* value.

What is the value of

```
(run* (x)
  (condᵉ
    ((≡ virgin x) #u)
    ((≡ olive x) #s)
    (#s #s)
    ((≡ oil x) #s)
    (else #u)))
```

50 (olive $\_{0}$ oil).
Once the first **cond**^{e} line fails, it is as if that line were not there. Thus what results is identical to

```
(condᵉ
  ((≡ olive x) #s)
  (#s #s)
  ((≡ oil x) #s)
  (else #u)).
```

In the previous **run*** expression, which **cond**^{e} line led to $\_{0}$

51 (#s #s),
since it succeeds without x getting an association.

---

52 What is the value of

```
(run² (x)
  (condᵉ
    ((≡ extra x) #s)
    ((≡ virgin x) #u)
    ((≡ olive x) #s)
    ((≡ oil x) #s)
    (else #u)))
```

† When we give **run** a positive integer n and the **run** expression terminates, it produces a list whose length is less than or equal to n.

(extra olive),
since we do not want every value; we want only the first *two* values.

---

53 What value is associated with r in

```
(run* (r)
  (fresh (x y)
    (≡ split x)
    (≡ pea y)
    (≡ (cons x (cons y ())) r)))
```

(split pea).

---

54 What is the value of

```
(run* (r)
  (fresh (x y)
    (condᵉ
      ((≡ split x) (≡ pea y))
      ((≡ navy x) (≡ bean y))
      (else #u))
    (≡ (cons x (cons y ())) r)))
```

The list ((split pea) (navy bean)).

---

55 What is the value of

```
(run* (r)
  (fresh (x y)
    (condᵉ
      ((≡ split x) (≡ pea y))
      ((≡ navy x) (≡ bean y))
      (else #u))
    (≡ (cons x (cons y (cons soup ()))) r)))
```

The list ((split pea soup) (navy bean soup)).

Consider this very simple definition.

```
(define teacupᵒ
  (lambda (x)
    (condᵉ
      ((≡ tea x) #s)
      ((≡ cup x) #s)
      (else #u))))
```

What is the value of

```
(run* (x)
  (teacupᵒ x))
```

56 (tea cup).

Also, what is the value of

```
(run* (r)
  (fresh (x y)
    (condᵉ
      ((teacupᵒ x) (≡ #t y) #s)†
      ((≡ #f x) (≡ #t y))
      (else #u))
    (≡ (cons x (cons y ())) r)))
```

† The *question* is the first goal of a line, however the *answer* is the rest of the goals of the line. They must all succeed for the line to succeed.

57 ((tea #t) (cup #t) (#f #t)).
From (*teacup*^{o} *x*), *x* gets two associations, and from (≡ #f *x*), *x* gets one association.

What is the value of

```
(run* (r)
  (fresh (x y z)
    (condᵉ
      ((≡ y x) (fresh (x) (≡ z x)))
      ((fresh (x) (≡ y x)) (≡ z x))
      (else #u))
    (≡ (cons y (cons z ())) r)))
```

58 (($\_{0}$ $\_{1}$) ($\_{0}$ $\_{1}$)),
but it looks like both occurrences of $\_{0}$ have come from the same variable and similarly for both occurrences of $\_{1}$.

Then, what is the value of

```
(run* (r)
  (fresh (x y z)
    (conde
      ((≡ y x) (fresh (x) (≡ z x)))
      ((fresh (x) (≡ y x)) (≡ z x))
      (else #u))
    (≡ #f x)
    (≡ (cons y (cons z ())) r)))
```

59 ((#f $_{-0}$) ($_{-0}$ #f)), which clearly shows that the two occurrences of $_{-0}$ in the previous frame represent different variables.

What is the value of

```
(run* (q)
  (let ((a (≡ #t q))
        (b (≡ #f q)))
    b))
```

60 (#f), which shows that (≡ #t q) and (≡ #f q) are expressions, each of whose value is a goal. But, here we only *treat* the (≡ #f q) expression's value, b, as a goal.

What is the value of

```
(run* (q)
  (let ((a (≡ #t q))
        (b (fresh (x)
             (≡ x q)
             (≡ #f x)))
        (c (conde
             ((≡ #t q) #s)
             (else (≡ #f q)))))
    b))
```

61 (#f), which shows that (≡ . . .), (**fresh** . . .), and (**cond**^{e} . . .) are expressions, each of whose value is a goal. But, here we only treat the **fresh** expression's value, b, as a goal. This is indeed interesting.

⟹ Now go make yourself a peanut butter and jam sandwich. ⟸

This space reserved for

JAM STAINS!

2.
Teaching Old Toys New Tricks

What is the value of

(**let** ((x (**lambda** (a) a))
(y c))
(x y))

1 c,
because (x y) applies (**lambda** (a) a) to c.

What value is associated with r in

(**run*** (r)
(**fresh** (y x)
(≡ **(**x y**)**† r)))

† This list is written as the expression `(,x ,y) or (*cons* x (*cons* y **()**)). This list is distinguished from the function application (x y) by the use of bold parentheses.

2 **(**$\_{0}$ $\_{1}$**)**†,
because the variables in **(**x y**)** have been introduced by **fresh**.

† It should be clear from context that this list is a value; it is not an expression. This list could have been built (see 9:52) using (*cons* (*reify-name* 0) (*cons* (*reify-name* 1) **()**)).

What is the value of

(**run*** (r)
(**fresh** (v w)
(≡ (**let** ((x v) (y w)) **(**x y**)**) r)))

3 **((**$\_{0}$ $\_{1}$**))**,
because v and w are variables introduced by **fresh**.

What is the value of

(*car* **(**grape raisin pear**)**)

4 grape.

What is the value of

(*car* **(**a c o r n**)**)

5 a.

What value is associated with r in†

(**run*** (r)
(*car*^{o} **(**a c o r n**)** r))

† *car*^{o} is written caro and is pronounced "car-oh". Henceforth, consult the index for how we write the names of functions.

6 a,
because a is the *car* of **(**a c o r n**)**.

What value is associated with q in

```
(run* (q)
  (car^o (a c o r n) a)
  (≡ #t q))
```

7 #t,
because a is the *car* of (a c o r n).

What value is associated with r in

```
(run* (r)
  (fresh (x y)
    (car^o (r y) x)
    (≡ pear x)))
```

8 pear,
since x is associated with the *car* of (r y), which is the fresh variable r. Then x is associated with pear, which in turn associates r with pear.

Here is the definition of car^o.

```
(define car^o
  (lambda (p a)
    (fresh (d)
      (≡ (cons a d) p))))
```

What is unusual about this definition?

9 Whereas *car* takes one argument, car^o takes two.

What is the value of

```
(cons
  (car (grape raisin pear))
  (car ((a) (b) (c))))
```

10 That's easy: (grape a).

What value is associated with r in

```
(run* (r)
  (fresh (x y)
    (car^o (grape raisin pear) x)
    (car^o ((a) (b) (c)) y)
    (≡ (cons x y) r)))
```

11 That's the same: (grape a).

Why can we use *cons*

12 Because variables introduced by **fresh** *are* values, and each argument to *cons* can be any value.

13 What is the value of

(*cdr* (grape raisin pear))

That's easy: (raisin pear).

---

14 What is the value of

(*car* (*cdr* (a c o r n)))

c.

---

15 What value is associated with r in

```
(run* (r)
  (fresh (v)
    (cdrᵒ (a c o r n) v)
    (carᵒ v r)))
```

c.

The process of transforming (*car* (*cdr* l)) into (*cdr*^{o} l v) and (*car*^{o} v r) is called *unnesting*.†

† Some readers may recognize the similarity between unnesting and continuation-passing style.

---

16 Here is the definition of *cdr*^{o}.

```
(define cdrᵒ
  (lambda (p d)
    (fresh (a)
      (≡ (cons a d) p))))
```

Oh. It is *almost* the same as *car*^{o}.

---

17 What is the value of

```
(cons
  (cdr (grape raisin pear))
  (car ((a) (b) (c))))
```

That's easy: ((raisin pear) a).

---

18 What value is associated with r in

```
(run* (r)
  (fresh (x y)
    (cdrᵒ (grape raisin pear) x)
    (carᵒ ((a) (b) (c)) y)
    (≡ (cons x y) r)))
```

That's the same: ((raisin pear) a).

What value is associated with q in

```
(run* (q)
  (cdrᵒ (a c o r n) (c o r n))
  (≡ #t q))
```

[19] #t,
because (c o r n) is the *cdr* of (a c o r n).

What value is associated with x in

```
(run* (x)
  (cdrᵒ (c o r n) (x r n)))
```

[20] o,
because (o r n) is the *cdr* of (c o r n), so x gets associated with o.

What value is associated with l in

```
(run* (l)
  (fresh (x)
    (cdrᵒ l (c o r n))
    (carᵒ l x)
    (≡ a x)))
```

[21] (a c o r n),
because if the *cdr* of l is (c o r n), then l must be the list (a c o r n), where a is the fresh variable introduced in the definition of cdr^o. Taking the car^o of l associates the *car* of l with x. When we associate x with a, we also associate a, the *car* of l, with a, so l is associated with the list (a c o r n).

What value is associated with l in

```
(run* (l)
  (consᵒ (a b c) (d e) l))
```

[22] ((a b c) d e),
since $cons^o$ associates l with (*cons* (a b c) (d e)).

What value is associated with x in

```
(run* (x)
  (consᵒ x (a b c) (d a b c)))
```

[23] d.
Since (*cons* d (a b c)) is (d a b c), $cons^o$ associates x with d.

What value is associated with r in

```
(run* (r)
  (fresh (x y z)
    (≡ (e a d x) r)
    (consᵒ y (a z c) r)))
```

[24] (e a d c),
because first we associate r with a list whose last element is the fresh variable x. We then perform the $cons^o$, associating x with c, z with d, and y with e.

What value is associated with x in

```
(run* (x)
  (consᵒ x (a x c) (d a x c)))
```

[25] d.
What value can we associate with x so that (*cons* x (a x c)) is (d a x c)? Obviously, d is the value.

What value is associated with l in

```
(run* (l)
  (fresh (x)
    (≡ (d a x c) l)
    (conso x (a x c) l)))
```

26 (d a d c),
because l is (d a x c). Then when we *cons*^{o} x onto (a x c), we associate x with d.

What value is associated with l in

```
(run* (l)
  (fresh (x)
    (conso x (a x c) l)
    (≡ (d a x c) l)))
```

27 (d a d c),
because we *cons* x onto (a x c), and associate l with the list (x a x c). Then when we associate l with (d a x c), we associate x with d.

Define *cons*^{o} using ≡.

28

```
(define conso
  (lambda (a d p)
    (≡ (cons a d) p)))
```

What value is associated with l in

```
(run* (l)
  (fresh (d x y w s)
    (conso w (a n s) s)
    (cdro l s)
    (caro l x)
    (≡ b x)
    (cdro l d)
    (caro d y)
    (≡ e y)))
```

29 (b e a n s).
l must clearly be a five element list, since s is (*cdr* l). Since l is fresh, (*cdr*^{o} l s) places a fresh variable in the first position of l, while associating w and (a n s) with the second position and the *cdr* of the *cdr* of l, respectively. The first variable in l gets associated with x, which in turn gets associated with b. The *cdr* of l is a list whose *car* is the variable w. That variable gets associated with y, which in turn gets associated with e.

What is the value of

(*null?* (grape raisin pear))

30 #f.

What is the value of

(*null?* ())

31 #t.

What is the value of

```
(run* (q)
  (nullᵒ (grape raisin pear))
  (≡ #t q))
```

32 ().

What is the value of

```
(run* (q)
  (nullᵒ ())
  (≡ #t q))
```

33 (#t).

What is the value of

```
(run* (x)
  (nullᵒ x))
```

34 (()).

Define $null^o$ using ≡.

35

```
(define nullᵒ
  (lambda (x)
    (≡ () x)))
```

What is the value of

```
(eq? pear plum)
```

36 #f.

What is the value of

```
(eq? plum plum)
```

37 #t.

What is the value of

```
(run* (q)
  (eqᵒ pear plum)
  (≡ #t q))
```

38 ().

What is the value of

```
(run* (q)
  (eqᵒ plum plum)
  (≡ #t q))
```

39 (#t).

Define eq^o using ≡.

40 It is easy.

```
(define eqᵒ
  (lambda (x y)
    (≡ x y)))
```

Is (split . pea) a pair?

41 Yes.

Is (split . *x*) a pair?

42 Yes.

What is the value of

(*pair?* ((split) . pea))

43 #t.

What is the value of

(*pair?* ())

44 #f.

Is pair a pair?

45 No.

Is pear a pair?

46 No.

Is (pear) a pair?

47 Yes,
it is the pair (pear . ()).

What is the value of

(*car* (pear))

48 pear.

What is the value of

(*cdr* (pear))

49 ().

How can we build these pairs?

50 Use *Cons the Magnificent*.

What is the value of

(*cons* (split) pea)

51 ((split) . pea).

What value is associated with r in

```
(run* (r)
  (fresh (x y)
    (≡ (cons x (cons y salad)) r)))
```

52 ($\_{0}$ $\_{1}$. salad).

Here is the definition of $pair^o$.

```
(define pairᵒ
  (lambda (p)
    (fresh (a d)
      (consᵒ a d p))))
```

Is $pair^o$ recursive?

53 No, it is not.

What is the value of

```
(run* (q)
  (pairᵒ (cons q q))
  (≡ #t q))
```

54 (#t).

What is the value of

```
(run* (q)
  (pairᵒ ())
  (≡ #t q))
```

55 ().

What is the value of

```
(run* (q)
  (pair^o pair)
  (≡ #t q))
```

[56] ().

What value is associated with x in

```
(run* (x)
  (pair^o x))
```

[57] $(\_{0} \cdot \_{1})$.

What value is associated with r in

```
(run* (r)
  (pair^o (cons r pear)))
```

[58] $\_{0}$.

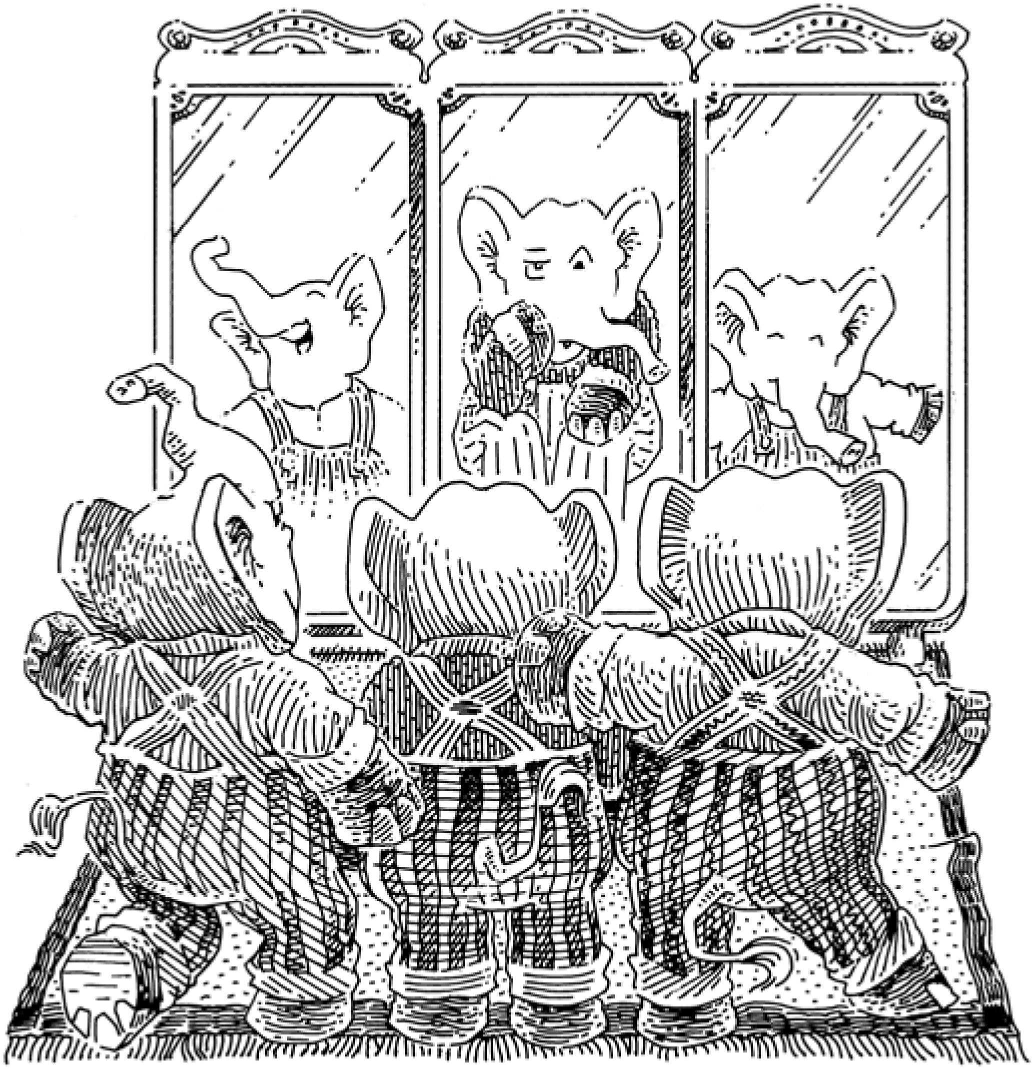
Is it possible to define car^o, cdr^o, and $pair^o$ using $cons^o$

[59] Yes.

This space reserved for

"Cons^o the Magnificent^o"

3.
Seeing Old Friends in New Ways



Consider the definition of *list?*.

```
(define list?
  (lambda (l)
    (cond
      ((null? l) #t)
      ((pair? l) (list? (cdr l)))
      (else #f))))
```

What is the value of

(*list?* ((a) (a b) c))

1 #t.

What is the value of

(*list?* ())

2 #t.

What is the value of

(*list?* s)

3 #f.

What is the value of

(*list?* (d a t e . s))

4 #f,
because (d a t e . s) is not a proper list.†

† A list is *proper* if it is the empty list or if its *cdr* is proper.

Consider the definition of $list^o$.

```
(define listᵒ
  (lambda (l)
    (condᵉ
      ((nullᵒ l) #s)
      ((pairᵒ l)
       (fresh (d)
         (cdrᵒ l d)
         (listᵒ d)))
      (else #u))))
```

How does $list^o$ differ from *list?*

5 The definition of *list?* has Boolean values as questions and answers. $list^o$ has goals as questions† and answers. Hence, it uses **cond**^{e} instead of **cond**.

† **else** is like #t in a **cond** line, whereas **else** is like #s in a **cond**^{e} line.

Where does

(**fresh** (d)
  (cdr^o l d)
  ($list^o$ d))

come from?

6 It is an unnesting of (*list?* (*cdr* l)). First we take the *cdr* of l and associate it with a fresh variable d, and then we use d in the recursive call.

The First Commandment

To transform a function whose value is a Boolean into a function whose value is a goal, replace cond with cond^e and unnest each question and answer. Unnest the answer #t (or #f) by replacing it with #s (or #u).

What value is associated with x in

(**run*** (x)
  ($list^o$ (a b x d)[†]))

where a, b, and d are symbols, and x is a variable.

† Reminder: This is the same as `(a b ,x d).

7 $\_{0}$,
since x remains fresh.

Why is $\_{0}$ the value associated with x in

(**run*** (x)
  ($list^o$ (a b x d)))

8 When determining the goal returned by $list^o$, it is not necessary to determine the value of x. Therefore x remains fresh, which means that the goal returned from the call to $list^o$ succeeds *for all* values associated with x.

How is $\_{0}$ the value associated with x in

(**run*** (x)
  ($list^o$ (a b x d)))

9 When $list^o$ reaches the end of its argument, it succeeds. But x does not get associated with any value.

What value is associated with x in (**run¹** (x) (*list°* (a b c . x)))	10 ().
Why is () the value associated with x in (**run¹** (x) (*list°* (a b c . x)))	11 Because (a b c . x) is a proper list when x is the empty list.
How is () the value associated with x in (**run¹** (x) (*list°* (a b c . x)))	12 When *list°* reaches the end of (a b c . x), (*null°* x) succeeds and associates x with the empty list.
What is the value of (**run*** (x) (*list°* (a b c . x)))	13 It has *no value.* Maybe we should use **run⁵** to get the first five values.
What is the value of (**run⁵** (x) (*list°* (a b c . x)))	14 (() ($\_{0}$) ($\_{0}$ $\_{1}$) ($\_{0}$ $\_{1}$ $\_{2}$) ($\_{0}$ $\_{1}$ $\_{2}$ $\_{3}$)).
Describe what we have seen in transforming *list?* into *list°*.	15 In *list?* each **cond** line results in a value, whereas in *list°* each **cond**^{e} line results in a goal. To have each **cond**^{e} result in a goal, we unnest each **cond** question and each **cond** answer. Used with recursion, a **cond**^{e} expression can produce an unbounded number of values. We have used an upper bound, 5 in the previous frame, to keep from creating a list with an unbounded number of values.

Consider the definition of *lol?*, where *lol?* stands for *list-of-lists?*.

```
(define lol?
  (lambda (l)
    (cond
      ((null? l) #t)
      ((list? (car l)) (lol? (cdr l)))
      (else #f))))
```

Describe what *lol?* does.

16 As long as each top-level value in the list l is a proper list, *lol?* returns #t. Otherwise, *lol?* returns #f.

Here is the definition of lol^o.

```
(define lolᵒ
  (lambda (l)
    (condᵉ
      ((nullᵒ l) #s)
      ((fresh (a)
         (carᵒ l a)
         (listᵒ a))
        (fresh (d)
          (cdrᵒ l d)
          (lolᵒ d)))
      (else #u))))
```

How does lol^o differ from *lol?*

17 The definition of *lol?* has Boolean values as questions and answers. lol^o has goals as questions and answers. Hence, it uses **cond**^{e} instead of **cond**.

What else is different?

18 (*list?* (*car* l)) and (*lol?* (*cdr* l)) have been unnested.

Is the value of (lol^o l) always a goal?

19 Yes.

What is the value of

```
(run¹ (l)
  (lolᵒ l))
```

20 **(())**.
Since l is fresh, ($null^o$ l) succeeds and in the process associates l with **()**.

What value is associated with q in

```
(run* (q)
  (fresh (x y)
    (lolᵒ ((a b) (x c) (d y)))
    (≡ #t q)))
```

21 #t,
since ((a b) (x c) (d y)) is a list of lists.

What value is associated with q in

```
(run¹ (q)
  (fresh (x)
    (lolᵒ ((a b) . x))
    (≡ #t q)))
```

22 #t,
because *null*^{o} of a fresh variable always succeeds and associates the fresh variable, in this case x, with ().

What is the value of

```
(run¹ (x)
  (lolᵒ ((a b) (c d) . x)))
```

23 (()),
since replacing x with the empty list in ((a b) (c d) . x) transforms it to ((a b) (c d) . ()), which is the same as ((a b) (c d)).

What is the value of

```
(run⁵ (x)
  (lolᵒ ((a b) (c d) . x)))
```

24
```
(()
 (())
 (() ())
 (() () ())
 (() () () ())).
```

What do we get when we replace x by the last list in the previous frame?

25 ((a b) (c d) . (() () () ())),
which is the same as

((a b) (c d) () () () ()).

Is (tofu tofu) a *twin*?

26 Yes,
because it is a list of two identical values.

Is (e tofu) a twin?

27 No,
because e and tofu differ.

Is (g g g) a twin?

[28] No,
because it is not a list of two values.

Is ((g g) (tofu tofu)) a list of twins?

[29] Yes,
since both (g g) and (tofu tofu) are twins.

Is ((g g) (e tofu)) a list of twins?

[30] No,
since (e tofu) is not a twin.

Consider the definition of *twins^o*.

```
(define twins^o
  (lambda (s)
    (fresh (x y)
      (cons^o x y s)
      (cons^o x () y))))
```

Is *twins^o* recursive?

[31] No, it isn't.

What value is associated with q in

```
(run* (q)
  (twins^o (tofu tofu))
  (≡ #t q))
```

[32] #t.

What value is associated with z in

```
(run* (z)
  (twins^o (z tofu)))
```

[33] tofu.

Why is tofu the value associated with z in

```
(run* (z)
  (twins^o (z tofu)))
```

[34] Because (z tofu) is a twin only when z is associated with tofu.

How is tofu the value associated with z in

```
(run* (z)
  (twinso (z tofu)))
```

35 In the call to *twins*^{o} the first *cons*^{o} associates x with the *car* of $(z$ tofu$)$, which is z, and associates y with the *cdr* of $(z$ tofu$)$, which is (tofu). Remember that (tofu) is the same as (tofu . ()). The second *cons*^{o} associates x, and therefore z, with the *car* of y, which is tofu.

Redefine *twins*^{o} without using *cons*^{o}.

36 Here it is.

```
(define twinso
  (lambda (s)
    (fresh (x)
      (≡ (x x) s))))
```

Consider the definition of *lot*^{o}.

```
(define loto
  (lambda (l)
    (conde
      ((nullo l) #s)
      ((fresh (a)
          (caro l a)
          (twinso a))
        (fresh (d)
          (cdro l d)
          (loto d)))
      (else #u))))
```

What does *lot* stand for?

37 *lot* stands for *list-of-twins*.

What value is associated with z in

```
(run1 (z)
  (loto ((g g) . z)))
```

38 ().

Why is () the value associated with z in

```
(run1 (z)
  (loto ((g g) . z)))
```

39 Because ((g g) . z) is a list of twins when z is the empty list.

What do we get when we replace z by **()**

40 **((g g) . ())**,

which is the same as

((g g)).

How is **()** the value associated with z in

(**run**$^{\mathbf{1}}$ (z)
  (*lot*^{o} **((g g) .** z**)**))

41 In the first call to *lot*^{o}, l is the list **((g g) .** z**)**. Since this list is not null, (*null*^{o} l) fails and we move on to the second **cond**^{e} line. In the second **cond**^{e} line, d is associated with the *cdr* of **((g g) .** z**)**, which is z. The variable d is then passed in the recursive call to *lot*^{o}. Since the variable z associated with d is fresh, (*null*^{o} l) succeeds and associates d and therefore z with the empty list.

What is the value of

(**run**$^{\mathbf{5}}$ (z)
  (*lot*^{o} **((g g) .** z**)**))

42 (()
 (($\_{0}$ $\_{0}$))
 (($\_{0}$ $\_{0}$) ($\_{1}$ $\_{1}$))
 (($\_{0}$ $\_{0}$) ($\_{1}$ $\_{1}$) ($\_{2}$ $\_{2}$))
 (($\_{0}$ $\_{0}$) ($\_{1}$ $\_{1}$) ($\_{2}$ $\_{2}$) ($\_{3}$ $\_{3}$))).

Why are the nonempty values **(**$\_{n}$ $\_{n}$**)**

43 Each $\_{n}$ corresponds to a fresh variable that has been introduced in the question of the second **cond**^{e} line of *lot*^{o}.

What do we get when we replace z by the fourth list in frame 42?

44 **((g g) . ((**$\_{0}$ $\_{0}$**) (**$\_{1}$ $\_{1}$**) (**$\_{2}$ $\_{2}$**)))**,

which is the same as

((g g) ($\_{0}$ $\_{0}$**) (**$\_{1}$ $\_{1}$**) (**$\_{2}$ $\_{2}$**))**.

What is the value of

(**run**$^{\mathbf{5}}$ (r)
  (**fresh** (w x y z)
    (*lot*^{o} **((g g) (e** w**) (**x y**) .** z**)**)
    ($\equiv$ **(**w **(**x y**)** z**)** r)))

45 ((e ($\_{0}$ $\_{0}$) ())
 (e ($\_{0}$ $\_{0}$) (($\_{1}$ $\_{1}$)))
 (e ($\_{0}$ $\_{0}$) (($\_{1}$ $\_{1}$) ($\_{2}$ $\_{2}$)))
 (e ($\_{0}$ $\_{0}$) (($\_{1}$ $\_{1}$) ($\_{2}$ $\_{2}$) ($\_{3}$ $\_{3}$)))
 (e ($\_{0}$ $\_{0}$) (($\_{1}$ $\_{1}$) ($\_{2}$ $\_{2}$) ($\_{3}$ $\_{3}$) ($\_{4}$ $\_{4}$)))).

46 What do we get when we replace w, x, y, and z by the third list in the previous frame?

((g g) (e e) ($\_0$ $\_0$) . (($\_1$ $\_1$) ($\_2$ $\_2$))), which is the same as

((g g) (e e) ($\_0$ $\_0$) ($\_1$ $\_1$) ($\_2$ $\_2$)).

---

47 What is the value of

```
(run³ (out)
  (fresh (w x y z)
    (≡ ((g g) (e w) (x y) . z) out)
    (lotᵒ out)))
```

(((g g) (e e) ($\_0$ $\_0$))
 ((g g) (e e) ($\_0$ $\_0$) ($\_1$ $\_1$))
 ((g g) (e e) ($\_0$ $\_0$) ($\_1$ $\_1$) ($\_2$ $\_2$))).

---

48 Here is *listof*^{o}.

```
(define listofᵒ
  (lambda (predᵒ l)
    (condᵉ
      ((nullᵒ l) #s)
      ((fresh (a)
         (carᵒ l a)
         (predᵒ a))
       (fresh (d)
         (cdrᵒ l d)
         (listofᵒ predᵒ d)))
      (else #u))))
```

Is *listof*^{o} recursive?

Yes.

---

49 What is the value of

```
(run³ (out)
  (fresh (w x y z)
    (≡ ((g g) (e w) (x y) . z) out)
    (listofᵒ twinsᵒ out)))
```

(((g g) (e e) ($\_0$ $\_0$))
 ((g g) (e e) ($\_0$ $\_0$) ($\_1$ $\_1$))
 ((g g) (e e) ($\_0$ $\_0$) ($\_1$ $\_1$) ($\_2$ $\_2$))).

---

50 Now redefine *lot*^{o} using *listof*^{o} and *twins*^{o}.

That's simple.

```
(define lotᵒ
  (lambda (l)
    (listofᵒ twinsᵒ l)))
```

Remember *member?*

```
(define member?
  (lambda (x l)
    (cond
      ((null? l) #f)
      ((eq-car? l x) #t)
      (else (member? x (cdr l))))))
```

Define *eq-car?*.

51 *member?* is an old friend, but that's a strange way to define it.

```
(define eq-car?
  (lambda (l x)
    (eq? (car l) x)))
```

Don't worry. It will make sense soon.

52 Okay.

What is the value of

(*member?* olive (virgin olive oil))

53 #t, but this is uninteresting.

Consider this definition of *eq-car*^{o}.

```
(define eq-carᵒ
  (lambda (l x)
    (carᵒ l x)))
```

Define *member*^{o} using *eq-car*^{o}.

54

```
(define memberᵒ
  (lambda (x l)
    (condᵉ
      ((nullᵒ l) #u)
      ((eq-carᵒ l x) #s)
      (else
        (fresh (d)
          (cdrᵒ l d)
          (memberᵒ x d))))))
```

Is the first **cond**^{e} line unnecessary?

55 Yes.
Whenever a **cond**^{e} line is guaranteed to fail, it is unnecessary.

Which expression has been unnested?

56 (*member?* *x* (*cdr* *l*)).

What value is associated with *q* in

```
(run* (q)
  (memberᵒ olive (virgin olive oil))
  (≡ #t q))
```

57 #t,
because (*member*^{o} *a* *l*) succeeds, but this is still uninteresting.

What value is associated with y in

(**run¹** (y)
  ($member^o$ y (hummus with pita)))

[58] hummus,
because we can ignore the first **cond**^{e} line since l is not the empty list, and because the second **cond**^{e} line associates the fresh variable y with the value of (*car* l), which is hummus.

What value is associated with y in

(**run¹** (y)
  ($member^o$ y (with pita)))

[59] with,
because we can ignore the first **cond**^{e} line since l is not the empty list, and because the second **cond**^{e} line associates the fresh variable y with the value of (*car* l), which is with.

What value is associated with y in

(**run¹** (y)
  ($member^o$ y (pita)))

[60] pita,
because we can ignore the first **cond**^{e} line since l is not the empty list, and because the second **cond**^{e} line associates the fresh variable y with the value of (*car* l), which is pita.

What is the value of

(**run*** (y)
  ($member^o$ y ()))

[61] (),
because the ($null^o$ l) question of the first **cond**^{e} line now holds, resulting in failure of the goal ($member^o$ y l).

What is the value of

(**run*** (y)
  ($member^o$ y (hummus with pita)))

[62] (hummus with pita),
since we already know the value of each recursive call to $member^o$, provided y is fresh.

Why is y a fresh variable each time we enter $member^o$ recursively?

[63] Since we pretend that the second **cond**^{e} line has failed, we also get to assume that y has been refreshed.

So is the value of

```
(run* (y)
  (memberᵒ y l))
```

always the value of l

64 Yes.

Using **run***, define a function called *identity* whose argument is a list, and which returns that list.

65

```
(define identity
  (lambda (l)
    (run* (y)
      (memberᵒ y l))))
```

What value is associated with x in

```
(run* (x)
  (memberᵒ e (pasta x fagioli)))
```

66 e.

The list contains three values with a variable in the middle. The *member*^{o} function determines that x's value should be e.

Why is e the value associated with x in

```
(run* (x)
  (memberᵒ e (pasta x fagioli)))
```

67 Because (*member*^{o} e (pasta e fagioli)) succeeds.

What have we just done?

68 We filled in a blank in the list so that *member*^{o} succeeds.

What value is associated with x in

```
(run¹ (x)
  (memberᵒ e (pasta e x fagioli)))
```

69 $\_{0}$,

because the recursion succeeds *before* it gets to the variable x.

What value is associated with x in

```
(run¹ (x)
  (memberᵒ e (pasta x e fagioli)))
```

70 e,

because the recursion succeeds *when* it gets to the variable x.

What is the value of
(**run*** (*r*)
  (**fresh** (*x* *y*)
    (*member*^{o} e (pasta *x* fagioli *y*))
    (≡ (*x* *y*) *r*)))
	71	((e $_{0}$) ($_{0}$ e)).

What does each value in the list mean?	72	There are two values in the list. We know from frame 70 that when *x* gets associated with e, (*member*^{o} e (pasta *x* fagioli *y*)) succeeds, leaving *y* fresh. Then *x* is refreshed. For the second value, *y* gets an association, but *x* does not.
What is the value of (**run**$^{\mathbf{1}}$ (*l*) (*member*^{o} tofu *l*))	73	((tofu . $_{0}$)).
Which lists are represented by (tofu . $_{0}$)	74	Every list whose *car* is tofu.
What is the value of (**run*** (*l*) (*member*^{o} tofu *l*))	75	It has no value, because **run*** never finishes building the list.
What is the value of (**run**$^{\mathbf{5}}$ (*l*) (*member*^{o} tofu *l*))	76	((tofu . $_{0}$) ($_{0}$ tofu . $_{1}$) ($_{0}$ $_{1}$ tofu . $_{2}$) ($_{0}$ $_{1}$ $_{2}$ tofu . $_{3}$) ($_{0}$ $_{1}$ $_{2}$ $_{3}$ tofu . $_{4}$)). Clearly each list satisfies *member*^{o}, since tofu is in every list.

Explain why the answer is

((tofu . $\_{0}$)
($\_{0}$ tofu . $\_{1}$)
($\_{0}$ $\_{1}$ tofu . $\_{2}$)
($\_{0}$ $\_{1}$ $\_{2}$ tofu . $\_{3}$)
($\_{0}$ $\_{1}$ $\_{2}$ $\_{3}$ tofu . $\_{4}$))

77 Assume that we know how the first four lists are determined. Now we address how the fifth list appears. When we pretend that *eq-car*^{o} fails, l is refreshed and the last **cond**^{e} line is tried. l is refreshed, but we recur on its *cdr*, which is also fresh. So each value becomes one longer than the previous value. In the recursive call (*member*^{o} *x* *d*), the call to *eq-car*^{o} associates tofu with the *car* of the *cdr* of l. Thus $\_{3}$ will appear where tofu appeared in the fourth list.

Is it possible to remove the dotted variable at the end of each list, making it proper?

78 Perhaps,
but we do know when we've found the value we're looking for.

Yes, that's right. That should give us enough of a clue. What should the *cdr* be when we find this value?

79 It should be the empty list if we find the value at the end of the list.

Here is a definition of *pmember*^{o}.

```
(define pmemberᵒ
  (lambda (x l)
    (condᵉ
      ((nullᵒ l) #u)
      ((eq-carᵒ l x) (cdrᵒ l ()))
      (else
        (fresh (d)
          (cdrᵒ l d)
          (pmemberᵒ x d))))))
```

What is the value of

```
(run⁵ (l)
  (pmemberᵒ tofu l))
```

80 ((tofu)
($\_{0}$ tofu)
($\_{0}$ $\_{1}$ tofu)
($\_{0}$ $\_{1}$ $\_{2}$ tofu)
($\_{0}$ $\_{1}$ $\_{2}$ $\_{3}$ tofu)).

What is the value of

```
(run* (q)
  (pmemberᵒ tofu (a b tofu d tofu))
  (≡ #t q))
```

81 Is it (#t #t)?

No, the value is (#t). Explain why.

82 The test for being at the end of the list caused this definition to miss the first tofu.

Here is a refined definition of *pmember*^{o}.

```
(define pmemberᵒ
  (lambda (x l)
    (condᵉ
      ((nullᵒ l) #u)
      ((eq-carᵒ l x) (cdrᵒ l ()))
      ((eq-carᵒ l x) #s)
      (else
        (fresh (d)
          (cdrᵒ l d)
          (pmemberᵒ x d))))))
```

How does this refined definition differ from the original definition of *pmember*^{o}

83 We have included an additional **cond**^{e} line that succeeds when the *car* of *l* matches *x*.

What is the value of

```
(run* (q)
  (pmemberᵒ tofu (a b tofu d tofu))
  (≡ #t q))
```

84 Is it (#t #t)?

No, the value is (#t #t #t). Explain why.

85 The second **cond**^{e} line contributes a value because there is a tofu at the end of the list. Then the third **cond**^{e} line contributes a value for the first tofu in the list and it contributes a value for the second tofu in the list. Thus in all, three values are contributed.

Here is a more refined definition of *pmember*^{o}.

```
(define pmemberᵒ
  (lambda (x l)
    (condᵉ
      ((nullᵒ l) #u)
      ((eq-carᵒ l x) (cdrᵒ l ()))
      ((eq-carᵒ l x)
       (fresh (a d)
         (cdrᵒ l (a . d))))
      (else
        (fresh (d)
          (cdrᵒ l d)
          (pmemberᵒ x d))))))
```

How does this definition differ from the previous definition of *pmember*^{o}

86 We have included a test to make sure that its *cdr* is not the empty list.

How can we simplify this definition a bit more?

87 We know that a **cond**^{e} line that always fails, like the first **cond**^{e} line, can be removed.

Now what is the value of

```
(run* (q)
  (pmemberᵒ tofu (a b tofu d tofu))
  (≡ #t q))
```

88 (#t #t) as expected.

Now what is the value of

```
(run¹² (l)
  (pmemberᵒ tofu l))
```

89
```
((tofu)
 (tofu _₀ . _₁)
 (_₀ tofu)
 (_₀ tofu _₁ . _₂)
 (_₀ _₁ tofu)
 (_₀ _₁ tofu _₂ . _₃)
 (_₀ _₁ _₂ tofu)
 (_₀ _₁ _₂ tofu _₃ . _₄)
 (_₀ _₁ _₂ _₃ tofu)
 (_₀ _₁ _₂ _₃ tofu _₄ . _₅)
 (_₀ _₁ _₂ _₃ _₄ tofu)
 (_₀ _₁ _₂ _₃ _₄ tofu _₅ . _₆)).
```

90 How can we characterize this list of values?

All of the odd positions are proper lists.

91 Why are the odd positions proper lists?

Because in the second **cond**^{e} line the *cdr* of *l* is the empty list.

92 Why are the even positions improper lists?

Because in the third **cond**^{e} line the *cdr* of *l* is a pair.

93 How can we redefine *pmember*^{o} so that the lists in the odd and even positions are swapped?

We merely swap the first two **cond**^{e} lines of the simplified definition.

```
(define pmember°
  (lambda (x l)
    (conde
      ((eq-car° l x)
       (fresh (a d)
         (cdr° l (a . d))))
      ((eq-car° l x) (cdr° l ()))
      (else
        (fresh (d)
          (cdr° l d)
          (pmember° x d)))))))
```

94 Now what is the value of

```
(run12 (l)
  (pmember° tofu l))
```

$((\text{tofu}\ \_{0} \, . \, \_{1})$
(tofu)
$(\_{0}\ \text{tofu}\ \_{1} \, . \, \_{2})$
$(\_{0}\ \text{tofu})$
$(\_{0}\ \_{1}\ \text{tofu}\ \_{2} \, . \, \_{3})$
$(\_{0}\ \_{1}\ \text{tofu})$
$(\_{0}\ \_{1}\ \_{2}\ \text{tofu}\ \_{3} \, . \, \_{4})$
$(\_{0}\ \_{1}\ \_{2}\ \text{tofu})$
$(\_{0}\ \_{1}\ \_{2}\ \_{3}\ \text{tofu}\ \_{4} \, . \, \_{5})$
$(\_{0}\ \_{1}\ \_{2}\ \_{3}\ \text{tofu})$
$(\_{0}\ \_{1}\ \_{2}\ \_{3}\ \_{4}\ \text{tofu}\ \_{5} \, . \, \_{6})$
$(\_{0}\ \_{1}\ \_{2}\ \_{3}\ \_{4}\ \text{tofu}))$.

Consider the definition of *first-value*, which takes a list of values *l* and returns a list that contains the first value in *l*.

```
(define first-value
  (lambda (l)
    (run¹ (y)
      (memberᵒ y l))))
```

Given that its argument is a list, how does *first-value* differ from *car*

95 If *l* is the empty list or not a list, (*first-value* *l*) returns **()**, whereas with *car* there is no meaning. Also, instead of returning the first value, it returns the list of the first value.

What is the value of

(*first-value* (pasta e fagioli))

96 (pasta).

What value is associated with *y* in

(*first-value* (pasta e fagioli))

97 pasta.

Consider this variant of *memberᵒ*.

```
(define memberrevᵒ
  (lambda (x l)
    (condᵉ
      ((nullᵒ l) #u)
      (#s
        (fresh (d)
          (cdrᵒ l d)
          (memberrevᵒ x d)))
      (else (eq-carᵒ l x)))))
```

How does it differ from the definition of *memberᵒ* in frame 54?

98 We have *swapped* the second **cond**^{e} line with the third **cond**^{e} line[†].

[†] Clearly, **#s** corresponds to **else**. The (*eq-carᵒ* *l* *x*) is now the last question, so we can insert an **else** to improve clarity. We haven't swapped the expressions in the second **cond**^{e} line of *memberrevᵒ*, but we could have, since we can add or remove **#s** from a **cond**^{e} line without affecting the line.

How can we simplify this definition?

99 By removing a **cond**^{e} line that is guaranteed to fail.

What is the value of

```
(run* (x)
  (memberrevᵒ x (pasta e fagioli)))
```

100 (fagioli e pasta).

Define *reverse-list*, which reverses a list, using the definition of *memberrev*^{o}.

101 Here it is.

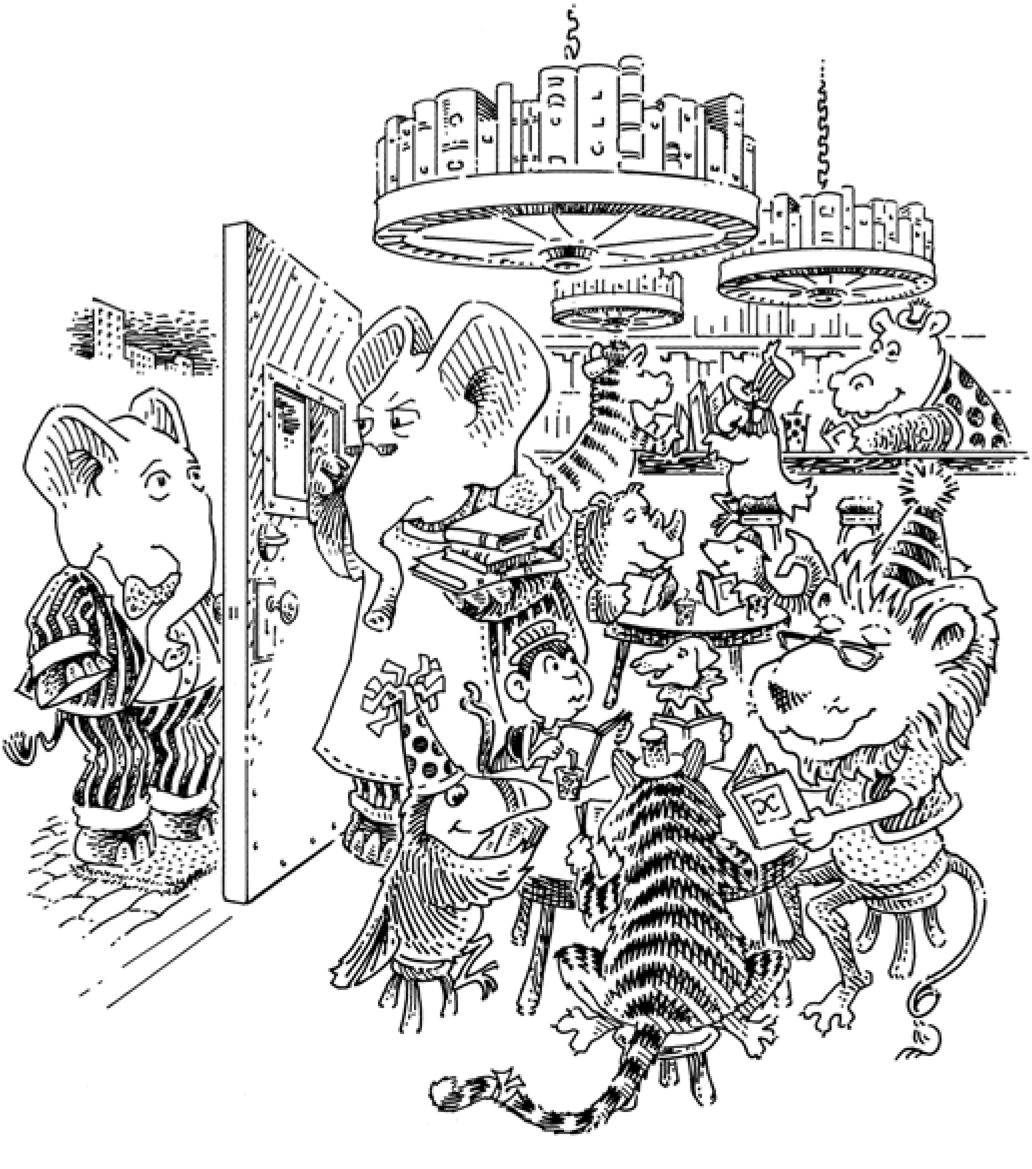
```
(define reverse-list
  (lambda (l)
    (run* (y)
      (memberrevᵒ y l))))
```

⟹ **Now go make yourself a peanut butter and marmalade sandwich.** ⟸

This space reserved for

MARMALADE STAINS!

4.
Members Only



Consider this very simple function.

```
(define mem
  (lambda (x l)
    (cond
      ((null? l) #f)
      ((eq-car? l x) l)
      (else (mem x (cdr l))))))
```

What is the value of

(*mem* tofu (a b tofu d peas e))

1 (tofu d peas e).

What is the value of

(*mem* tofu (a b peas d peas e))

2 #f.

What value is associated with *out* in

(**run*** (*out*)
  (≡ (*mem* tofu (a b tofu d peas e)) *out*))

3 (tofu d peas e).

What is the value of

(*mem* peas
  (*mem* tofu (a b tofu d peas e)))

4 (peas e).

What is the value of

(*mem* tofu
  (*mem* tofu (a b tofu d tofu e)))

5 (tofu d tofu e),
because the value of
(*mem* tofu (a b tofu d tofu e)) is
(tofu d tofu e), and because the value of
(*mem* tofu (tofu d tofu e)) is
(tofu d tofu e).

What is the value of

(*mem* tofu
  (*cdr* (*mem* tofu (a b tofu d tofu e))))

6 (tofu e),
because the value of
(*mem* tofu (a b tofu d tofu e)) is
(tofu d tofu e), the value of
(*cdr* (tofu d tofu e)) is (d tofu e), and the
value of (*mem* tofu (d tofu e)) is (tofu e).

Here is *mem*^{o}.

```
(define memᵒ
  (lambda (x l out)
    (condᵉ
      ((nullᵒ l) #u)
      ((eq-carᵒ l x) (≡ l out))
      (else
        (fresh (d)
          (cdrᵒ l d)
          (memᵒ x d out))))))
```

How does *mem*^{o} differ from *list*^{o}, *lol*^{o}, and *member*^{o}

7 The *list?*, *lol?*, and *member?* definitions from the previous chapter have only Booleans as their values, but *mem*, on the other hand, does not. Because of this we need an additional variable, which here we call *out*, that holds *mem*^{o}'s value.

Which expression has been unnested?

8 (*mem* *x* (*cdr* *l*)).

The Second Commandment

To transform a function whose value is not a Boolean into a function whose value is a goal, add an extra argument to hold its value, replace cond with cond^e, and unnest each question and answer.

In a call to *mem*^{o} from **run**^{1}, how many times does *out* get an association?

9 At most once.

What is the value of

```
(run¹ (out)
  (memᵒ tofu (a b tofu d tofu e) out))
```

10 ((tofu d tofu e)).

What is the value of

```
(run¹ (out)
  (fresh (x)
    (memᵒ tofu (a b x d tofu e) out)))
```

11 ((tofu d tofu e)), which would be correct if *x* were tofu.

What value is associated with r in

```
(run* (r)
  (memo r
    (a b tofu d tofu e)
    (tofu d tofu e)))
```

12 tofu.

What value is associated with q in

```
(run* (q)
  (memo tofu (tofu e) (tofu e))
  (≡ #t q))
```

13 #t,
since (tofu e), the last argument to mem^o, is the right value.

What is the value of

```
(run* (q)
  (memo tofu (tofu e) (tofu))
  (≡ #t q))
```

14 (),
since (tofu), the last argument to mem^o, is the wrong value.

What value is associated with x in

```
(run* (x)
  (memo tofu (tofu e) (x e)))
```

15 tofu,
when the value associated with x is tofu, then (x e) is (tofu e).

What is the value of

```
(run* (x)
  (memo tofu (tofu e) (peas x)))
```

16 (),
because there is no value that, when associated with x, makes (peas x) be (tofu e).

What is the value of

```
(run* (out)
  (fresh (x)
    (memo tofu (a b x d tofu e) out)))
```

17 ((tofu d tofu e) (tofu e)).

What is the value of

```
(run12 (z)
  (fresh (u)
    (memo tofu (a b tofu d tofu e . z) u)))
```

18

```
(_.0
 _.0
 (tofu . _.0)
 (_.0 tofu . _.1)
 (_.0 _.1 tofu . _.2)
 (_.0 _.1 _.2 tofu . _.3)
 (_.0 _.1 _.2 _.3 tofu . _.4)
 (_.0 _.1 _.2 _.3 _.4 tofu . _.5)
 (_.0 _.1 _.2 _.3 _.4 _.5 tofu . _.6)
 (_.0 _.1 _.2 _.3 _.4 _.5 _.6 tofu . _.7)
 (_.0 _.1 _.2 _.3 _.4 _.5 _.6 _.7 tofu . _.8)
 (_.0 _.1 _.2 _.3 _.4 _.5 _.6 _.7 _.8 tofu . _.9)).
```

How do we get the first two $\_{}_0$'s?

19 The first $\_{}_0$ corresponds to finding the first tofu. The second $\_{}_0$ corresponds to finding the second tofu.

Where do the other ten lists come from?

20 In order for

```
(memo tofu (a b tofu d tofu e . z) u)
```

to succeed, there must be a tofu in *z*. So *mem*^{o} creates all the possible lists with tofu as one element of the list. That's very interesting!

How can *mem*^{o} be simplified?

21 The first **cond**^{e} line always fails, so it can be removed.

```
(define memo
  (lambda (x l out)
    (conde
      ((eq-caro l x) (≡ l out))
      (else
        (fresh (d)
          (cdro l d)
          (memo x d out))))))
```

Remember *rember*.

```
(define rember
  (lambda (x l)
    (cond
      ((null? l) ())
      ((eq-car? l x) (cdr l))
      (else
        (cons (car l)
          (rember x (cdr l)))))))
```

22 Of course, it's an old friend.

What is the value of

(*rember* peas (a b peas d peas e))

23 (a b d peas e).

Consider *rember*^{o}.

```
(define rembero
  (lambda (x l out)
    (conde
      ((nullo l) (≡ () out))
      ((eq-caro l x) (cdro l out))
      (else
        (fresh (res)
          (fresh (d)
            (cdro l d)
            (rembero x d res))
          (fresh (a)
            (caro l a)
            (conso a res out)))))))
```

Is *rember*^{o} recursive?

24 Yes, just like *rember*.

Why are there three **fresh**es in

```
(fresh (res)
  (fresh (d)
    (cdro l d)
    (rembero x d res))
  (fresh (a)
    (caro l a)
    (conso a res out)))
```

25 Because d is only mentioned in (*cdr*^{o} *l d*) and (*rember*^{o} *x d res*); a is only mentioned in (*car*^{o} *l a*) and (*cons*^{o} *a res out*); but *res* is mentioned throughout.

Rewrite

(**fresh** (*res*)
  (**fresh** (*d*)
    (*cdr*^{o} *l* *d*)
    (*rember*^{o} *x* *d* *res*))
  (**fresh** (*a*)
    (*car*^{o} *l* *a*)
    (*cons*^{o} *a* *res* *out*)))

using only one **fresh**.

26 (**fresh** (*a* *d* *res*)
  (*cdr*^{o} *l* *d*)
  (*rember*^{o} *x* *d* *res*)
  (*car*^{o} *l* *a*)
  (*cons*^{o} *a* *res* *out*)).

How might we use *cons*^{o} in place of the *car*^{o} and the *cdr*^{o}

27 (**fresh** (*a* *d* *res*)
  (*cons*^{o} *a* *d* *l*)
  (*rember*^{o} *x* *d* *res*)
  (*cons*^{o} *a* *res* *out*)).

How does the first *cons*^{o} differ from the second one?

28 The first *cons*^{o}, (*cons*^{o} *a* *d* *l*), *appears* to associate values with the variables *a* and *d*. In other words, it appears to take apart a *cons* pair, whereas (*cons*^{o} *a* *res* *out*) *appears* to be used to build a *cons* pair.

But, can appearances be deceiving?

29 Indeed they can.

What is the value of

(**run**$^\mathbf{1}$ (*out*)
  (**fresh** (*y*)
    (*rember*^{o} peas (a b *y* d peas e) *out*)))

30 ((a b d peas e)),
because *y* is a variable and can take on values. The *car*^{o} within the (*eq-car*^{o} *l* *x*) associates *y* with peas, forcing *y* to be removed from the list. Of course we can associate with *y* a value other than peas. That will still cause (*rember*^{o} peas (a b *y* d peas e) *out*) to succeed, but **run**$^\mathbf{1}$ produces only one value.

What is the value of

(**run*** (*out*)
  (**fresh** (*y* *z*)
    (*rember*^{o} *y* (a b *y* d *z* e) *out*)))

31 ((b a d $\_{0}$ e)
 (a b d $\_{0}$ e)
 (a b d $\_{0}$ e)
 (a b d $\_{0}$ e)
 (a b $\_{0}$ d e)
 (a b e d $\_{0}$)
 (a b $\_{0}$ d $\_{1}$ e)).

Why is

(b a d $\_{0}$ e)

the first value?

32 It looks like b and a have been swapped, and *y* has disappeared.

No. Why does b come first?

33 The b comes first because the a has been removed.

Why does the list still contain a

34 In order to remove the a, *y* gets associated with a. The *y* in the list is then replaced with its value.

Why is

(a b d $\_{0}$ e)

the second value?

35 It looks like *y* has disappeared.

No. Has the b in the original list been removed?

36 Yes.

Why does the list still contain a b

37 In order to remove the b, *y* gets associated with b. The *y* in the list is then replaced with its value.

Why is

(a b d $\_{0}$ e)

the third value?

38 Is it for the same reason that (a b d $\_{0}$ e) is the second value?

Not quite. Has the b in the original list been removed?

39 No,
but the y has been removed.

Why is

(a b d $\_{0}$ e)

the fourth value?

40 Because the d has been removed from the list.

Why does the list still contain a d

41 In order to remove the d, y gets associated with d. Also the y in the list is replaced with its value.

Why is

(a b $\_{0}$ d e)

the fifth value?

42 Because the z has been removed from the list.

Why does the list contain $\_{0}$

43 When $(car\ l)$ is y, $(car^o\ l\ a)$ associates the fresh variable y with the fresh variable a. In order to remove the y, y gets associated with z. Since z is also a fresh variable, the a, y, and z co-refer.

Why is

(a b e d $\_{0}$)

the sixth value?

44 Because the e has been removed from the list.

Why does the list contain $\_{0}$

45 When $(car\ l)$ is z, $(car^o\ l\ a)$ associates the fresh variable z with the fresh variable a.

Why don't z and y co-refer?

46 Because we are within a **run***, we get to pretend that $(eq\text{-}car^o\ l\ x)$ fails when $(car\ l)$ is z and x is y. Thus z and y no longer co-refer.

Why is (a b $\_\_{0}$ d $\_\_{1}$ e) the seventh value?	47	Because we have not removed anything from the list.
Why does the list contain $\_\_{0}$ and $\_\_{1}$	48	When (*car* *l*) is *y*, (*car*^{o} *l* *a*) associates the fresh variable *y* with the fresh variable *a*. When (*car* *l*) is *z*, (*car*^{o} *l* *a*) associates the fresh variable *z* with a *new* fresh variable *a*. Also the *y* and *z* in the list are replaced respectively with their reified values.

What is the value of
(**run*** (*r*)
  (**fresh** (*y* *z*)
    (*rember*^{o} *y* (*y* d *z* e) (*y* d e))
    ($\equiv$ (*y* *z*) *r*)))
	49	((d d)
		 (d d)
		 ($\_\_{0}$ $\_\_{0}$)
		 (e e)).

Why is (d d) the first value?	50	When *y* is d and *z* is d, then (*rember*^{o} d (d d d e) (d d e)) succeeds.
Why is (d d) the second value?	51	When *y* is d and *z* is d, then (*rember*^{o} d (d d d e) (d d e)) succeeds.
Why is ($\_\_{0}$ $\_\_{0}$) the third value?	52	As long as *y* and *z* are the same, *y* can be anything.
How is (d d) the first value?	53	*rember*^{o} removes *y* from the list (*y* d *z* e), yielding the list (d *z* e); (d *z* e) is the same as *out*, (*y* d e), only when both *y* and *z* are the value d.

How is

(d d)

the second value?

54 Next, *rember*^{o} removes d from the list (*y* d *z* e), yielding the list (*y* *z* e); (*y* *z* e) is the same as *out*, (*y* d e), only when *z* is d. Also, in order to remove the d, *y* gets associated with d.

How is

($\_0$ $\_0$)

the third value?

55 Next, *rember*^{o} removes *z* from the list (*y* d *z* e), yielding the list (*y* d e); (*y* d e) is always the same as *out*, (*y* d e). Also, in order to remove the *z*, *y* gets associated with *z*, so they co-refer.

How is

(e e)

the fourth value?

56 Next, *rember*^{o} removes e from the list (*y* d *z* e), yielding the list (*y* d *z*); (*y* d *z*) is the same as *out*, (*y* d e), only when *z* is e. Also, in order to remove the e, *y* gets associated with e.

What is the value of

(**run**^{13} (*w*)
  (**fresh** (*y* *z* *out*)
    (*rember*^{o} *y* (a b *y* d *z* . *w*) *out*)))

57 ($\_0$
$\_0$
$\_0$
$\_0$
$\_0$
()
($\_0$. $\_1$)
($\_0$)
($\_0$ $\_1$. $\_2$)
($\_0$ $\_1$)
($\_0$ $\_1$ $\_2$. $\_3$)
($\_0$ $\_1$ $\_2$)
($\_0$ $\_1$ $\_2$ $\_3$. $\_4$)).

Why is

$\_0$

the first value?

58 When *y* is a, *out* becomes (b *y* d *z* . *w*), which makes

(*rember*^{o} *y* (a b *y* d *z* . *w*) (b *y* d *z* . *w*))

succeed for all values of *w*.

How is

$\_{0}$

the first value?

59 *rember^o* removes a from *l*, while ignoring the fresh variable *w*.

How is

$\_{0}$

the second, third, and fourth value?

60 This is the same as in the previous frame, except that *rember^o* removes b from the original *l*, *y* from the original *l*, and d from the original *l*, respectively.

How is

$\_{0}$

the fifth value?

61 Next, *rember^o* removes *z* from *l*. When the (*eq-car^o l x*) question of the second **cond**^{e} line succeeds, (*car l*) is *z*. The answer of the second **cond**^{e} line, (*cdr^o l out*), also succeeds, associating the *cdr* of *l* (the fresh variable *w*) with the fresh variable *out*. The variable *out*, however, is just *res*, the fresh variable passed into the recursive call to *rember^o*.

How is

()

the sixth value?

62 Because none of the first five values in *l* are removed. The (*null^o l*) question of the first **cond**^{e} line then succeeds, associating *w* with the empty list.

How is

$(\_{0} \;.\; \_{1})$

the seventh value?

63 Because none of the first five values in *l* are removed, and because we pretend that the (*null^o l*) question of the first **cond**^{e} line fails. The (*eq-car^o l x*) question of the second **cond**^{e} line succeeds, however, and associates *w* with a pair whose *car* is *y*. The answer (*cdr^o l out*) of the second **cond**^{e} line also succeeds, associating *w* with a pair whose *cdr* is *out*. The variable *out*, however, is just *res*, the fresh variable passed into the recursive call to *rember^o*. During the recursion, the *car^o* inside the second **cond**^{e} line's *eq-car^o* associates the fresh variable *y* with the fresh variable *a*.

How is

($\_{0}$)

the eighth value?

64 This is the same as the seventh value, ($\_{0}$. $\_{1}$), except that the (*null*^{o} *l*) question of the first **cond**^{e} line succeeds, associating *out* (and, therefore, *res*) with the empty list.

How is

($\_{0}$ $\_{1}$. $\_{2}$)

the ninth value?

65 For the same reason that ($\_{0}$. $\_{1}$) is the seventh value, except that the ninth value performs an additional recursive call, which results in an additional *cons*^{o}.

Do the tenth and twelfth values correspond to the eighth value?

66 Yes.

Do the eleventh and thirteenth values correspond to the ninth value?

67 Yes.

All *w* of the form

($\_{0}$ $\cdots$ $\_{n}$. $\_{n+1}$)

make (*rember*^{o} *y* (a b *y* d *z* . *w*) *out*) succeed.

Here is *surprise*^{o}.

```
(define surpriseᵒ
  (lambda (s)
    (memberᵒ s (a b c) (a b c))))
```

Are there any values of *s* for which (*surprise*^{o} *s*) should succeed?

68 Yes, (*surprise*^{o} *s*) should succeed for all values of *s* other than a, b, and c.

What value is associated with *r* in

```
(run* (r)
  (≡ d r)
  (surpriseᵒ r))
```

69 d.

What is the value of

```
(run* (r)
  (surpriseᵒ r))
```

70 ($\_{0}$).

When *r* is fresh, (*surprise*^{o} *r*) succeeds and leaves *r* fresh.

Write an expression that shows why this definition of *surprise*^{o} should not succeed when r is fresh.

71 Here is such an expression:

(**run*** (r)
  (*surprise*^{o} r)
  ($\equiv$ b r)).

If (*surprise*^{o} r) were to leave r fresh, then ($\equiv$ b r) would associate r with b. But if r were b, then (*rember*^{o} r (a b c) (a b c)) should have failed, since removing b from the list (a b c) results in (a c), not (a b c).

And what is the value of

(**run*** (r)
  ($\equiv$ b r)
  (*surprise*^{o} r))

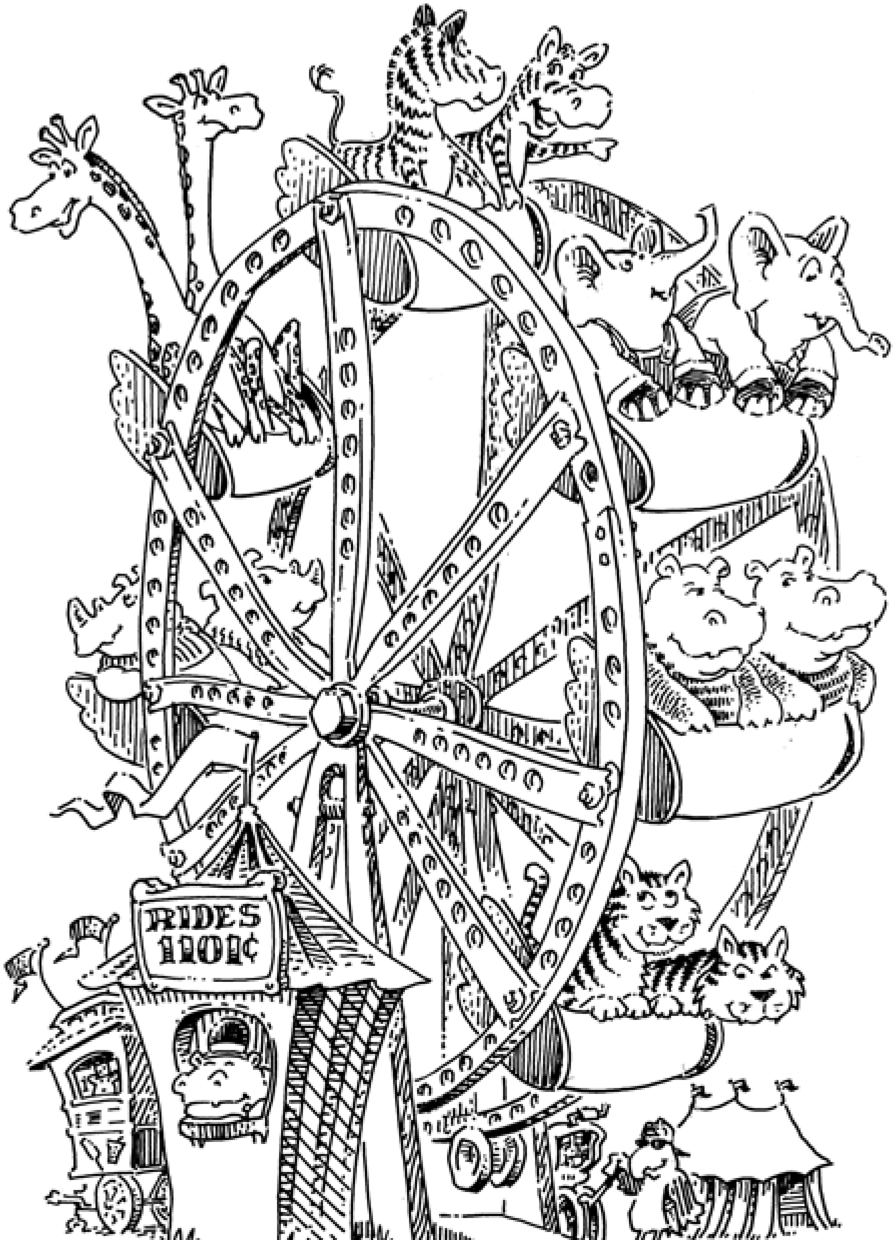
72 (b),
which also makes no sense. Please pass the aspirin!

⟹ Now go munch on some carrots. ⟸

This space reserved for

CARROT STAINS!

5.
Double Your Fun



Ever seen *append* | 1 No.

Here it is.†

```
(define append
  (lambda (l s)
    (cond
      ((null? l) s)
      (else (cons (car l)
              (append (cdr l) s))))))
```

What is the value of
(*append* (a b c) (d e))

2 (a b c d e).

† For a different approach to *append*, see William F. Clocksin. *Clause and Effect*. Springer, 1997, page 59.

What is the value of
(*append* (a b c) ())

3 (a b c).

What is the value of
(*append* () (d e))

4 (d e).

What is the value of
(*append* a (d e))

5 It has no meaning,
because a is neither the empty list nor a proper list.

What is the value of
(*append* (d e) a)

6 It has no meaning, again?

No. The value is (d e . a).

7 How is that possible?

Look closely at the definition of *append*; there are no questions asked about s.

8 Ouch.

Define $append^o$.

9

```
(define appendᵒ
  (lambda (l s out)
    (condᵉ
      ((nullᵒ l) (≡ s out))
      (else
        (fresh (a d res)
          (carᵒ l a)
          (cdrᵒ l d)
          (appendᵒ d s res)
          (consᵒ a res out))))))
```

What value is associated with x in

```
(run* (x)
  (appendᵒ
    (cake)
    (tastes yummy)
    x))
```

10 (cake tastes yummy).

What value is associated with x in

```
(run* (x)
  (fresh (y)
    (appendᵒ
      (cake with ice y)
      (tastes yummy)
      x)))
```

11 (cake with ice $_{-0}$ tastes yummy).

What value is associated with x in

```
(run* (x)
  (fresh (y)
    (appendᵒ
      (cake with ice cream)
      y
      x)))
```

12 (cake with ice cream . $_{-0}$).

What value is associated with x in

```
(run¹ (x)
  (fresh (y)
    (appendᵒ (cake with ice . y) (d t) x)))
```

13 (cake with ice d t),
because the last call to $null^o$ associates y with the empty list.

How can we show that y is associated with the empty list?

14 By this example

```
(run¹ (y)
  (fresh (x)
    (appendᵒ (cake with ice . y) (d t) x)))
```

which associates y with the empty list.

Redefine $append^o$ to use a single $cons^o$ in place of the car^o and cdr^o (see 4:27).

15

```
(define appendᵒ
  (lambda (l s out)
    (condᵉ
      ((nullᵒ l) (≡ s out))
      (else
        (fresh (a d res)
          (consᵒ a d l)
          (appendᵒ d s res)
          (consᵒ a res out))))))
```

What is the value of

```
(run⁵ (x)
  (fresh (y)
    (appendᵒ (cake with ice . y) (d t) x)))
```

16

```
((cake with ice d t)
 (cake with ice _₀ d t)
 (cake with ice _₀ _₁ d t)
 (cake with ice _₀ _₁ _₂ d t)
 (cake with ice _₀ _₁ _₂ _₃ d t)).
```

What is the value of

```
(run⁵ (y)
  (fresh (x)
    (appendᵒ (cake with ice . y) (d t) x)))
```

17

```
(()
 (_₀)
 (_₀ _₁)
 (_₀ _₁ _₂)
 (_₀ _₁ _₂ _₃)).
```

Let's consider plugging in ($\_{0}$ $\_{1}$ $\_{2}$) for *y* in

(cake with ice . *y*).

Then we get

(cake with ice . ($\_{0}$ $\_{1}$ $\_{2}$)).

What list is this the same as?

18 (cake with ice $\_{0}$ $\_{1}$ $\_{2}$).

Right. What is

(*append* (cake with ice $\_{0}$ $\_{1}$ $\_{2}$) (d t))

19 The fourth list in frame 16.

What is the value of

```
(run5 (x)
  (fresh (y)
    (appendo
      (cake with ice . y)
      (d t . y)
      x)))
```

20
```
((cake with ice d t)
 (cake with ice _0 d t _0)
 (cake with ice _0 _1 d t _0 _1)
 (cake with ice _0 _1 _2 d t _0 _1 _2)
 (cake with ice _0 _1 _2 _3 d t _0 _1 _2 _3)).
```

What is the value of

```
(run* (x)
  (fresh (z)
    (appendo
      (cake with ice cream)
      (d t . z)
      x)))
```

21 ((cake with ice cream d t . $\_{0}$)).

Why does the list contain only one value?

22 Because *z* stays fresh.

Let's try an example in which the first two arguments are variables. What is the value of

```
(run6 (x)
  (fresh (y)
    (appendo x y (cake with ice d t))))
```

23
```
(()
 (cake)
 (cake with)
 (cake with ice)
 (cake with ice d)
 (cake with ice d t)).
```

How might we describe these values?

24 The values include all of the prefixes of the list (cake with ice d t).

Now let's try this variation.

```
(run⁶ (y)
  (fresh (x)
    (appendᵒ x y (cake with ice d t))))
```

What is its value?

25
```
((cake with ice d t)
 (with ice d t)
 (ice d t)
 (d t)
 (t)
 ()).
```

How might we describe these values?

26 The values include all of the suffixes of the list (cake with ice d t).

Let's combine the previous two results. What is the value of

```
(run⁶ (r)
  (fresh (x y)
    (appendᵒ x y (cake with ice d t))
    (≡ (x y) r)))
```

27
```
((() (cake with ice d t))
 ((cake) (with ice d t))
 ((cake with) (ice d t))
 ((cake with ice) (d t))
 ((cake with ice d) (t))
 ((cake with ice d t) ())).
```

How might we describe these values?

28 Each value includes two lists that, when appended together, form the list

(cake with ice d t).

What is the value of

```
(run⁷ (r)
  (fresh (x y)
    (appendᵒ x y (cake with ice d t))
    (≡ (x y) r)))
```

29 It has no value,
since it is still looking for the seventh value.

Should its value be the same as if we asked for only six values?

30 Yes, that would make sense.

31 How can we change the definition of $append^o$ so that is indeed what happens?

Swap the last two goals of $append^o$.

```
(define appendᵒ
  (lambda (l s out)
    (condᵉ
      ((nullᵒ l) (≡ s out))
      (else
        (fresh (a d res)
          (consᵒ a d l)
          (consᵒ a res out)
          (appendᵒ d s res))))))
```

---

32 Now, using this revised definition of $append^o$, what is the value of

```
(run⁷ (r)
  (fresh (x y)
    (appendᵒ x y (cake with ice d t))
    (≡ (x y) r)))
```

The value is in frame 27.

---

33 What is the value of

```
(run⁷ (x)
  (fresh (y z)
    (appendᵒ x y z)))
```

```
(()
 (_₀)
 (_₀ _₁)
 (_₀ _₁ _₂)
 (_₀ _₁ _₂ _₃)
 (_₀ _₁ _₂ _₃ _₄)
 (_₀ _₁ _₂ _₃ _₄ _₅)).
```

---

34 What is the value of

```
(run⁷ (y)
  (fresh (x z)
    (appendᵒ x y z)))
```

```
(_₀
 _₀
 _₀
 _₀
 _₀
 _₀
 _₀).
```

---

35 It should be obvious how we get the first value. Where do the last four values come from?

A new fresh variable *res* is passed into each recursive call to $append^o$. After ($null^o$ l) succeeds, *res* is associated with s, which is the fresh variable z.

What is the value of

```
(run 7 (z)
  (fresh (x y)
    (appendo x y z)))
```

36
```
(_0
 (_0 . _1)
 (_0 _1 . _2)
 (_0 _1 _2 . _3)
 (_0 _1 _2 _3 . _4)
 (_0 _1 _2 _3 _4 . _5)
 (_0 _1 _2 _3 _4 _5 . _6)).
```

Let's combine the previous three results. What is the value of

```
(run 7 (r)
  (fresh (x y z)
    (appendo x y z)
    (≡ (x y z) r)))
```

37
```
((() _0 _0)
 ((_0) _1 (_0 . _1))
 ((_0 _1) _2 (_0 _1 . _2))
 ((_0 _1 _2) _3 (_0 _1 _2 . _3))
 ((_0 _1 _2 _3) _4 (_0 _1 _2 _3 . _4))
 ((_0 _1 _2 _3 _4) _5 (_0 _1 _2 _3 _4 . _5))
 ((_0 _1 _2 _3 _4 _5) _6 (_0 _1 _2 _3 _4 _5 . _6))).
```

Define *swappend*^{o}, which is just *append*^{o} with its two **cond**^{e} lines swapped.

38 That's a snap.

```
(define swappendo
  (lambda (l s out)
    (conde
      (#s
        (fresh (a d res)
          (conso a d l)
          (conso a res out)
          (swappendo d s res)))
      (else (nullo l) (≡ s out)))))
```

What is the value of

```
(run 1 (z)
  (fresh (x y)
    (swappendo x y z)))
```

39 It has no value.

Why does

```
(run^1 (z)
  (fresh (x y)
    (swappend^o x y z)))
```

have no value?†

40 In ($swappend^o$ d s res) the variables d, s, and res remain fresh, which is where we started.

† We can redefine $swappend^o$ so that this **run** expression has a value.

```
(define swappend^o
  (lambda-limited 5 (l s out)
    (cond^e
      (#s
        (fresh (a d res)
          (cons^o a d l)
          (cons^o a res out)
          (swappend^o d s res)))
      (else (null^o l) (≡ s out)))))
```

Where **lambda-limited** is defined on the right.

Here is **lambda-limited** with its auxiliary function *ll*.

```
(define-syntax lambda-limited
  (syntax-rules ()
    ((_ n formals g)
     (let ((x (var x)))
       (lambda formals
         (ll n x g))))))

(define ll
  (lambda (n x g)
    (λ_G (s)
      (let ((v (walk x s)))
        (cond
          ((var? v) (g (ext-s x 1 s)))
          ((< v n) (g (ext-s x (+ v 1) s)))
          (else (#u s)))))))
```

The functions *var*, *walk*, and *ext-s* are described in 9:6, 9:27, and 9:29, respectively. λ_G (see appendix) is just **lambda**.

Consider this definition.

```
(define unwrap
  (lambda (x)
    (cond
      ((pair? x) (unwrap (car x)))
      (else x))))
```

What is the value of

(*unwrap* ((((pizza)))))

41 pizza.

What is the value of

(*unwrap* ((((pizza pie) with)) extra cheese))

42 pizza.

This might be a good time for a pizza break.

43 Good idea.

Back so soon? Hope you are not too full.

44 Not too.

Define *unwrap*^{o}.

45 That's a slice of pizza!

```
(define unwrapᵒ
  (lambda (x out)
    (condᵉ
      ((pairᵒ x)
       (fresh (a)
         (carᵒ x a)
         (unwrapᵒ a out)))
      (else (≡ x out)))))
```

What is the value of

```
(run* (x)
  (unwrapᵒ (((pizza))) x))
```

46 (pizza
(pizza)
((pizza))
(((pizza)))).

The first value of the list seems right. In what way are the other values correct?

47 They represent partially wrapped versions of the list (((pizza))). And the last value is the fully-wrapped original list (((pizza))).

What is the value of

```
(run¹ (x)
  (unwrapᵒ x pizza))
```

48 It has no value.

What is the value of

```
(run¹ (x)
  (unwrapᵒ ((x)) pizza))
```

49 It has no value.

Why doesn't

```
(run¹ (x)
  (unwrapᵒ ((x)) pizza))
```

have a value?

50 The recursion happens too early. Therefore the (≡ *x* *out*) goal is not reached.

What can we do about that?

51 Introduce a revised definition of *unwrap*^{o}?

52 Yes. Let's swap the two **cond**^{e} lines as in 3:98.

Like this.

```
(define unwrapᵒ
  (lambda (x out)
    (condᵉ
      (#s (≡ x out))
      (else
        (fresh (a)
          (carᵒ x a)
          (unwrapᵒ a out))))))
```

53 What is the value of

```
(run⁵ (x)
  (unwrapᵒ x pizza))
```

```
(pizza
 (pizza . _0)
 ((pizza . _0) . _1)
 (((pizza . _0) . _1) . _2)
 ((((pizza . _0) . _1) . _2) . _3)).
```

54 What is the value of

```
(run⁵ (x)
  (unwrapᵒ x ((pizza))))
```

```
(((pizza))
 (((pizza)) . _0)
 ((((pizza)) . _0) . _1)
 (((((pizza)) . _0) . _1) . _2)
 ((((((pizza)) . _0) . _1) . _2) . _3)).
```

55 What is the value of

```
(run⁵ (x)
  (unwrapᵒ ((x)) pizza))
```

```
(pizza
 (pizza . _0)
 ((pizza . _0) . _1)
 (((pizza . _0) . _1) . _2)
 ((((pizza . _0) . _1) . _2) . _3)).
```

56 If you haven't taken a pizza break yet, stop and take one now! We're taking an ice cream break.

Okay, okay!

57 Did you enjoy the pizza as much as we enjoyed the ice cream?

Indubitably!

Consider this definition.

```
(define flatten
  (lambda (s)
    (cond
      ((null? s) ())
      ((pair? s)
       (append
         (flatten (car s))
         (flatten (cdr s))))
      (else (cons s ())))))
```

What is the value of

(*flatten* ((a b) c))

58 (a b c).

Define *flatten*^{o}.

59 Here it is.

```
(define flattenᵒ
  (lambda (s out)
    (condᵉ
      ((nullᵒ s) (≡ () out))
      ((pairᵒ s)
       (fresh (a d res-a res-d)
         (consᵒ a d s)†
         (flattenᵒ a res-a)
         (flattenᵒ d res-d)
         (appendᵒ res-a res-d out)))
      (else (consᵒ s () out)))))
```

† See 4:27.

What value is associated with x in

(**run¹** (x)
  (*flatten*^{o} ((a b) c) x))

60 (a b c).
No surprises here.

What value is associated with x in

(**run¹** (x)
  (*flatten*^{o} (a (b c)) x))

61 (a b c).

What is the value of

(**run*** (x)
  (*flatten*^{o} (a) x))

[62] ((a)
 (a ())
 ((a))).
 Here is a surprise!

The value in the previous frame contains three lists. Which of the lists, if any, are the same?

[63] None of the lists are the same.

What is the value of

(**run*** (x)
  (*flatten*^{o} ((a)) x))

[64] ((a)
 (a ())
 (a ())
 (a () ())
 ((a))
 ((a) ())
 (((a)))).

The value in the previous frame contains seven lists. Which of the lists, if any, are the same?

[65] The second and third lists are the same.

What is the value of

(**run*** (x)
  (*flatten*^{o} (((a))) x))

[66] ((a)
 (a ())
 (a ())
 (a () ())
 (a ())
 (a () ())
 (a () ())
 (a () () ())
 ((a))
 ((a) ())
 ((a) ())
 ((a) () ())
 (((a)))
 (((a)) ())
 ((((a))))).

The value in the previous frame contains fifteen lists. Which of the lists, if any, are the same?	67	The second, third, and fifth lists are the same; the fourth, sixth, and seventh lists are the same; and the tenth and eleventh lists are the same.
What is the value of (**run*** (x) (*flatten*^{o} ((a b) c) x))	68	((a b c) (a b c ()) (a b (c)) (a b () c) (a b () c ()) (a b () (c)) (a (b) c) (a (b) c ()) (a (b) (c)) ((a b) c) ((a b) c ()) ((a b) (c)) (((a b) c))).
The value in the previous frame contains thirteen lists. Which of the lists, if any, are the same?	69	None of the lists are the same.
Characterize that list of lists.	70	Each list flattens to (a b c). These are all the lists generated by attempting to flatten ((a b) c). Remember that a singleton list (a) is really the same as (a . ()), and with that additional perspective the pattern becomes clearer.
What is the value of (**run*** (x) (*flatten*^{o} x (a b c)))	71	It has no value.
What can we do about it?	72	Swap some of the **cond**^{e} lines?

Yes. Here is a variant of *flatten*^{o}.

```
(define flattenrevᵒ
  (lambda (s out)
    (condᵉ
      (#s (consᵒ s () out))
      ((nullᵒ s) (≡ () out))
      (else
        (fresh (a d res-a res-d)
          (consᵒ a d s)
          (flattenrevᵒ a res-a)
          (flattenrevᵒ d res-d)
          (appendᵒ res-a res-d out))))))
```

How does *flatten*^{o} differ from this variant?

73 The last **cond**^{e} line of *flatten*^{o} is the first **cond**^{e} line of this variant (see 3:98).

In *flatten*^{o} there is a (*pair*^{o} *s*) test. Why doesn't *flattenrev*^{o} have the same test?

74 Because (*cons*^{o} *a d s*) in the **fresh** expression guarantees that *s* is a pair. In other words, the (*pair*^{o} *s*) question is unnecessary in *flatten*^{o}.

What is the value of

```
(run* (x)
  (flattenrevᵒ ((a b) c) x))
```

75

```
((((a b) c))
 ((a b) (c))
 ((a b) c ())
 ((a b) c)
 (a (b) (c))
 (a (b) c ())
 (a (b) c)
 (a b () (c))
 (a b () c ())
 (a b () c)
 (a b (c))
 (a b c ())
 (a b c)).
```

What is the value of

```
(reverse
  (run* (x)
    (flattenrevᵒ ((a b) c) x)))
```

76 The value in frame 68.

What is the value of (**run²** (*x*) (*flattenrev*^{o} *x* (a b c)))	77	((a b . c) (a b c)).
Why is the value ((a b . c) (a b c))	78	Because (*flattenrev*^{o} (a b . c) (a b c)) and (*flattenrev*^{o} (a b c) (a b c)) both succeed.
What is the value of (**run³** (*x*) (*flattenrev*^{o} *x* (a b c)))	79	It has no value. In fact, it is still trying to determine the third value.

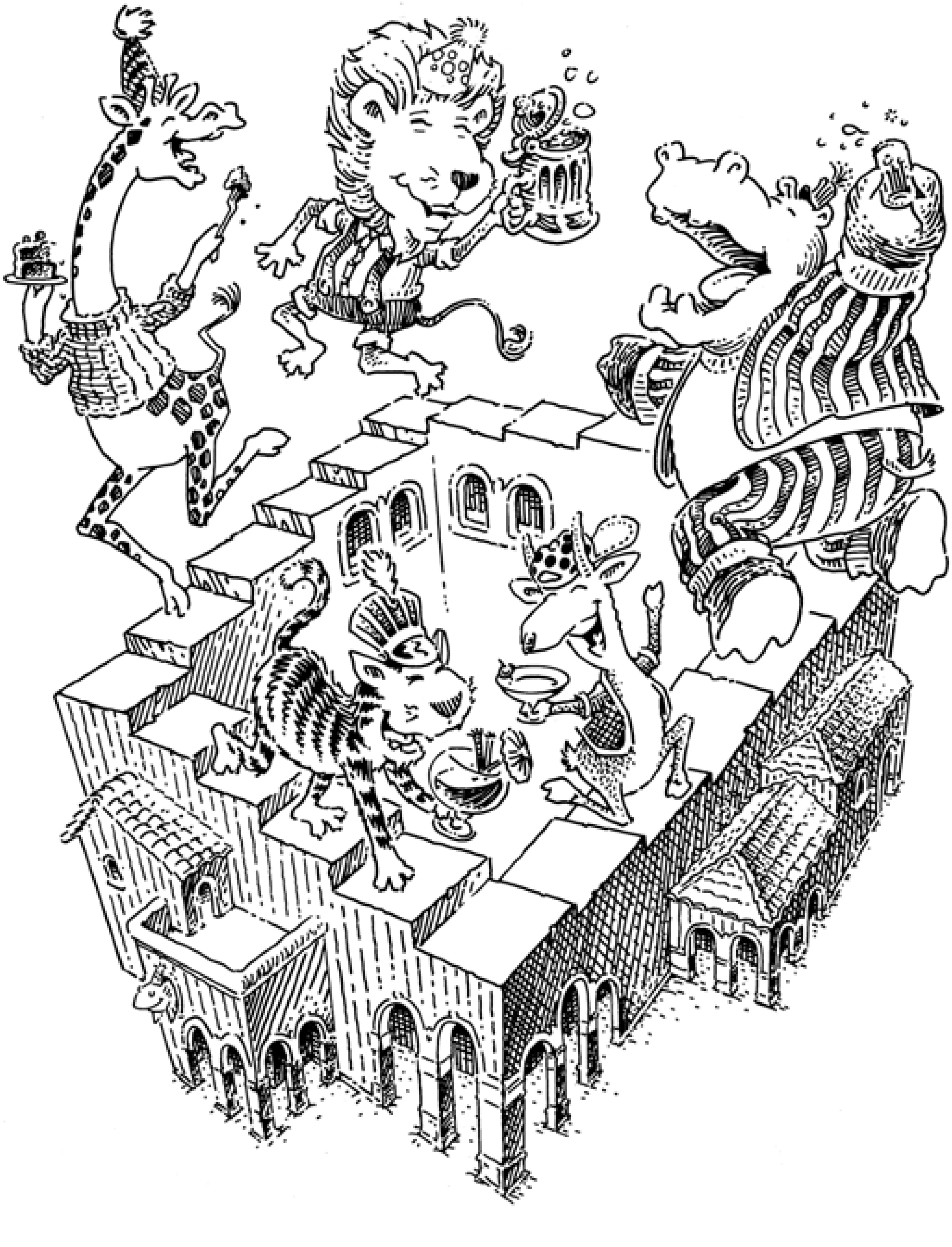
What is the value of
(*length*
  (**run*** (*x*)
    (*flattenrev*^{o} ((((a (((b))) c))) d) *x*)))
	80	574.
		Wow!

⟹ Now go make yourself a cashew butter and chutney sandwich. ⟸

This space reserved for

CHUTNEY STAINS!

6.
The Fun Never Ends...



[1] Here is an unusual definition.

```
(define anyᵒ
  (lambda (g)
    (condᵉ
      (g #s)
      (else (anyᵒ g)))))
```

Is it recursive?

Yes.

[2] Is there a base case?

Yes.

[3] Can *any*^{o} ever succeed?

Yes, if the goal g succeeds.

[4] Here is another definition.

```
(define neverᵒ (anyᵒ #u))
```

Can *never*^{o} ever succeed or fail?

No,
because although the question of the first **cond**^{e} line within *any*^{o} fails, the answer of the second **cond**^{e} line, (*any*^{o} #u), is where we started.

[5] What is the value of

```
(run¹ (q)
  neverᵒ
  (≡ #t q))
```

Of course, the **run**^{1} expression has no value.

[6] What is the value of

```
(run¹ (q)
  #u
  neverᵒ)
```

(),
because #u fails before *never*^{o} is reached.

[7] Here is a useful definition.

```
(define alwaysᵒ (anyᵒ #s))
```

What value is associated with q in

```
(run¹ (q)
  alwaysᵒ
  (≡ #t q))
```

#t.

Compare $always^o$ to #s.

8 $always^o$ always can succeed any number of times, whereas #s can succeed only once.

What is the value of

```
(run* (q)
  always°
  (≡ #t q))
```

9 It has no value,
since **run*** never finishes building the list
(#t #t #t ...

What is the value of

```
(run5 (q)
  always°
  (≡ #t q))
```

10 (#t #t #t #t #t).

And what is the value of

```
(run5 (q)
  (≡ #t q)
  always°)
```

11 It's the same: (#t #t #t #t #t).

Here is the definition of sal^o.†

```
(define salᵒ
  (lambda (g)
    (condᵉ
      (#s #s)
      (else g))))
```

Is sal^o recursive?

† sal^o stands for "succeeds at least once".

12 No.

What is the value of

```
(run1 (q)
  (salᵒ alwaysᵒ)
  (≡ #t q))
```

13 (#t),
because the first **cond**^{e} line of sal^o succeeds.

What is the value of

```
(run¹ (q)
  (salᵒ neverᵒ)
  (≡ #t q))
```

14 **(#t)**,
because the first **cond**^{e} line of *sal*^{o} succeeds.

What is the value of

```
(run* (q)
  (salᵒ neverᵒ)
  (≡ #t q))
```

15 It has no value,
because **run*** never finishes determining the *second* value.

What is the value of

```
(run¹ (q)
  (salᵒ neverᵒ)
  #u
  (≡ #t q))
```

16 It has no value,
because when the #u occurs, we pretend that the first **cond**^{e} line of *sal*^{o} fails, which causes **cond**^{e} to try *never*^{o}, which neither succeeds nor fails.

What is the value of

```
(run¹ (q)
  alwaysᵒ
  #u
  (≡ #t q))
```

17 It has no value,
because *always*^{o} succeeds, followed by #u, which causes *always*^{o} to be retried, which succeeds again, which leads to #u again, which causes *always*^{o} to be retried again, which succeeds again, which leads to #u, etc.

What is the value of

```
(run¹ (q)
  (condᵉ
    ((≡ #f q) alwaysᵒ)
    (else (anyᵒ (≡ #t q))))
  (≡ #t q))
```

18 It has no value.
First, #f gets associated with q, then *always*^{o} succeeds once. But in the outer (≡ #t q) we can't associate #t with q since q is already associated with #f. So the outer (≡ #t q) fails, then *always*^{o} succeeds again, and then (≡ #t q) fails again, etc.

What is the value of[†]

```
(run¹ (q)
  (condⁱ
    ((≡ #f q) alwaysᵒ)
    (else (≡ #t q)))
  (≡ #t q))
```

† **cond**^{i} is written `condi` and is pronounced "con-deye".

19 **(#t)**,
because after the first failure, instead of staying on the first line we try the second **cond**^{i} line.

What happens if we try for more values?

```
(run² (q)
  (condⁱ
    ((≡ #f q) alwaysᵒ)
    (else (≡ #t q)))
  (≡ #t q))
```

20 It has no value,
since the second **cond**^{i} line is out of values.

So does this give more values?

```
(run⁵ (q)
  (condⁱ
    ((≡ #f q) alwaysᵒ)
    (else (anyᵒ (≡ #t q))))
  (≡ #t q))
```

21 Yes, it yields as many as are requested,

(#t #t #t #t #t).

always^o succeeds five times, but contributes none of the five values, since then #f would be in the list.

Compare **cond**^{i} to **cond**^{e}.

22 **cond**^{i} looks and feels like **cond**^{e}. **cond**^{i} does not, however, wait until all the successful goals on a line are exhausted before it tries the next line.

Are there other differences?

23 Yes. A **cond**^{i} line that has additional values is not forgotten. That is why there is no value in frame 20.

The Law of $\mathbf{cond}^i$

$\mathbf{cond}^i$ behaves like $\mathbf{cond}^e$, except that its values are interleaved.

What is the value of

```
(run⁵ (r)
  (condⁱ
    ((teacupᵒ† r) #s)
    ((≡ #f r) #s)
    (else #u)))
```

† See 1:56.

24 (tea #f cup).

Let's be sure that we understand the difference between $\mathbf{cond}^e$ and $\mathbf{cond}^i$. What is the value of

```
(run⁵ (q)
  (condⁱ
    ((≡ #f q) alwaysᵒ)
    ((≡ #t q) alwaysᵒ)
    (else #u))
  (≡ #t q))
```

25 (#t #t #t #t #t).

And if we replace $\mathbf{cond}^i$ by $\mathbf{cond}^e$, do we get the same value?

26 No,
then the expression has no value.

Why does

```
(run⁵ (q)
  (condᵉ
    ((≡ #f q) alwaysᵒ)
    ((≡ #t q) alwaysᵒ)
    (else #u))
  (≡ #t q))
```

have no value?

27 It has no value,
because the first $\mathbf{cond}^e$ line succeeds, but the outer (≡ #t q) fails. This causes the first $\mathbf{cond}^e$ line to succeed again, etc.

What is the value of

(**run⁵** (q)
  (**cond**^{e}
    (*always*^{o} `#s`)
    (**else** *never*^{o}))
  (≡ `#t` q))

[28] It is (`#t #t #t #t #t`).

And if we replace **cond**^{e} by **cond**^{i}, do we get the same value?

[29] No.

And what about the value of

(**run⁵** (q)
  (**cond**^{i}
    (*always*^{o} `#s`)
    (**else** *never*^{o}))
  (≡ `#t` q))

[30] It has no value, because after the first **cond**^{i} line succeeds, rather than staying on the same **cond**^{i} line, it tries for more values on the second **cond**^{i} line, but that line is *never*^{o}.

What is the value of†

(**run¹** (q)
  (**all**
    (**cond**^{e}
      ((≡ `#f` q) `#s`)
      (**else** (≡ `#t` q)))
    always^o)
  (≡ `#t` q))

[31] It has no value. First, `#f` is associated with q. Then *always*^{o}, the second goal of the **all** expression, succeeds, so the entire **all** expression succeeds. Then (≡ `#t` q) tries to associate a value that is different from `#f` with q. This fails. So *always*^{o} succeeds again, and once again the second goal, (≡ `#t` q), fails. Since *always*^{o} always succeeds, there is no value.

† The goals of an **all** must succeed for the **all** to succeed.

Have a slice of Key lime pie.

Now, what is the value of[†]

```
(run¹ (q)
  (allⁱ
    (condᵉ
      ((≡ #f q) #s)
      (else (≡ #t q)))
    alwaysᵒ)
  (≡ #t q))
```

[†] **all**^{i} is written `alli` and is pronounced "all-eye".

32 (#t).
First, #f is associated with q. Then, *always*^{o} succeeds. Then the outer goal (≡ #t q) fails. This time, however, **all**^{i} moves on to the second **cond**^{e} line and associates #t with q. Then *always*^{o} succeeds, as does the outer (≡ #t q).

Now, what if we want more values?

```
(run⁵ (q)
  (allⁱ
    (condᵉ
      ((≡ #f q) #s)
      (else (≡ #t q)))
    alwaysᵒ)
  (≡ #t q))
```

33 (#t #t #t #t #t).
always^o succeeds ten times, with the value associated with q alternating between #f and #t.

What if we swap the two **cond**^{e} lines?

```
(run⁵ (q)
  (allⁱ
    (condᵉ
      ((≡ #t q) #s)
      (else (≡ #f q)))
    alwaysᵒ)
  (≡ #t q))
```

34 Its value is the same: (#t #t #t #t #t).

What does the "*i*" stand for in **cond**^{i} and **all**^{i}

35 It stands for *interleave*.

Let's be sure that we understand the difference between **all** and **all**^{i}. What is the value of

```
(run⁵ (q)
  (all
    (condᵉ
      (#s #s)
      (else neverᵒ))
    alwaysᵒ)
  (≡ #t q))
```

36 (#t #t #t #t #t).

And if we replace **all** by **all**^{i}, do we get the same value?

37 No,
it has no value.

Why does

```
(run⁵ (q)
  (allⁱ
    (condᵉ
      (#s #s)
      (else neverᵒ))
    alwaysᵒ)
  (≡ #t q))
```

have no value?

38 It has no value,
because the first **cond**^{e} line succeeds, and the outer (≡ #t q) succeeds. This yields one value, but when we go for a second value, we reach *never*^{o}.

Could **cond**^{i} have been used instead of **cond**^{e} in these last two examples?

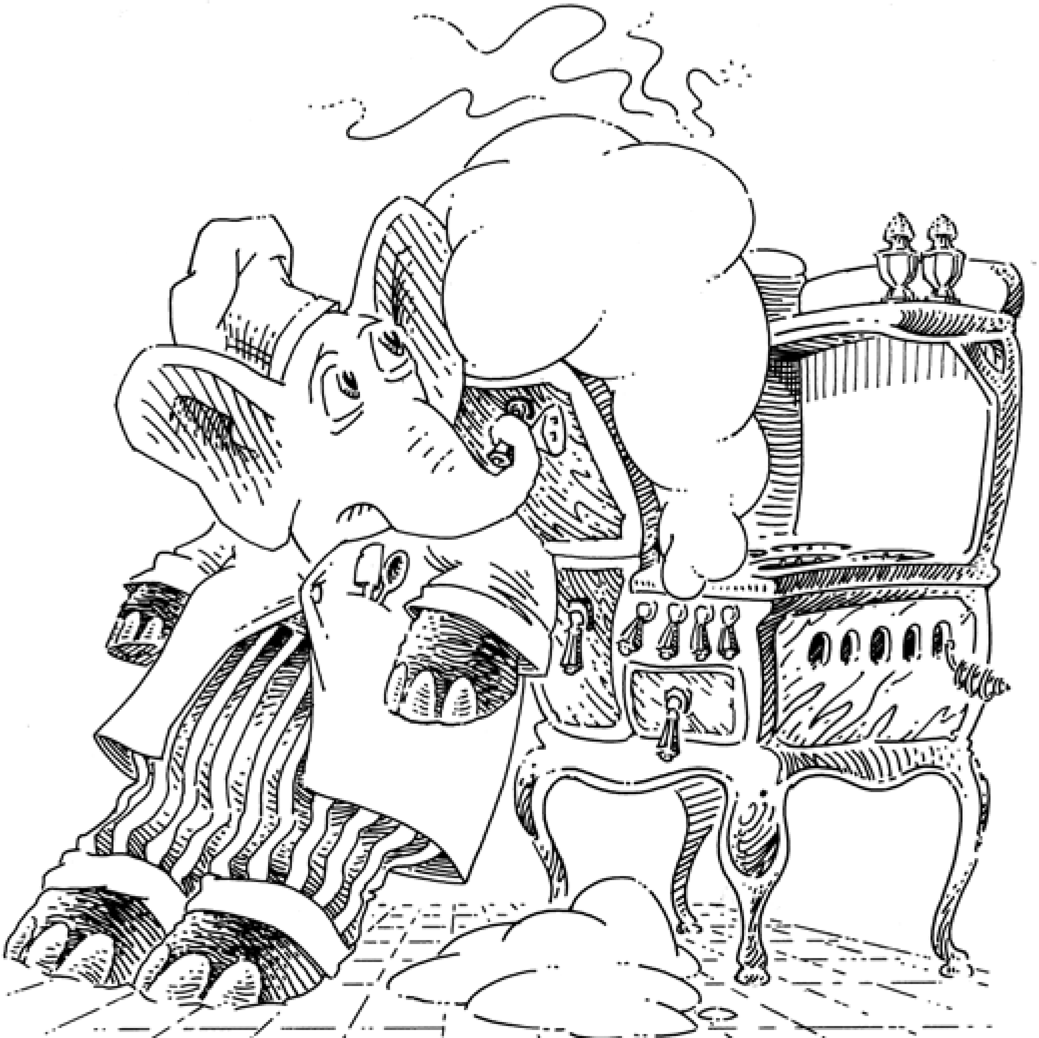
39 Yes,
since none of the **cond**^{e} lines contribute more than one value.

⟹ This is a good time to take a break. ⟸

This is

A BREAK

7.
A Bit Too Much



Is 0 a *bit*?

1 Yes.

Is 1 a bit?

2 Yes.

Is 2 a bit?

3 No.

A bit is either a 0 or a 1.

Which bits are represented by x

4 0 and 1.

Consider the definition of *bit-xor*^{o}.

```
(define bit-xoro
  (lambda (x y r)
    (conde
      ((≡ 0 x) (≡ 0 y) (≡ 0 r))
      ((≡ 1 x) (≡ 0 y) (≡ 1 r))
      ((≡ 0 x) (≡ 1 y) (≡ 1 r))
      ((≡ 1 x) (≡ 1 y) (≡ 0 r))
      (else #u))))
```

When is 0 the value of r

5 When x and y are the same.†

† Another way to define *bit-xor*^{o} is to use *bit-nand*^{o}

```
(define bit-xoro
  (lambda (x y r)
    (fresh (s t u)
      (bit-nando x y s)
      (bit-nando x s t)
      (bit-nando s y u)
      (bit-nando t u r))))
```

where *bit-nand*^{o} is

```
(define bit-nando
  (lambda (x y r)
    (conde
      ((≡ 0 x) (≡ 0 y) (≡ 1 r))
      ((≡ 1 x) (≡ 0 y) (≡ 1 r))
      ((≡ 0 x) (≡ 1 y) (≡ 1 r))
      ((≡ 1 x) (≡ 1 y) (≡ 0 r))
      (else #u))))
```

bit-nand^o is a universal binary boolean relation, since it can be used to define all other binary boolean relations.

Demonstrate this using **run***.

6
```
(run* (s)
  (fresh (x y)
    (bit-xoro x y 0)
    (≡ (x y) s)))
```

which has the value

```
((0 0)
 (1 1)).
```

When is 1 the value of r

7 When x and y are different.

Demonstrate this using **run***.

8
```
(run* (s)
  (fresh (x y)
    (bit-xoro x y 1)
    (≡ (x y) s)))
```
which has the value
```
((1 0)
 (0 1)).
```

What is the value of
```
(run* (s)
  (fresh (x y r)
    (bit-xoro x y r)
    (≡ (x y r) s)))
```

9
```
((0 0 0)
 (1 0 1)
 (0 1 1)
 (1 1 0)).
```

Consider the definition of *bit-and*^{o}.

```
(define bit-ando
  (lambda (x y r)
    (conde
      ((≡ 0 x) (≡ 0 y) (≡ 0 r))
      ((≡ 1 x) (≡ 0 y) (≡ 0 r))
      ((≡ 0 x) (≡ 1 y) (≡ 0 r))
      ((≡ 1 x) (≡ 1 y) (≡ 1 r))
      (else #u))))
```

When is 1 the value of r

10 When x and y are both 1.†

† Another way to define *bit-and*^{o} is to use *bit-nand*^{o} and *bit-not*^{o}

```
(define bit-ando
  (lambda (x y r)
    (fresh (s)
      (bit-nando x y s)
      (bit-noto s r))))
```

where *bit-not*^{o} itself is defined in terms of *bit-nand*^{o}

```
(define bit-noto
  (lambda (x r)
    (bit-nando x x r)))
```

Demonstrate this using **run***.

11
```
(run* (s)
  (fresh (x y)
    (bit-ando x y 1)
    (≡ (x y) s)))
```
which has the value
```
((1 1)).
```

Consider the definition of $half\text{-}adder^o$.

```
(define half-adderᵒ
  (lambda (x y r c)
    (all
      (bit-xorᵒ x y r)
      (bit-andᵒ x y c))))
```

What value is associated with r in

```
(run* (r)
  (half-adderᵒ 1 1 r 1))
```

12 0.†

† $half\text{-}adder^o$ can be redefined as follows.

```
(define half-adderᵒ
  (lambda (x y r c)
    (condᵉ
      ((≡ 0 x) (≡ 0 y) (≡ 0 r) (≡ 0 c))
      ((≡ 1 x) (≡ 0 y) (≡ 1 r) (≡ 0 c))
      ((≡ 0 x) (≡ 1 y) (≡ 1 r) (≡ 0 c))
      ((≡ 1 x) (≡ 1 y) (≡ 0 r) (≡ 1 c))
      (else #u))))
```

What is the value of

```
(run* (s)
  (fresh (x y r c)
    (half-adderᵒ x y r c)
    (≡ (x y r c) s)))
```

13

```
((0 0 0 0)
 (1 0 1 0)
 (0 1 1 0)
 (1 1 0 1)).
```

Describe $half\text{-}adder^o$.

14 Given the bits x, y, r, and c, $half\text{-}adder^o$ satisfies $x + y = r + 2 \cdot c$.

Here is $full\text{-}adder^o$.

```
(define full-adderᵒ
  (lambda (b x y r c)
    (fresh (w xy wz)
      (half-adderᵒ x y w xy)
      (half-adderᵒ w b r wz)
      (bit-xorᵒ xy wz c))))
```

The x, y, r, and c variables serve the same purpose as in $half\text{-}adder^o$. $full\text{-}adder^o$ also takes a carry-in bit, b. What value is associated with s in

```
(run* (s)
  (fresh (r c)
    (full-adderᵒ 0 1 1 r c)
    (≡ (r c) s)))
```

15 (0 1).†

† $full\text{-}adder^o$ can be redefined as follows.

```
(define full-adderᵒ
  (lambda (b x y r c)
    (condᵉ
      ((≡ 0 b) (≡ 0 x) (≡ 0 y) (≡ 0 r) (≡ 0 c))
      ((≡ 1 b) (≡ 0 x) (≡ 0 y) (≡ 1 r) (≡ 0 c))
      ((≡ 0 b) (≡ 1 x) (≡ 0 y) (≡ 1 r) (≡ 0 c))
      ((≡ 1 b) (≡ 1 x) (≡ 0 y) (≡ 0 r) (≡ 1 c))
      ((≡ 0 b) (≡ 0 x) (≡ 1 y) (≡ 1 r) (≡ 0 c))
      ((≡ 1 b) (≡ 0 x) (≡ 1 y) (≡ 0 r) (≡ 1 c))
      ((≡ 0 b) (≡ 1 x) (≡ 1 y) (≡ 0 r) (≡ 1 c))
      ((≡ 1 b) (≡ 1 x) (≡ 1 y) (≡ 1 r) (≡ 1 c))
      (else #u))))
```

What value is associated with s in

```
(run* (s)
  (fresh (r c)
    (full-adderᵒ 1 1 1 r c)
    (≡ (r c) s)))
```

16 (1 1).

What is the value of

```
(run* (s)
  (fresh (b x y r c)
    (full-adderᵒ b x y r c)
    (≡ (b x y r c) s)))
```

17
```
((0 0 0 0 0)
 (1 0 0 1 0)
 (0 1 0 1 0)
 (1 1 0 0 1)
 (0 0 1 1 0)
 (1 0 1 0 1)
 (0 1 1 0 1)
 (1 1 1 1 1)).
```

Describe *full-adder*^{o}.

18 Given the bits b, x, y, r, and c, *full-adder*^{o} satisfies $b + x + y = r + 2 \cdot c$.

What is a *number*?

19 A number is an integer greater than or equal to zero.

Is each number represented by a bit?

20 No.
Each number is represented as a *list* of bits.

Which list represents the number zero?

21 (0)?

Not quite. Try again.

22 How about the empty list ()?

Correct. Is there any number that (0) represents?

23 No.
Each number is represented uniquely, therefore (0) cannot also represent the number zero.

Which list represents the number one?	24	`(1)`, because the value of `(1)` is $1 \cdot 2^0$, which is the number one.
Which number is represented by `(1 0 1)`	25	`5`, because the value of `(1 0 1)` is $1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2$, which is the same as $1 + 0 + 4$, which is five.
Correct. Which number is represented by `(1 1 1)`	26	`7`, because the value of `(1 1 1)` is $1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2$, which is the same as $1 + 2 + 4$, which is seven.
Also correct. Which list represents `9`	27	`(1 0 0 1)`, because the value of `(1 0 0 1)` is $1 \cdot 2^0 + 0 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3$, which is the same as $1 + 0 + 0 + 8$, which is nine.
Yes. How do we represent `6`	28	As the list `(1 1 0)`?
No. Try again.	29	Then it must be `(0 1 1)`, because the value of `(0 1 1)` is $0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2$, which is the same as $0 + 2 + 4$, which is six.
Correct. Does this seem unusual?	30	Yes, it seems very unusual.
How do we represent `19`	31	As the list `(1 1 0 0 1)`?
Yes. How do we represent `17290`	32	As the list `(0 1 0 1 0 0 0 1 1 1 0 0 0 0 1)`?

Correct again. What is interesting about the lists that represent the numbers that we have seen?	33	They contain only 0's and 1's.
Yes. What else is interesting?	34	Every list ends with a 1.
Does every list representation of a number end with a 1?	35	Yes, except for the empty list (), which represents zero.
Compare the numbers represented by n and (0 . n)	36	(0 . n) is twice n. But n cannot be (), since (0 . n) is (0), which does not represent a number.
If n were (1 0 1), what would (0 . n) be?	37	(0 1 0 1), since twice five is ten.
Compare the numbers represented by n and (1 . n)	38	(1 . n) is one more than twice n, even when n is ().
If n were (1 0 1), what would (1 . n) be?	39	(1 1 0 1), since one more than twice five is eleven.
What is the value of (*build-num* 0)	40	().
What is the value of (*build-num* 36)	41	(0 0 1 0 0 1).
What is the value of (*build-num* 19)	42	(1 1 0 0 1).

Define *build-num*.

43 Here is one way to define it.

```
(define build-num
  (lambda (n)
    (cond
      ((zero? n) ())
      ((and (not (zero? n)) (even? n))
       (cons 0
         (build-num (÷ n 2))))
      ((odd? n)
       (cons 1
         (build-num (÷ (− n 1) 2)))))))
```

Redefine *build-num*, where (*zero? n*) is not the question of the first **cond** line.

44 That's easy.

```
(define build-num
  (lambda (n)
    (cond
      ((odd? n)
       (cons 1
         (build-num (÷ (− n 1) 2))))
      ((and (not (zero? n)) (even? n))
       (cons 0
         (build-num (÷ n 2))))
      ((zero? n) ()))))
```

Is there anything interesting about these definitions of *build-num*

45 For any number *n*, one and only one **cond** question is true.†

† Thank you Edsger W. Dijkstra (1930–2002).

Can we rearrange the **cond** lines in any order?

46 Yes.
This is called the *non-overlapping property*. It appears rather frequently throughout this and the next chapter.

What is the sum of `(1)` and `(1)`	47	`(0 1)`, which is just two.
What is the sum of `(0 0 0 1)` and `(1 1 1)`	48	`(1 1 1 1)`, which is just fifteen.
What is the sum of `(1 1 1)` and `(0 0 0 1)`	49	`(1 1 1 1)`, which is just fifteen.
What is the sum of `(1 1 0 0 1)` and `()`	50	`(1 1 0 0 1)`, which is just nineteen.
What is the sum of `()` and `(1 1 0 0 1)`	51	`(1 1 0 0 1)`, which is just nineteen.
What is the sum of `(1 1 1 0 1)` and `(1)`	52	`(0 0 0 1 1)`, which is just twenty-four.
Which number is represented by `(`x `1)`	53	It depends on what x is.
Which number would be represented by `(`x `1)` if x were `0`?	54	Two, which is represented by `(0 1)`.
Which number would be represented by `(`x `1)` if x were `1`?	55	Three, which is represented by `(1 1)`.
So which numbers are represented by `(`x `1)`	56	Two and three.
Which numbers are represented by `(`x x `1)`	57	Four and seven, which are represented by `(0 0 1)` and `(1 1 1)`, respectively.

Which numbers are represented by

$(x \ 0 \ y \ 1)$

[58] Eight, nine, twelve, and thirteen, which are represented by (0 0 0 1), (1 0 0 1), (0 0 1 1), and (1 0 1 1), respectively.

Which numbers are represented by

$(x \ 0 \ y \ z)$

[59] Once again, eight, nine, twelve, and thirteen, which are represented by (0 0 0 1), (1 0 0 1), (0 0 1 1), and (1 0 1 1), respectively.

Why do both $(x \ 0 \ y \ 1)$ and $(x \ 0 \ y \ z)$ represent the same numbers?

[60] Because z must be either a 0 or a 1. If z were 0, then $(x \ 0 \ y \ z)$ would not represent any number. Therefore z must be 1.

Which number is represented by

(x)

[61] One, which is represented by (1), since (0) does not represent a number.

What does z represent?

[62] Every number greater than or equal to zero.

Which numbers are represented by

$(1 \ . \ z)$

[63] It depends on what z is.

Which number is represented by

$(1 \ . \ z)$

where z is ()

[64] One, since (1 . ()) is (1).

Which number is represented by

$(1 \ . \ z)$

where z is (1)

[65] Three, since (1 . (1)) is (1 1).

Which number is represented by `(1 . ` z`)` where z is `(0 1)`	66 Five, since `(1 . (0 1))` is `(1 0 1)`.
So which numbers are represented by `(1 . ` z`)`	67 All the odd numbers?
Right. Then, which numbers are represented by `(0 . ` z`)`	68 All the even numbers?
Not quite. Which even number is not of the form `(0 . ` z`)`	69 Zero, which is represented by `()`.
For which values of z does `(0 . ` z`)` represent numbers?	70 All numbers greater than zero.
Are the even numbers all the numbers that are multiples of two?	71 Yes.
Which numbers are represented by `(0 0 . ` z`)`	72 Every other even number, starting with four.
Which numbers are represented by `(0 1 . ` z`)`	73 Every other even number, starting with two.
Which numbers are represented by `(1 0 . ` z`)`	74 Every other odd number, starting with five.

Which numbers are represented by

$(1\ 0\ y\ .\ z)$

75 Once again, every other odd number, starting with five.

Why do $(1\ 0\ .\ z)$ and $(1\ 0\ y\ .\ z)$ represent the same numbers?

76 Because z cannot be the empty list in $(1\ 0\ .\ z)$ and y cannot be 0 when z is the empty list in $(1\ 0\ y\ .\ z)$.

Which numbers are represented by

$(0\ y\ .\ z)$

77 *Every* even number, starting with two.

Which numbers are represented by

$(1\ y\ .\ z)$

78 *Every* odd number, starting with three.

Which numbers are represented by

$(y\ .\ z)$

79 *Every* number, starting with one—in other words, the positive numbers.

Consider the definition of pos^o.

```
(define pos^o
  (lambda (n)
    (fresh (a d)
      (≡ (a . d) n))))
```

What value is associated with q in

```
(run* (q)
  (pos^o (0 1 1))
  (≡ #t q))
```

80 #t.

What value is associated with q in

```
(run* (q)
  (pos^o (1))
  (≡ #t q))
```

81 #t.

What is the value of

(**run*** (q)
  (*pos*^{o} **()**)
  ($\equiv$ `#t` q))

82 **()**.

What value is associated with r in

(**run*** (r)
  (*pos*^{o} r))

83 **(**$\_{0}$ **.** $\_{1}$**)**.

Does this mean that (*pos*^{o} r) always succeeds when r is a fresh variable?

84 Yes.

Which numbers are represented by

(x y **.** z**)**

85 *Every* number, starting with two—in other words, every number greater than one.

Consider the definition of $>$**1**^{o}.

```
(define >1°
  (lambda (n)
    (fresh (a ad dd)†
      (≡ (a ad . dd) n))))
```

What value is associated with q in

(**run*** (q)
  ($>$**1**^{o} **(**0 1 1**)**)
  ($\equiv$ `#t` q))

† The names *a*, *ad*, and *dd* correspond to *car*, *cadr*, and *cddr*.

86 `#t`.

What is the value of

(**run*** (q)
  ($>$**1**^{o} **(**0 1**)**)
  ($\equiv$ `#t` q))

87 **(**`#t`**)**.

What is the value of
(**run*** (q)
  (>**1**^{o} **(**1**)**)
  (≡ #t q))
	[88] **()**.

What is the value of
(**run*** (q)
  (>**1**^{o} **()**)
  (≡ #t q))
	[89] **()**.

What value is associated with r in (**run*** (r) (>**1**^{o} r))	[90] **(**$\_{0}$ $\_{1}$ **.** $\_{2}$**)**.
Does this mean that (>**1**^{o} r) always succeeds when r is a fresh variable?	[91] Yes.
An *n-representative* is the first n bits of a number, up to and including the rightmost 1. If there is no rightmost 1, then the n-representative is the empty list. What is the n-representative of **(**0 1 1**)**	[92] **(**0 1 1**)**.
What is the n-representative of **(**0 x 1 0 y **.** z**)**	[93] **(**0 x 1**)**, since everything to the right of the rightmost 1 is ignored.
What is the n-representative of **(**0 0 y **.** z**)**	[94] **()**, since there is no rightmost 1.
What is the n-representative of z	[95] **()**.

What is the value of†

(**run³** (*s*)
  (**fresh** (*x y r*)
    (*adder*^{o} 0 *x y r*)
    (≡ (*x y r*) *s*)))

96 That depends on the definition of *adder*^{o}, which we do not see until frame 118. But we can understand *adder*^{o}: given the bit *d*, and the numbers *n*, *m*, and *r*, *adder*^{o} satisfies $d + n + m = r$.

What is the value of†

(**run³** (*s*)
  (**fresh** (*x y r*)
    (*adder*^{o} 0 *x y r*)
    (≡ (*x y r*) *s*)))

97 $((\_{0}\ ()\ \_{0})$
$(()\ (\_{0} \cdot \_{1})\ (\_{0} \cdot \_{1}))$
$((1)\ (1)\ (0\ 1)))$.

(*adder*^{o} 0 *x y r*) sums *x* and *y* to produce *r*. For example, in the first value, zero added to a number is the number. In the second value, the sum of () and $(\_{0} \cdot \_{1})$ is $(\_{0} \cdot \_{1})$. In other words, the sum of zero and a positive number is the positive number.

Is ((1) (1) (0 1)) a *ground* value?

98 Yes.

Is $(\_{0}\ ()\ \_{0})$ a ground value?

99 No,
because it contains one or more variables.†

† In fact, $(\_{0}\ ()\ \_{0})$ has no variables, however prior to being reified, it contained two occurrences of the same variable.

What can we say about the three values in frame 97?

100 The third value is ground and the other two values are not.

Before reading the next frame,

Treat Yourself to a Hot Fudge Sundae!

What is the value of

(**run**$^{\mathbf{22}}$ (s)
  (**fresh** $(x\ y\ r)$
    ($adder^{o}$ 0 $x\ y\ r$)
    ($\equiv$ ($x\ y\ r$) s)))

101 (($\_{0}$ () $\_{0}$)
(() ($\_{0}$. $\_{1}$) ($\_{0}$. $\_{1}$))
((1) (1) (0 1))
((1) (0 $\_{0}$. $\_{1}$) (1 $\_{0}$. $\_{1}$))
((0 $\_{0}$. $\_{1}$) (1) (1 $\_{0}$. $\_{1}$))
((1) (1 1) (0 0 1))
((0 1) (0 1) (0 0 1))
((1) (1 0 $\_{0}$. $\_{1}$) (0 1 $\_{0}$. $\_{1}$))
((1 1) (1) (0 0 1))
((1) (1 1 1) (0 0 0 1))
((1 1) (0 1) (1 0 1))
((1) (1 1 0 $\_{0}$. $\_{1}$) (0 0 1 $\_{0}$. $\_{1}$))
((1 0 $\_{0}$. $\_{1}$) (1) (0 1 $\_{0}$. $\_{1}$))
((1) (1 1 1 1) (0 0 0 0 1))
((0 1) (0 0 $\_{0}$. $\_{1}$) (0 1 $\_{0}$. $\_{1}$))
((1) (1 1 1 0 $\_{0}$. $\_{1}$) (0 0 0 1 $\_{0}$. $\_{1}$))
((1 1 1) (1) (0 0 0 1))
((1) (1 1 1 1 1) (0 0 0 0 0 1))
((0 1) (1 1) (1 0 1))
((1) (1 1 1 1 0 $\_{0}$. $\_{1}$) (0 0 0 0 1 $\_{0}$. $\_{1}$))
((1 1 0 $\_{0}$. $\_{1}$) (1) (0 0 1 $\_{0}$. $\_{1}$))
((1) (1 1 1 1 1 1) (0 0 0 0 0 0 1))).

How many of its values are ground, and how many are not?

102 Eleven values are ground and eleven values are not.

What are the nonground values?

103 (($\_{0}$ () $\_{0}$)
(() ($\_{0}$. $\_{1}$) ($\_{0}$. $\_{1}$))
((1) (0 $\_{0}$. $\_{1}$) (1 $\_{0}$. $\_{1}$))
((0 $\_{0}$. $\_{1}$) (1) (1 $\_{0}$. $\_{1}$))
((1) (1 0 $\_{0}$. $\_{1}$) (0 1 $\_{0}$. $\_{1}$))
((1) (1 1 0 $\_{0}$. $\_{1}$) (0 0 1 $\_{0}$. $\_{1}$))
((1 0 $\_{0}$. $\_{1}$) (1) (0 1 $\_{0}$. $\_{1}$))
((0 1) (0 0 $\_{0}$. $\_{1}$) (0 1 $\_{0}$. $\_{1}$))
((1) (1 1 1 0 $\_{0}$. $\_{1}$) (0 0 0 1 $\_{0}$. $\_{1}$))
((1) (1 1 1 1 0 $\_{0}$. $\_{1}$) (0 0 0 0 1 $\_{0}$. $\_{1}$))
((1 1 0 $\_{0}$. $\_{1}$) (1) (0 0 1 $\_{0}$. $\_{1}$))).

What interesting property do these eleven values possess?

104 The *width*† of r is the same as the width of the wider of x and y.

† The *width* of a number n can be defined as

```
(define width
  (lambda (n)
    (cond
      ((null? n) 0)
      ((pair? n) (+ (width (cdr n)) 1))
      (else 1))))
```

What is another interesting property that these eleven values possess?

105 Variables appear in r, and in either x or y, but not in both.

What is another interesting property that these eleven values possess?

106 Except for the first value, r always ends with $\_{0}\ .\ \_{1}$ as does the wider of x and y.

What is another interesting property that these eleven values possess?

107 The n-representative of r is equal to the sum of the n-representatives of x and y.

In the ninth value, for example, the sum of `(1)` and `(1 1 1)` is `(0 0 0 1)`.

Describe the third value.

108 Huh?

Here x is `(1)` and y is $(0\ \_{0}\ .\ \_{1})$, a positive even number. Adding x to y yields the odd numbers greater than one. Is the fifth value the same as the seventh?

109 Almost,

since $x + y = y + x$.

Does each value have a corresponding value in which x and y are swapped?

110 No.

For example, the first two values do not correspond to any other values.

111 What is the corresponding value for the tenth value?

`((1 1 1 1 0 _0 . _1) (1) (0 0 0 0 1 _0 . _1))`. However, this is the nineteenth nonground value, and we have presented only the first eleven.

---

112 Describe the seventh value.

Frame 75 shows that `(1 0 _0 . _1)` represents every other odd number, starting at five. Incrementing each of those numbers by one produces every other even number, starting at six, which is represented by `(0 1 _0 . _1)`.

---

113 Describe the eighth value.

The eighth value is like the third value, but with an additional leading `0`. In other words, each number is doubled.

---

114 Describe the 198th value, which has the value `((0 0 1) (1 0 0 _0 . _1) (1 0 1 _0 . _1))`.

`(1 0 0 _0 . _1)` represents every fourth odd number, starting at nine. Incrementing each of those numbers by four produces every fourth odd number, starting at thirteen, which is represented by `(1 0 1 _0 . _1)`.

---

115 What are the ground values of frame 101?

```
(((1) (1) (0 1))
 ((1) (1 1) (0 0 1))
 ((0 1) (0 1) (0 0 1))
 ((1 1) (1) (0 0 1))
 ((1) (1 1 1) (0 0 0 1))
 ((1 1) (0 1) (1 0 1))
 ((1) (1 1 1 1) (0 0 0 0 1))
 ((1 1 1) (1) (0 0 0 1))
 ((1) (1 1 1 1 1) (0 0 0 0 0 1))
 ((0 1) (1 1) (1 0 1))
 ((1) (1 1 1 1 1 1) (0 0 0 0 0 0 1))).
```

---

116 What interesting property do these values possess?

The width of r is *one greater* than the width of the wider of x and y.

What is another interesting property of these values?

117 Each list cannot be created from any list in frame 103, regardless of which values are chosen for the variables there. This is an example of the non-overlapping property described in frame 46.

Here are *adder*^{o} and *gen-adder*^{o}.

```
(define addero
  (lambda (d n m r)
    (condi
      ((≡ 0 d) (≡ () m) (≡ n r))
      ((≡ 0 d) (≡ () n) (≡ m r)
       (poso m))
      ((≡ 1 d) (≡ () m)
       (addero 0 n (1) r))
      ((≡ 1 d) (≡ () n) (poso m)
       (addero 0 (1) m r))
      ((≡ (1) n) (≡ (1) m)
       (fresh (a c)
         (≡ (a c) r)
         (full-addero d 1 1 a c)))
      ((≡ (1) n) (gen-addero d n m r))
      ((≡ (1) m) (>1o n) (>1o r)
       (addero d (1) n r))
      ((>1o n) (gen-addero d n m r))
      (else #u))))
```

```
(define gen-addero
  (lambda (d n m r)
    (fresh (a b c e x y z)
      (≡ (a . x) n)
      (≡ (b . y) m) (poso y)
      (≡ (c . z) r) (poso z)
      (alli
        (full-addero d a b c e)
        (addero e x y z)))))
```

What is d

118 A *carry* bit.[†]

[†] See 10:26 for why *gen-adder*^{o} requires **all**^{i} instead of **all**.

What are n, m, and r

119 They are numbers.

What value is associated with s in

(**run*** (s)
  (*gen-adder*^{o} 1 (0 1 1) (1 1) s))

120 (0 1 0 1).

What are a, b, c, d, and e

121 They are bits.

What are n, m, r, x, y, and z

122 They are numbers.

In the definition of *gen-adder*^{o}, (*pos*^{o} y) and (*pos*^{o} z) follow ($\equiv$ (b . y) m) and ($\equiv$ (c . z) r), respectively. Why isn't there a (*pos*^{o} x)

123 Because in the first call to *gen-adder*^{o} from *adder*^{o}, n can be (1).

What about the other call to *gen-adder*^{o} from *adder*^{o}

124 The (>**1**^{o} n) call that precedes the call to *gen-adder*^{o} is the same as if we had placed a (*pos*^{o} x) following ($\equiv$ (a . x) n). But if we were to use (*pos*^{o} x) in *gen-adder*^{o}, then it would fail for n being (1).

Describe *gen-adder*^{o}.

125 Given the bit d, and the numbers n, m, and r, *gen-adder*^{o} satisfies $d + n + m = r$, provided that n is positive and m and r are greater than one.

What is the value of

(**run*** (s)
  (**fresh** (x y)
    (*adder*^{o} 0 x y (1 0 1))
    ($\equiv$ (x y) s)))

126 (((1 0 1) ())
 (() (1 0 1))
 ((1) (0 0 1))
 ((0 0 1) (1))
 ((1 1) (0 1))
 ((0 1) (1 1))).

Describe the values produced by

(**run*** (s)
  (**fresh** (x y)
    (*adder*^{o} 0 x y (1 0 1))
    ($\equiv$ (x y) s)))

127 The values are the pairs of numbers that sum to five.

We can define $+^o$ using $adder^o$.

```
(define +ᵒ
  (lambda (n m k)
    (adderᵒ 0 n m k)))
```

Use $+^o$ to generate the pairs of numbers that sum to five.

128 Here is an expression that generates the pairs of numbers that sum to five:

```
(run* (s)
  (fresh (x y)
    (+ᵒ x y (1 0 1))
    (≡ (x y) s))).
```

What is the value of

```
(run* (s)
  (fresh (x y)
    (+ᵒ x y (1 0 1))
    (≡ (x y) s)))
```

129

```
(((1 0 1) ())
 (() (1 0 1))
 ((1) (0 0 1))
 ((0 0 1) (1))
 ((1 1) (0 1))
 ((0 1) (1 1))).
```

Now define $-^o$ using $+^o$.

130 That is easy.

```
(define −ᵒ
  (lambda (n m k)
    (+ᵒ m k n)))
```

What is the value of

```
(run* (q)
  (−ᵒ (0 0 0 1) (1 0 1) q))
```

131 ((1 1)).

What is the value of

```
(run* (q)
  (−ᵒ (0 1 1) (0 1 1) q))
```

132 (()).

What is the value of

```
(run* (q)
  (−ᵒ (0 1 1) (0 0 0 1) q))
```

133 ().

Eight cannot be subtracted from six, since we do not represent negative numbers.

⟹ Now go make yourself a baba ghanoush pita wrap. ⟸

This space reserved for

BABA GHANOUSH STAINS!

8. Just a Bit More

What is the value of

```
(run34 (t)
  (fresh (x y r)
    (*o x y r)
    (≡ (x y r) t)))
```

1
```
((() _0 ())
 ((_0 . _1) () ())
 ((1) (_0 . _1) (_0 . _1))
 ((_0 _1 . _2) (1) (_0 _1 . _2))
 ((0 1) (_0 _1 . _2) (0 _0 _1 . _2))
 ((1 _0 . _1) (0 1) (0 1 _0 . _1))
 ((0 0 1) (_0 _1 . _2) (0 0 _0 _1 . _2))
 ((1 1) (1 1) (1 0 0 1))
 ((0 1 _0 . _1) (0 1) (0 0 1 _0 . _1))
 ((1 _0 . _1) (0 0 1) (0 0 1 _0 . _1))
 ((0 0 0 1) (_0 _1 . _2) (0 0 0 _0 _1 . _2))
 ((1 1) (1 0 1) (1 1 1 1))
 ((0 1 1) (1 1) (0 1 0 0 1))
 ((1 1) (0 1 1) (0 1 0 0 1))
 ((0 0 1 _0 . _1) (0 1) (0 0 0 1 _0 . _1))
 ((1 1) (1 1 1) (1 0 1 0 1))
 ((0 1 _0 . _1) (0 0 1) (0 0 0 1 _0 . _1))
 ((1 _0 . _1) (0 0 0 1) (0 0 0 1 _0 . _1))
 ((0 0 0 0 1) (_0 _1 . _2) (0 0 0 0 _0 _1 . _2))
 ((1 0 1) (1 1) (1 1 1 1))
 ((0 1 1) (1 0 1) (0 1 1 1 1))
 ((1 0 1) (0 1 1) (0 1 1 1 1))
 ((0 0 1 1) (1 1) (0 0 1 0 0 1))
 ((1 1) (1 0 0 1) (1 1 0 1 1))
 ((0 1 1) (0 1 1) (0 0 1 0 0 1))
 ((1 1) (0 0 1 1) (0 0 1 0 0 1))
 ((0 0 0 1 _0 . _1) (0 1) (0 0 0 0 1 _0 . _1))
 ((1 1) (1 1 0 1) (1 0 0 0 0 1))
 ((0 1 1) (1 1 1) (0 1 0 1 0 1))
 ((1 1 1) (0 1 1) (0 1 0 1 0 1))
 ((0 0 1 _0 . _1) (0 0 1) (0 0 0 0 1 _0 . _1))
 ((1 1) (1 0 1 1) (1 1 1 0 0 1))
 ((0 1 _0 . _1) (0 0 0 1) (0 0 0 0 1 _0 . _1))
 ((1 _0 . _1) (0 0 0 0 1) (0 0 0 0 1 _0 . _1))).
```

It is difficult to see patterns when looking at all thirty-four values. Would it be easier to examine only the nonground values?

2 Yes,
thanks.

What are the first eighteen nonground values?

3
```
((() _0 ())
 ((_0 . _1) () ())
 ((1) (_0 . _1) (_0 . _1))
 ((_0 _1 . _2) (1) (_0 _1 . _2))
 ((0 1) (_0 _1 . _2) (0 _0 _1 . _2))
 ((1 _0 . _1) (0 1) (0 1 _0 . _1))
 ((0 0 1) (_0 _1 . _2) (0 0 _0 _1 . _2))
 ((0 1 _0 . _1) (0 1) (0 0 1 _0 . _1))
 ((1 _0 . _1) (0 0 1) (0 0 1 _0 . _1))
 ((0 0 0 1) (_0 _1 . _2) (0 0 0 _0 _1 . _2))
 ((0 0 1 _0 . _1) (0 1) (0 0 0 1 _0 . _1))
 ((0 1 _0 . _1) (0 0 1) (0 0 0 1 _0 . _1))
 ((1 _0 . _1) (0 0 0 1) (0 0 0 1 _0 . _1))
 ((0 0 0 0 1) (_0 _1 . _2) (0 0 0 0 _0 _1 . _2))
 ((0 0 0 1 _0 . _1) (0 1) (0 0 0 0 1 _0 . _1))
 ((0 0 1 _0 . _1) (0 0 1) (0 0 0 0 1 _0 . _1))
 ((0 1 _0 . _1) (0 0 0 1) (0 0 0 0 1 _0 . _1))
 ((1 _0 . _1) (0 0 0 0 1) (0 0 0 0 1 _0 . _1))).
```

The value associated with *p* in

```
(run* (p)
  (*o (0 1) (0 0 1) p))
```

is `(0 0 0 1)`. To which nonground value does this correspond?

4 The fifth nonground value

```
((0 1) (_0 _1 . _2) (0 _0 _1 . _2)).
```

Describe the fifth nonground value.

5 The product of two and a number greater than one is twice the number greater than one.

Describe the sixth nonground value.

6 The product of an odd number, three or greater, and two is twice the odd number.

Is the product of `(1 _0 . _1)` and `(0 1)` odd or even?

7 It is even,
since the first bit of `(0 1 _0 . _1)` is `0`.

Is there a nonground value that shows that the product of three and three is nine?

8 No.

Is there a ground value that shows that the product of three and three is nine?

9 Yes,
the first ground value

((1 1) (1 1) (1 0 0 1))

shows that the product of three and three is nine.

Here is the definition of $*^o$.

```
(define *ᵒ
  (lambda (n m p)
    (condⁱ
      ((≡ () n) (≡ () p))
      ((posᵒ n) (≡ () m) (≡ () p))
      ((≡ (1) n) (posᵒ m) (≡ m p))
      ((>1ᵒ n) (≡ (1) m) (≡ n p))
      ((fresh (x z)
         (≡ (0 . x) n) (posᵒ x)
         (≡ (0 . z) p) (posᵒ z)
         (>1ᵒ m)
         (*ᵒ x m z)))
      ((fresh (x y)
         (≡ (1 . x) n) (posᵒ x)
         (≡ (0 . y) m) (posᵒ y)
         (*ᵒ m n p)))
      ((fresh (x y)
          (≡ (1 . x) n) (posᵒ x)
          (≡ (1 . y) m) (posᵒ y)
          (odd-*ᵒ x n m p)))
      (else #u))))
```

Describe the first and second **cond**^{i} lines.

10 The first **cond**^{i} line says that the product of zero and a number is zero. The second line says that the product of a positive number and zero is also equal to zero.

Why isn't ((≡ () m) (≡ () p)) the second **cond**^{i} line?

11 To avoid producing two values in which both n and m are zero. In other words, we enforce the non-overlapping property.

Describe the third and fourth **cond**^{i} lines.

12 The third **cond**^{i} line says that the product of one and a positive number is the number. The fourth line says that the product of a number greater than one and one is the number.

Describe the fifth **cond**^{i} line.

13 The fifth **cond**^{i} line says that the product of an even positive number and a number greater than one is an even positive number, using the equation $n \cdot m = 2 \cdot (\frac{n}{2} \cdot m)$.

Why do we use this equation?

14 In order for the recursive call to have a value, one of the arguments to $*^o$ must shrink. Dividing n by two clearly shrinks n.

How do we divide n by two?

15 With ($\equiv$ `(0 . x)` n), where x is not `()`.

Describe the sixth **cond**^{i} line.

16 This one is easy. The sixth **cond**^{i} line says that the product of an odd positive number and an even positive number is the same as the product of the even positive number and the odd positive number.

Describe the seventh **cond**^{i} line.

17 This one is also easy. The seventh **cond**^{i} line says that the product of an odd number greater than one and another odd number greater than one is the result of (odd-$*^o$ x n m p), where x is $\frac{n-1}{2}$.

Here is odd-$*^o$.

```
(define odd-*o
  (lambda (x n m p)
    (fresh (q)
      (bound-*o q p n m)
      (*o x m q)
      (+o (0 . q) m p))))
```

If we ignore $bound$-$*^o$, what equation describes the work done in odd-$*^o$

18 We know that x is $\frac{n-1}{2}$. Therefore,
$n \cdot m = 2 \cdot (\frac{n-1}{2} \cdot m) + m.$

Here is a hypothetical definition of *bound-*$*^o$.

```
(define bound-*ᵒ
  (lambda (q p n m)
    #s))
```

19 Okay, so this is not the final definition of *bound-*$*^o$.

Using the hypothetical definition of *bound-*$*^o$, what value would be associated with t in

```
(run¹ (t)
  (fresh (n m)
    (*ᵒ n m (1))
    (≡ (n m) t)))
```

20 **((1) (1))**.
This value is contributed by the third **cond**^{i} line of $*^o$.

Now what would be the value of

```
(run² (t)
  (fresh (n m)
    (*ᵒ n m (1))
    (≡ (n m) t)))
```

21 It would have no value,
because **run** would never finish determining the *second* value.

Here is *bound-*$*^o$.

```
(define bound-*ᵒ
  (lambda (q p n m)
    (condᵉ
      ((nullᵒ q) (pairᵒ p))
      (else
        (fresh (x y z)
          (cdrᵒ q x)
          (cdrᵒ p y)
          (condⁱ
            ((nullᵒ n)
             (cdrᵒ m z)
             (bound-*ᵒ x y z ()))
            (else
              (cdrᵒ n z)
              (bound-*ᵒ x y z m)))))))))
```

Is this definition recursive?

22 Clearly.

What is the value of

```
(run² (t)
  (fresh (n m)
    (*ᵒ n m (1))
    (≡ (n m) t)))
```

23 (((1) (1))),
because *bound-*ᵒ* fails when the product of n and m is larger than p, and since the length of n plus the length of m is an upper bound on the length of p.

What value is associated with p in

```
(run* (p)
  (*ᵒ (1 1 1) (1 1 1 1 1 1) p))
```

24 (1 0 0 1 1 1 0 1 1),
which contains nine bits.

If we replace a 1 by a 0 in

```
(*ᵒ (1 1 1) (1 1 1 1 1 1) p),
```

is nine still the maximum length of p

25 Yes,
because (1 1 1) and (1 1 1 1 1 1) represent the largest numbers of lengths three and six, respectively. Of course the rightmost 1 in each number cannot be replaced by a 0.

Here is the definition of $=l^o$.

```
(define =lᵒ
  (lambda (n m)
    (condᵉ
      ((≡ () n) (≡ () m))
      ((≡ (1) n) (≡ (1) m))
      (else
        (fresh (a x b y)
          (≡ (a . x) n) (posᵒ x)
          (≡ (b . y) m) (posᵒ y)
          (=lᵒ x y))))))
```

Is this definition recursive?

26 Yes, it is.

What value is associated with t in

```
(run* (t)
  (fresh (w x y)
    (=lᵒ (1 w x . y) (0 1 1 0 1))
    (≡ (w x y) t)))
```

27 ($\_{0}$ $\_{1}$ ($\_{2}$ 1)),
since y is ($\_{2}$ 1), the *length* of (1 w x . y) is the same as the length of (0 1 1 0 1).

What value is associated with *b* in

```
(run* (b)
  (=lo (1) (b)))
```

28 1,
because if *b* were associated with 0, then (*b*) would have become (0), which does not represent a number.

What value is associated with *n* in

```
(run* (n)
  (=lo (1 0 1 . n) (0 1 1 0 1)))
```

29 ($\_{0}$ 1),
because if *n* were ($\_{0}$ 1), then the length of (1 0 1 . *n*) would be the same as the length of (0 1 1 0 1).

What is the value of

```
(run5 (t)
  (fresh (y z)
    (=lo (1 . y) (1 . z))
    (≡ (y z) t)))
```

30
```
((() ())
 ((1) (1))
 ((_0 1) (_1 1))
 ((_0 _1 1) (_2 _3 1))
 ((_0 _1 _2 1) (_3 _4 _5 1))),
```
because each *y* and *z* must be the same length in order for (1 . *y*) and (1 . *z*) to be the same length.

What is the value of

```
(run5 (t)
  (fresh (y z)
    (=lo (1 . y) (0 . z))
    (≡ (y z) t)))
```

31
```
(((1) (1))
 ((_0 1) (_1 1))
 ((_0 _1 1) (_2 _3 1))
 ((_0 _1 _2 1) (_3 _4 _5 1))
 ((_0 _1 _2 _3 1) (_4 _5 _6 _7 1))).
```

Why isn't (() ()) the first value?

32 Because if *z* were (), then (0 . *z*) would not represent a number.

What is the value of

```
(run5 (t)
  (fresh (y z)
    (=lo (1 . y) (0 1 1 0 1 . z))
    (≡ (y z) t)))
```

33
```
(((_0 _1 _2 1) ())
 ((_0 _1 _2 _3 1) (1))
 ((_0 _1 _2 _3 _4 1) (_5 1))
 ((_0 _1 _2 _3 _4 _5 1) (_6 _7 1))
 ((_0 _1 _2 _3 _4 _5 _6 1) (_7 _8 _9 1))),
```
because the shortest *z* is (), which forces *y* to be a list of length four. Thereafter, as *y* grows in length, so does *z*.

Here is the definition of $<l^o$.

```
(define <lᵒ
  (lambda (n m)
    (condeᵉ
      ((≡ () n) (posᵒ m))
      ((≡ (1) n) (>1ᵒ m))
      (else
        (fresh (a x b y)
          (≡ (a . x) n) (posᵒ x)
          (≡ (b . y) m) (posᵒ y)
          (<lᵒ x y))))))
```

How does this definition differ from the definition of $=l^o$

34 In the first **cond**^{e} line, ($\equiv$ **()** m) is replaced by (pos^o m). In the second line, ($\equiv$ **(1)** m) is replaced by ($>$**1**^{o} m). This guarantees that n is shorter than m.

What is the value of

```
(run⁸ (t)
  (fresh (y z)
    (<lᵒ (1 . y) (0 1 1 0 1 . z))
    (≡ (y z) t)))
```

35

```
((() _0)
 ((1) _0)
 ((_0 1) _1)
 ((_0 _1 1) _2)
 ((_0 _1 _2 1) (_3 . _4))
 ((_0 _1 _2 _3 1) (_4 _5 . _6))
 ((_0 _1 _2 _3 _4 1) (_5 _6 _7 . _8))
 ((_0 _1 _2 _3 _4 _5 1) (_6 _7 _8 _9 . _10))).
```

Why does z remain fresh in the first four values?

36 The variable y is associated with a list that represents a number. If the length of this list is at most three, then **(**1 **.** y**)** is shorter than **(**0 1 1 0 1 **.** z**)**, regardless of the value associated with z.

What is the value of

```
(run¹ (n)
  (<lᵒ n n))
```

37 It has no value.

Clearly the first two **cond**^{e} lines fail. In the recursive call, x and y are associated with the same fresh variable, which is where we started.

Define $\leqslant l^o$ using $=l^o$ and $<l^o$.

38 Is this correct?

```
(define ≤lᵒ
  (lambda (n m)
    (condᵉ
      ((=lᵒ n m) #s)
      ((<lᵒ n m) #s)
      (else #u))))
```

It looks like it might be correct. What is the value of

```
(run⁸ (t)
  (fresh (n m)
    (≤lᵒ n m)
    (≡ (n m) t)))
```

39

```
((() ())
 ((1) (1))
 ((_0 1) (_1 1))
 ((_0 _1 1) (_2 _3 1))
 ((_0 _1 _2 1) (_3 _4 _5 1))
 ((_0 _1 _2 _3 1) (_4 _5 _6 _7 1))
 ((_0 _1 _2 _3 _4 1) (_5 _6 _7 _8 _9 1))
 ((_0 _1 _2 _3 _4 _5 1) (_6 _7 _8 _9 _10 _11 1)))
```

What value is associated with t in

```
(run¹ (t)
  (fresh (n m)
    (≤lᵒ n m)
    (*ᵒ n (0 1) m)
    (≡ (n m) t)))
```

40 (() ()).

What is the value of

```
(run² (t)
  (fresh (n m)
    (≤lᵒ n m)
    (*ᵒ n (0 1) m)
    (≡ (n m) t)))
```

41 It has no value,
because the first **cond**^{e} line of $\leqslant l^o$ always succeeds, which means that n and m are always the same length. Therefore ($*^o$ n (0 1) m) succeeds only when n is ().

How can we redefine $\leqslant l^o$ so that

```
(run^2 (t)
  (fresh (n m)
    (≤l^o n m)
    (*^o n (0 1) m)
    (≡ (n m) t)))
```

has a value?

42 Let's use **cond**^{i}.

```
(define ≤l^o
  (lambda (n m)
    (cond^i
      ((=l^o n m) #s)
      ((<l^o n m) #s)
      (else #u))))
```

What is the value of

```
(run^10 (t)
  (fresh (n m)
    (≤l^o n m)
    (*^o n (0 1) m)
    (≡ (n m) t)))
```

43

```
((() ())
 ((1) (0 1))
 ((0 1) (0 0 1))
 ((1 1) (0 1 1))
 ((0 0 1) (0 0 0 1))
 ((1 _0 1) (0 1 _0 1))
 ((0 1 1) (0 0 1 1))
 ((0 0 0 1) (0 0 0 0 1))
 ((1 _0 _1 1) (0 1 _0 _1 1))
 ((0 1 _0 1) (0 0 1 _0 1))).
```

Now what is the value of

```
(run^15 (t)
  (fresh (n m)
    (≤l^o n m)
    (≡ (n m) t)))
```

44

```
((() ())
 (() (_0 . _1))
 ((1) (1))
 ((1) (_0 _1 . _2))
 ((_0 1) (_1 1))
 ((_0 1) (_1 _2 _3 . _4))
 ((_0 _1 1) (_2 _3 1))
 ((_0 _1 1) (_2 _3 _4 _5 . _6))
 ((_0 _1 _2 1) (_3 _4 _5 1))
 ((_0 _1 _2 1) (_3 _4 _5 _6 _7 . _8))
 ((_0 _1 _2 _3 1) (_4 _5 _6 _7 1))
 ((_0 _1 _2 _3 1) (_4 _5 _6 _7 _8 _9 . _10))
 ((_0 _1 _2 _3 _4 1) (_5 _6 _7 _8 _9 1))
 ((_0 _1 _2 _3 _4 1) (_5 _6 _7 _8 _9 _10 _11 . _12))
 ((_0 _1 _2 _3 _4 _5 1) (_6 _7 _8 _9 _10 _11 1))).
```

Do these values include all of the values produced in frame 39?

45 Yes.

Here is the definition of $<^o$.

```
(define <o
  (lambda (n m)
    (condi
      ((<lo n m) #s)
      ((=lo n m)
       (fresh (x)
         (poso x)
         (+o n x m)))
      (else #u))))
```

Define $\leqslant^o$ using $<^o$.

46 That is easy.

```
(define ≤o
  (lambda (n m)
    (condi
      ((≡ n m) #s)
      ((<o n m) #s)
      (else #u))))
```

What value is associated with q in

```
(run* (q)
  (<o (1 0 1) (1 1 1))
  (≡ #t q))
```

47 #t,
since five is less than seven.

What is the value of

```
(run* (q)
  (<o (1 1 1) (1 0 1))
  (≡ #t q))
```

48 (),
since seven is not less than five.

What is the value of

```
(run* (q)
  (<o (1 0 1) (1 0 1))
  (≡ #t q))
```

49 (),
since five is not less than five. But if we were to replace $<^o$ with $\leqslant^o$, the value would be (#t).

What is the value of

```
(run6 (n)
  (<o n (1 0 1)))
```

50 (() (0 0 1) (1) ($\_{0}$ 1)),
since ($\_{0}$ 1) represents the numbers two and three.

What is the value of

```
(run6 (m)
  (<o (1 0 1) m))
```

51 (($\_{0}$ $\_{1}$ $\_{2}$ $\_{3}$. $\_{4}$) (0 1 1) (1 1 1)),
since ($\_{0}$ $\_{1}$ $\_{2}$ $\_{3}$. $\_{4}$) represents all the numbers greater than seven.

What is the value of

```
(run* (n)
  (<o n n))
```

52 It has no value,
since $<^o$ calls $<l^o$.

What is the value of

```
(run15 (t)
  (fresh (n m q r)
    (÷o n m q r)
    (≡ (n m q r) t)))
```

53
```
((() (_0 . _1) () ())
 ((1) (1) (1) ())
 ((0 1) (1 1) () (0 1))
 ((0 1) (1) (0 1) ())
 ((1) (_0 _1 . _2) () (1))
 ((_0 1) (_0 1) (1) ())
 ((0 _0 1) (1 _0 1) () (0 _0 1))
 ((0 _0 1) (_0 1) (0 1) ())
 ((_0 1) (_1 _2 _3 . _4) () (_0 1))
 ((1 1) (0 1) (1) (1))
 ((0 0 1) (0 1 1) () (0 0 1))
 ((1 1) (1) (1 1) ())
 ((_0 _1 1) (_2 _3 _4 _5 . _6) () (_0 _1 1))
 ((_0 _1 1) (_0 _1 1) (1) ())
 ((1 0 1) (0 1 1) () (1 0 1))).
```

$\div^o$ divides n by m, producing a quotient q and remainder r.

List all of the values that contain variables.

54
```
((() (_0 . _1) () ())
 ((1) (_0 _1 . _2) () (1))
 ((_0 1) (_0 1) (1) ())
 ((0 _0 1) (1 _0 1) () (0 _0 1))
 ((0 _0 1) (_0 1) (0 1) ())
 ((_0 1) (_1 _2 _3 . _4) () (_0 1))
 ((_0 _1 1) (_2 _3 _4 _5 . _6) () (_0 _1 1))
 ((_0 _1 1) (_0 _1 1) (1) ())).
```

Does the third value `((_0 1) (_0 1) (1) ())` represent two ground values?

55 Yes.
`((_0 1) (_0 1) (1) ())`
represents the two values
`((0 1) (0 1) (1) ())` and
`((1 1) (1 1) (1) ())`.

Do the fourth and fifth values in frame 54 each represent two ground values?

56 Yes.

Does the eighth value in frame 54,

(($\_0$ $\_1$ 1) ($\_0$ $\_1$ 1) (1) ()),

represent four ground values?

57 Yes.

(($\_0$ $\_1$ 1) ($\_0$ $\_1$ 1) (1) ())
represents the four values
((0 0 1) (0 0 1) (1) ()),
((1 0 1) (1 0 1) (1) ()),
((0 1 1) (0 1 1) (1) ()), and
((1 1 1) (1 1 1) (1) ()).

So is (($\_0$ $\_1$ 1) ($\_0$ $\_1$ 1) (1) ()) just shorthand notation?

58 Yes.

Does the first value in frame 54,

(() ($\_0$. $\_1$) () ()),

represent ground values?

59 Yes.

(() ($\_0$. $\_1$) () ())
represents the values
(() (1) () ())
(() (0 1) () ())
(() (1 1) () ())
(() (0 0 1) () ())
(() (1 0 1) () ())
(() (0 1 1) () ())
(() (1 1 1) () ())
(() (0 0 0 1) () ())
(() (1 0 0 1) () ())
(() (0 1 0 1) () ())
(() (1 1 0 1) () ())
(() (0 0 1 1) () ())
(() (1 0 1 1) () ())
...

Is (() ($\_0$. $\_1$) () ()) just shorthand notation?

60 No,

since it is impossible to write every ground value that is represented by
(() ($\_0$. $\_1$) () ()).

Is it possible to write every ground value that is represented by the second, sixth, and seventh values in frame 54?

61 No.

62 How do the first, second, sixth, and seventh values in frame 54 differ from the other values in that frame?

They each contain an improper list whose last *cdr* is a variable.

---

63 Define $\div^o$.

```
(define ÷ᵒ
  (lambda (n m q r)
    (condⁱ
      ((≡ () q) (≡ n r) (<ᵒ n m))
      ((≡ (1) q) (≡ () r) (≡ n m)
       (<ᵒ r m))
      ((<ᵒ m n) (<ᵒ r m)
       (fresh (mq)
         (≤lᵒ mq n)
         (*ᵒ m q mq)
         (+ᵒ mq r n)))
      (else #u))))).
```

---

64 With which three cases do the three **cond**^{i} lines correspond?

The cases in which the dividend n is less than, equal to, or greater than the divisor m, respectively.

---

65 Describe the first **cond**^{i} line.

The first **cond**^{i} line divides a number n by a number m greater than n. Therefore the quotient is zero, and the remainder is equal to n.

---

66 According to the standard definition of division, division by zero is undefined and the remainder r must always be less than the divisor m. Does the first **cond**^{i} line enforce both of these restrictions?

Yes.
The divisor m is greater than the dividend n, which means that m cannot be zero. Also, since m is greater than n and n is equal to r, we know that m is greater than the remainder r. By enforcing the second restriction, we automatically enforce the first.

In the second **cond**^{i} line the dividend and divisor are equal, so the quotient obviously must be one. Why, then, is the $(<^o\ r\ m)$ goal necessary?	67	Because this goal enforces both of the restrictions given in the previous frame.
Describe the first two goals in the third **cond**^{i} line.	68	The goal $(<^o\ m\ n)$ ensures that the divisor is less than the dividend, while the goal $(<^o\ r\ m)$ enforces the restrictions in frame 66.
Describe the last three goals in the third **cond**^{i} line.	69	The last three goals perform division in terms of multiplication and addition. The equation $$\frac{n}{m} = q \text{ with remainder } r$$ can be rewritten as $$n = m \cdot q + r.$$ That is, if mq is the product of m and q, then n is the sum of mq and r. Also, since r cannot be less than zero, mq cannot be greater than n.
Why does the third goal in the last **cond**^{i} line use $\leqslant l^o$ instead of $<^o$	70	Because $\leqslant l^o$ is a more efficient approximation of $<^o$. If mq is less than or equal to n, then certainly the length of the list representing mq cannot exceed the length of the list representing n.

What is the value of
(**run*** (m)
  (**fresh** (r)
    ($\div^o$ (1 0 1) m (1 1 1) r)))
	71	**()**,
		since it fails.

72 Why is () the value of

```
(run* (m)
  (fresh (r)
    (÷ᵒ (1 0 1) m (1 1 1) r)))
```

We are trying to find a number m such that dividing five by m produces seven. Of course, no such m exists.

73 How is () the value of

```
(run* (m)
  (fresh (r)
    (÷ᵒ (1 0 1) m (1 1 1) r)))
```

The third **cond**^{i} line of $\div^o$ ensures that m is less than n when q is greater than one. Therefore $\div^o$ can stop looking for possible values of m when m reaches four.

74 Why do we need the first two **cond**^{i} lines, given that the third **cond**^{i} line seems so general? Why don't we just remove the first two **cond**^{i} lines and remove the $(<^o\ m\ n)$ goal from the third **cond**^{i} line, giving us a simpler definition of $\div^o$

```
(define ÷ᵒ
  (lambda (n m q r)
    (fresh (mq)
      (<ᵒ r m)
      (≤lᵒ mq n)
      (*ᵒ m q mq)
      (+ᵒ mq r n))))
```

Unfortunately, our "improved" definition of $\div^o$ has a problem—the expression

```
(run* (m)
  (fresh (r)
    (÷ᵒ (1 0 1) m (1 1 1) r)))
```

no longer has a value.

75 Why doesn't the expression

```
(run* (m)
  (fresh (r)
    (÷ᵒ (1 0 1) m (1 1 1) r)))
```

have a value when we use the new definition of $\div^o$

Because the new $\div^o$ does not ensure that m is less than n when q is greater than one. Therefore $\div^o$ will never stop trying to find an m such that dividing five by m produces seven.

Hold on! It's going to get subtle!

76 Here is an improved definition of $\div^o$ which is more sophisticated than the ones given in frames 63 and 74. All three definitions implement division with remainder, which means that ($\div^o$ n m q r) satisfies $n = m \cdot q + r$ with $0 \leqslant r < m$.

```
(define ÷ᵒ
  (lambda (n m q r)
    (condⁱ
      ((≡ r n) (≡ () q) (<ᵒ n m))
      ((≡ (1) q) (=lᵒ n m) (+ᵒ r m n)
       (<ᵒ r m))
      (else
        (allⁱ
          (<lᵒ m n)
          (<ᵒ r m)
          (posᵒ q)
          (fresh (n_h n_l q_h q_l qlm qlmr rr r_h)
            (allⁱ
              (splitᵒ n r n_l n_h)
              (splitᵒ q r q_l q_h)
              (condᵉ
                ((≡ () n_h)
                 (≡ () q_h)
                 (−ᵒ n_l r qlm)
                 (*ᵒ q_l m qlm))
                (else
                  (allⁱ
                    (posᵒ n_h)
                    (*ᵒ q_l m qlm)
                    (+ᵒ qlm r qlmr)
                    (−ᵒ qlmr n_l rr)
                    (splitᵒ rr r () r_h)
                    (÷ᵒ n_h m q_h r_h)))))))))))
```

Does the redefined $\div^o$ use any new helper functions?

Yes,
the new $\div^o$ relies on *split*^{o}.

```
(define splitᵒ
  (lambda (n r l h)
    (condⁱ
      ((≡ () n) (≡ () h) (≡ () l))
      ((fresh (b n̂)
         (≡ (0 b . n̂) n)
         (≡ () r)
         (≡ (b . n̂) h)
         (≡ () l)))
      ((fresh (n̂)
         (≡  (1 . n̂) n)
         (≡ () r)
         (≡ n̂ h)
         (≡ (1) l)))
      ((fresh (b n̂ a r̂)
         (≡ (0 b . n̂) n)
         (≡ (a . r̂) r)
         (≡ () l)
         (splitᵒ (b . n̂) r̂ () h)))
      ((fresh (n̂ a r̂)
         (≡ (1 . n̂) n)
         (≡ (a . r̂) r)
         (≡ (1) l)
         (splitᵒ n̂ r̂ () h)))
      ((fresh (b n̂ a r̂ l̂)
         (≡ (b . n̂) n)
         (≡ (a . r̂) r)
         (≡ (b . l̂) l)
         (posᵒ l̂)
         (splitᵒ n̂ r̂ l̂ h)))
      (else #u))))
```

What does $split^o$ do?

77 The call ($split^o$ n () l h) *moves* the lowest bit[†] of n, if any, into l, and moves the remaining bits of n into h; ($split^o$ n (1) l h) moves the two lowest bits of n into l and moves the remaining bits of n into h; and ($split^o$ n (1 1 1 1) l h), ($split^o$ n (0 1 1 1) l h), or ($split^o$ n (0 0 0 1) l h) move the five lowest bits of n into l and move the remaining bits into h; and so on.

[†] The lowest bit of a positive number n is the *car* of n.

What else does $split^o$ do?

78 Since $split^o$ is a relation, it can construct n by combining the lower-order bits of l with the higher-order bits of h, inserting *padding* bits as specified by the length of r.

Why is $split^o$'s definition so complicated?

79 Because $split^o$ must not allow the list (0) to represent a number. For example, ($split^o$ (0 0 1) () () (0 1)) should succeed, but ($split^o$ (0 0 1) () (0) (0 1)) should not.

How does $split^o$ ensure that (0) is not constructed?

80 By removing the rightmost zeros after splitting the number n into its lower-order bits and its higher-order bits.

What is the value of this expression when using the original definition of $\div^o$, as defined in frame 63?

```
(run³ (t)
  (fresh (y z)
    (÷ᵒ (1 0 . y) (0 1) z ())
    (≡ (y z) t)))
```

81 It has no value.

We cannot divide an odd number by two and get a remainder of zero. The old definition of $\div^o$ never stops looking for values of y and z that satisfy the division relation, even though no such values exist. With the latest definition of $\div^o$ as defined in frame 76, however, the expression fails immediately.

82

Here is log^o and its two helper functions.

```
(define logº
  (lambda (n b q r)
    (condⁱ
      ((≡ (1) n) (posº b) (≡ () q) (≡ () r))
      ((≡ () q) (<º n b) (+º r (1) n))
      ((≡ (1) q) (>1º b) (=lº n b) (+º r b n))
      ((≡ (1) b) (posº q) (+º r (1) n))
      ((≡ () b) (posº q) (≡ r n))
      ((≡ (0 1) b)
       (fresh (a ad dd)
         (posº dd)
         (≡ (a ad . dd) n)
         (exp2º n () q)
         (fresh (s)
           (splitº n dd r s))))
      ((fresh (a ad add ddd)
         (condᵉ
           ((≡ (1 1) b))
           (else (≡ (a ad add . ddd) b))))
       (<lº b n)
       (fresh (bw1 bw nw nw1 ql1 ql s)
         (exp2º b () bw1)
         (+º bw1 (1) bw)
         (<lº q n)
         (fresh (q₁ bwq1)
           (+º q (1) q₁)
           (*º bw q₁ bwq1)
           (<º nw1 bwq1))
           (exp2º n () nw1)
           (+º nw1 (1) nw)
           (÷º nw bw ql1 s)
           (+º ql (1) ql1)
         (condᵉ
           ((≡ q ql))
           (else (<lº ql q)))
         (fresh (bql qh s qdh qd)
           (repeated-mulº b ql bql)
           (÷º nw bw1 qh s)
           (+º ql qdh qh)
           (+º ql qd q)
           (condᵉ
             ((≡ qd qdh))
             (else (<º qd qdh)))
           (fresh (bqd bq1 bq)
             (repeated-mulº b qd bqd)
             (*º bql bqd bq)
             (*º b bq bq1)
             (+º bq r n)
             (<º n bq1)))))
      (else #u))))
```

```
(define exp2º
  (lambda (n b q)
    (condⁱ
      ((≡ (1) n) (≡ () q))
      ((>1º n) (≡ (1) q)
       (fresh (s)
         (splitº n b s (1))))
      ((fresh (q₁ b₂)
         (allⁱ
           (≡ (0 . q₁) q)
           (posº q₁)
           (<lº b n)
           (appendº b (1 . b) b₂)
           (exp2º n b₂ q₁))))
      ((fresh (q₁ nh b₂ s)
          (allⁱ
            (≡ (1 . q₁) q)
            (posº q₁)
            (posº nh)
            (splitº n b s nh)
            (appendº b (1 . b) b₂)
            (exp2º nh b₂ q₁))))
      (else #u))))
```

```
(define repeated-mulº
  (lambda (n q nq)
    (condᵉ
      ((posº n) (≡ () q) (≡ (1) nq))
      ((≡ (1) q) (≡ n nq))
      ((>1º q)
       (fresh (q₁ nq1)
         (+º q₁ (1) q)
         (repeated-mulº n q₁ nq1)
         (*º nq1 n nq)))
      (else #u))))
```

Guess what log^o does?

83 It builds a split-rail fence.

Not quite. Try again.

84 It implements the logarithm relation: (log^o n b q r) holds if $n = b^q + r$.

Are there any other conditions that the logarithm relation must satisfy?

85 There had better be!
Otherwise, the relation would always hold if $q = 0$ and $r = n - 1$, regardless of the value of b.

Give the complete logarithm relation.

86 (log^o n b q r) holds if $n = b^q + r$, where $0 \leq r$ and q is the largest number that satisfies the relation.

Does the logarithm relation look familiar?

87 Yes.
The logarithm relation is similar to the division relation, but with exponentiation in place of multiplication.

In which ways are log^o and $\div^o$ similar?

88 Both log^o and $\div^o$ are relations that take four arguments, each of which can be fresh variables. The $\div^o$ relation can be used to define addition, multiplication, and subtraction. The log^o relation is equally flexible, and can be used to define exponentiation, to determine exact discrete logarithms, and even to determine discrete logarithms with a *remainder*. The log^o relation can also find the base b that corresponds to a given n and q.

What value is associated with r in

```
(run* (r)
  (logo (0 1 1 1) (0 1) (1 1) r))
```

89 `(0 1 1)`,
since $14 = 2^3 + 6$.

What is the value of

(**run**^{8} (s)
  (**fresh** $(b\ q\ r)$
    (log^o (0 0 1 0 0 0 1) $b\ q\ r$)
    ($>$**1**^{o} q)
    ($\equiv$ ($b\ q\ r$) s)))

90 (((1) ($\_{0}$ $\_{1}$. $\_{2}$) (1 1 0 0 0 0 1))
 (() ($\_{0}$ $\_{1}$. $\_{2}$) (0 0 1 0 0 0 1))
 ((0 1) (0 1 1) (0 0 1))
 ((0 0 1) (1 1) (0 0 1))
 ((1 0 1) (0 1) (1 1 0 1 0 1))
 ((0 1 1) (0 1) (0 0 0 0 0 1))
 ((1 1 1) (0 1) (1 1 0 0 1))
 ((0 0 0 1) (0 1) (0 0 1))),
since
$68 = 1^n + 67$ where n is greater than one,
$68 = 0^n + 68$ where n is greater than one,
$68 = 2^6 + 4$,
$68 = 4^3 + 4$,
$68 = 5^2 + 43$,
$68 = 6^2 + 32$,
$68 = 7^2 + 19$, and
$68 = 8^2 + 4$.

Define exp^o using log^o.

91 That's easy.

```
(define expᵒ
  (lambda (b q n)
    (logᵒ n b q ())))
```

What value is associated with t in

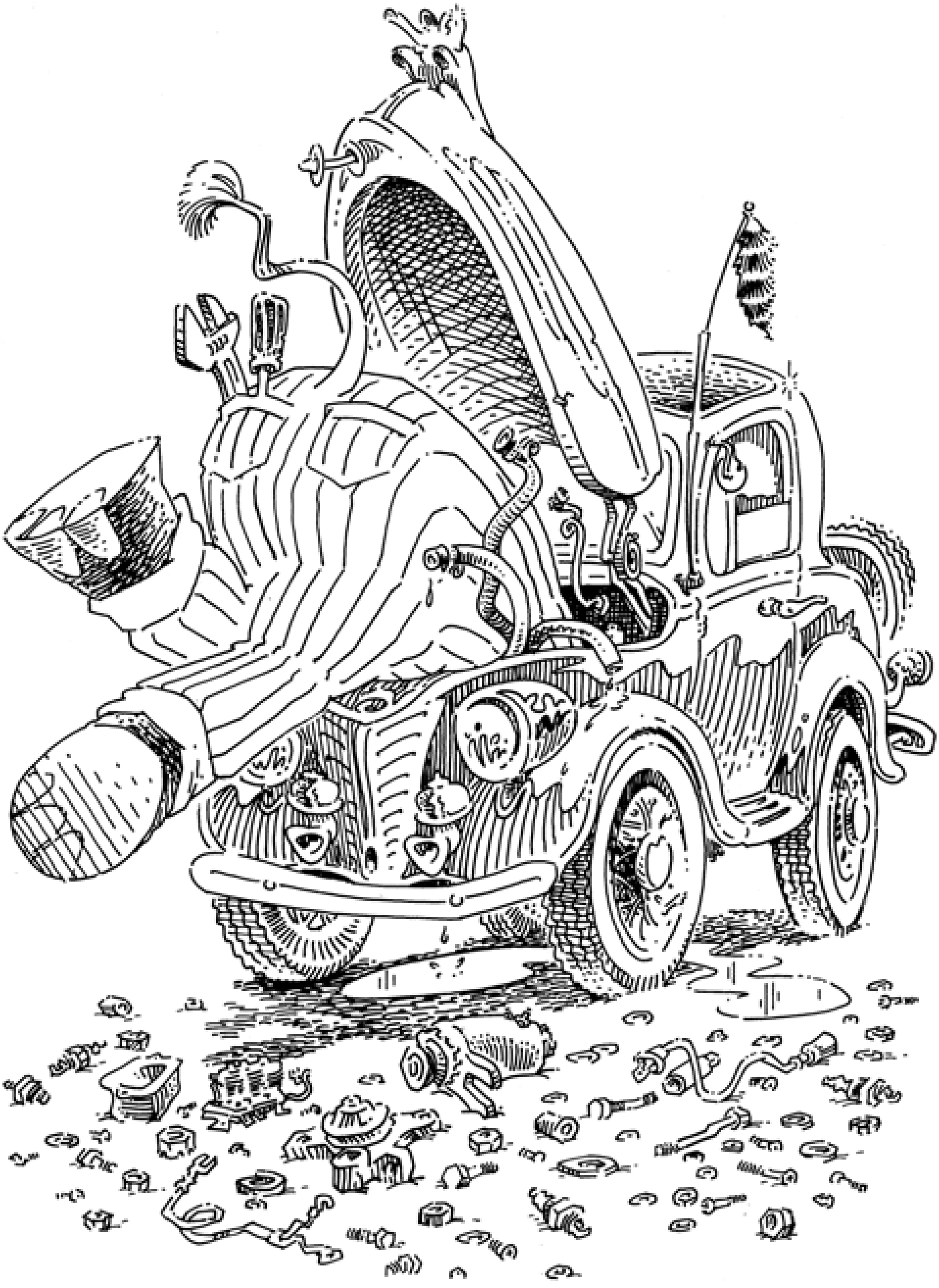
(**run*** (t)
  (exp^o (1 1) (1 0 1) t))

92 (1 1 0 0 1 1 1 1),
which is the same as (*build-num* 243).

⟹ Time for a banquet; you've earned it. ⟸

THIS IS A NAPKIN!

9.
Under the Hood



What is the essence of our style of definitions?

1 **cond**^{e} and **cond**^{i}?

No. Their job is to manage the order of values. Try again.

2 How about car^o, cdr^o, $cons^o$, $null^o$, eq^o, and $pair^o$?

Not quite, but closer. One more try.

3 Well, each of those six definitions rely on $\equiv$, so it must be the essence.

But, what about `#s` and `#u`

4 They too are simple goals, but $\equiv$ is the simplest goal that can succeed for some values and fail for others.

Yes. The definition of $\equiv$ relies on *unify*, which we are about to discuss.

5 Okay, let's begin.

Here are three variables u, v, and w.†

```
(define u (var u))

(define v (var v))

(define w (var w))
```

Define the variables x, y, and z.

6 That's easy.

```
(define x (var x))†

(define y (var y))

(define z (var z))
```

† Each invocation of *var* (*var* is implemented as *vector*) is given a symbol and creates a different fresh variable (a nonempty vector). *var?* (*var?* is implemented as *vector?*) determines if its argument had been created by *var*. Prolog's anonymous variable (see page 2 of William F. Clocksin. *Clause and Effect*. Springer, 1997.) can be defined as an identifier macro that expands to (*var* (**quote** _)). For discussion of identifier macros, see pages 193 and 204 of R. Kent Dybvig. *The Scheme Programming Language third ed.* MIT Press, 2003; and pages 47 and 48 of Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. Building little languages with macros. *Dr. Dobb's Journal*. April, 2004.

† As a reminder, (**define** x (*var* x)) is written as (**define** x (*var* (**quote** x))).

What is

$(z \,.\, \mathsf{a})$

7 It is our way of representing an *association.* The *lhs* (left-hand side) of an association *must* be a variable. The *rhs* (right-hand side) of an association may be any value.†

† *lhs* is *car* and *rhs* is *cdr.*

What is the value of

(*rhs* $(z \,.\, \mathsf{b})$)

8 b.

What is the value of

(*rhs* $(z \,.\, w)$)

9 The variable w.

What is the value of

(*rhs* $(z \,.\, (x\ \mathsf{e}\ y))$)

10 The list $(x\ \mathsf{e}\ y)$.

What is

$((z \,.\, \mathsf{a})\ (x \,.\, w)\ (y \,.\, z))$

11 It is our way of representing a *substitution*†, a list of associations.

† Most of this chapter is about substitutions and unification. Our *unify* is inspired by Franz Baader and Wayne Snyder. "Unification theory," Chapter 8 of *Handbook of Automated Reasoning*, edited by John Alan Robinson and Andrei Voronkov. Elsevier Science and MIT Press, 2001.

Is

$((z \,.\, \mathsf{a})\ (x \,.\, x)\ (y \,.\, z))$

a substitution?

12 Not for us,
since we do *not* permit associations like $(x \,.\, x)$ in which its *lhs* is the same as its *rhs*.

Here is *empty-s*.

(**define** *empty-s* ())

What does it represent?

13 It represents a substitution that does not contain any associations.

What is the value of $(walk\ z\ ((z\ .\ \mathsf{a})\ (x\ .\ w)\ (y\ .\ z)))$	[14] a, because we walk from z to the *rhs* of its association, which is a.
What is the value of $(walk\ y\ ((z\ .\ \mathsf{a})\ (x\ .\ w)\ (y\ .\ z)))$	[15] a, because we walk from y to the *rhs* of its association, which is z, and we walk from z to the *rhs* of its association, which is a.
What is the value of $(walk\ x\ ((z\ .\ \mathsf{a})\ (x\ .\ w)\ (y\ .\ z)))$	[16] The fresh variable w, because we walk from x to the *rhs* of its association, which is w.
What is the value of $(walk\ w\ ((z\ .\ \mathsf{a})\ (x\ .\ w)\ (y\ .\ z)))$	[17] The fresh variable w, because w is not the *lhs* of any association.
What is the value of $(walk\ x\ ((x\ .\ y)\ (z\ .\ x)\ (y\ .\ z)))$	[18] It has no value, because we never stop walking. The substitution $((x\ .\ y)\ (z\ .\ x)\ (y\ .\ z))$ is said to be *circular*.
What is the value of $(walk\ w\ ((x\ .\ y)\ (w\ .\ \mathsf{b})\ (z\ .\ x)\ (y\ .\ z)))$	[19] b, The substitution $((x\ .\ y)\ (w\ .\ \mathsf{b})\ (z\ .\ x)\ (y\ .\ z))$ is circular, but the walk still terminates.
If x is a variable and s is a substitution, does $(walk\ x\ s)$ always have a value?	[20] No. See frame 18.
If x is a variable and s is a substitution built by *unify*, does $(walk\ x\ s)$ always have a value?	[21] Yes.

If a walk has a value, what are the two possible types of values that it might have?

22 It could be a fresh variable, or it could be a value that is *not* a variable.

The value of the expression below is b. What are the values of the walks of u, v, and w

(*walk* x
  ((y . b) (x . y) (v . x) (w . x) (u . w)))

23 They are also b,
since each walk leads to x, and we know where x leads.

The value of the expression below is z. What are the values of the walks of u, v, and w

(*walk* x
  ((y . z) (x . y) (v . x) (w . x) (u . w)))

24 They are also z,
since each walk leads to x, and we know where x leads.

What is the value of

(*walk* u ((x . b) (w . (x e x)) (u . w)))

25 The list (x e x).

Why isn't (b e b) the value?

26 The job of *walk* is to find either a fresh variable or a non-variable value. In this case it finds the list (x e x), which is a non-variable value. This is a subtle point, because one could imagine a different version of *walk* (frame 47) that would return the value (b e b).

Here is the definition of *walk*.

```
(define walk
  (lambda (v s)
    (cond
      ((var? v)
       (cond
         ((assq v s) ⇒†
          (lambda (a)
            (walk (rhs a) s)))
         (else v)))
      (else v))))
```

Is *walk* recursive?

27 Yes.

† These two expressions are equivalent.

```
(cond                      (let ((a (assq v s)))
  ((assq v s) ⇒ f)           (cond
  (else v))                    (a (f a))
                               (else v)))
```

walk relies on *assq*, a standard function that we use to find an association in a substitution.

We could revise **fresh** (and **run**) so that it would mark (by associating a variable with itself—see frame 12) where in the substitution we have introduced a fresh variable. We could then recognize this pair with (*eq?* (*lhs* *a*) (*rhs* *a*)), thus avoiding a recursion and the need to search the entire substitution.

When does the recursion happen?

[28] When v is the *lhs* of an association in s.

Consider the definition of *ext-s*, which extends a substitution.

```
(define ext-s
  (lambda (x v s)
    (cons (x . v) s)))
```

What is the value of

(*walk* x (*ext-s* x y ((z . x) (y . z))))

[29] It has no value, since the extended substitution is the same as the one in frame 18.

What is the value of

(*walk* y ((x . e)))

[30] The fresh variable y.

What is the value of

(*walk* y (*ext-s* y x ((x . e))))

[31] e.

What is the value of

(*walk* x ((y . z) (x . y)))

[32] The fresh variable z.

What is the value of

(*walk* x (*ext-s* z b ((y . z) (x . y))))

[33] b.

What is the value of

(*walk* x (*ext-s* z w ((y . z) (x . y))))

[34] w.

What is the value of

(*unify* v w s)

for all values v and w and for all substitutions s

[35] It is either #f or it is a new substitution. The new substitution includes the associations of s and perhaps other associations.

Here is *unify*.

```
(define unify
  (lambda (v w s)
    (let ((v (walk v s))
          (w (walk w s)))
      (cond
        ((eq? v w) s)
        ((var? v) (ext-s v w s))
        ((var? w) (ext-s w v s))
        ((and (pair? v) (pair? w))
         (cond
           ((unify (car v) (car w) s) ⇒
            (lambda (s)
              (unify (cdr v) (cdr w) s)))
           (else #f)))
        ((equal? v w) s)
        (else #f)))))
```

What is the first thing that happens in *unify*

36 We **let**-bind v (and w) to a possibly different value. Thus, we know that the new binding of v (and w) is either to a fresh variable or to a non-variable value.†

† Our very simple representation of variables (frame 6) makes it unsafe to pass vectors, other than variables, as the first two arguments of *unify*. We could, however, define variables in many other ways, but it would unnecessarily complicate the definitions of *var* and *var?*. Nevertheless, the reader should not hesitate to experiment with refined definitions of *var* and *var?*.

What is a simple way to improve *unify*

37 We could determine if v is the same as w before **let**-binding v and w.

What is another way to improve *unify*

38 If we have two variables, we can walk one of them, but while it is being walked, we can see if we meet the other. Then, we know that the two variables unify. This generalizes the improvement in the previous frame.

What is the purpose of the *eq?* test?†

† We are using *eq?* primarily for comparing two fresh variables, but we also benefit from the *eq?* test on some non-variable values. Furthermore, although we use no effects, our definitions are not purely functional, since we rely on *eq?* to distinguish two variables (nonempty vectors) that were created at different times. This effect, however, could be avoided by including a *birthdate* variable in the substitution. Each time we would create variables, we would then extend the substitution with *birthdate* and the associated value of *birthdate* appropriately incremented.

39 If v and w are the same, we do not extend the substitution. Conveniently, this works whether or not v and w are fresh variables.

Explain why the next **cond** line uses *var?*

40 Because if v is a variable it must be fresh[†], since it has been walked.

† This behavior is necessary in order for $\equiv$ to satisfy "The Law of Fresh."

And what about the next **cond** line?

41 Because if w is a variable it must be fresh, since it has been walked.[†]

† The answer of this **cond** line could be replaced by (*unify* w v s), because for a value w and a substitution s, (*walk* (*walk* w s) s) = (*walk* w s).

What happens when both v and w are pairs?

42 We unify the *car* of v with the *car* of w. If they successfully unify, we get a new substitution, which we then use to unify the *cdr* of v with the *cdr* of w.

What is the purpose of the ((*equal?* v w) s) **cond** line?

43 This one is easy. If either v or w is a pair, and the other is not, then clearly no substitution exists that can make them equal. Also, the *equal?* works for other kinds of values.

What is the value of

(*walk** x
  ((y . (a z c)) (x . y) (z . a)))

44 (a a c).
The walked value of x is (a z c). Then the walk*ed values of each value in the list are used to create a new list.

What is the value of

(*walk** x
  ((y . (z w c)) (x . y) (z . a)))

45 (a w c).
The walked value of x is (z w c). Then the walk*ed values of each value in the list are used to create a new list.

What is the value of

(*walk** *y*
  ((*y* . (*w* *z* c)) (*v* . b) (*x* . *v*) (*z* . *x*)))

46 (*w* b c).
The walked value of *y* is (*w* *z* c). Then the walk*ed values of each value in the list are used to create a new list.

Here is *walk**.

```
(define walk*
  (lambda (v s)
    (let ((v (walk v s)))
      (cond
        ((var? v) v)
        ((pair? v)
         (cons
           (walk* (car v) s)
           (walk* (cdr v) s)))
        (else v)))))
```

Is *walk** recursive?

47 Yes, and it's also useful.†

† Here is **project** (pronounced "proh-ject").

```
(define-syntax project
  (syntax-rules ()
    ((_ (x ...) g ...)
     (λG (s)
       (let ((x (walk* x s)) ...)
         ((all g ...) s))))))
```

where λ_G (see appendix) is just **lambda**. **project** is syntactically like **fresh**, but it binds diffferent values to the lexical variables. **project** binds walk*ed values, whereas **fresh** binds variables using *var*. For example, the value of

```
(run* (q)
  (≡ #f q)
  (project (q)
    (≡ (not (not q)) q)))
```

is (#f); without projecting *q*, its value would be (), since *q*, which is represented using a vector (frame 6), is considered to be *non*false when passed as an argument to *not*.

How does *walk** differ from *walk* if its first argument is a fresh variable?

48 It doesn't.
If *v* is a fresh variable, then only the first **cond** line of *walk** is ever considered. Thus *walk* and *walk** behave the same if *v* is fresh.

How does *walk** differ from *walk* if its first argument is a nonfresh variable?

49 If its first argument is nonfresh, then the second **cond** line of *walk** must be considered. Then, if the walked *v* is a pair, *walk** constructs a new pair of the *walk** of each value in *v*, whereas the walked value is just *v*. Finally, if the walked value is not a pair, then *walk* and *walk** behave the same.

What property holds with a variable that has been walked?

50 We know that if the walked variable is itself a variable, then it must be fresh.

What property holds with a value that has been walk*ed?

51 We know that any variable that appears in the resultant value must be fresh.

Here is the definition of *reify-s*, whose first argument is assumed to have been walk*ed and whose second argument starts out as *empty-s*. The result of an invocation of *reify-s* is called a *reified-name* substitution.

```
(define reify-s
  (lambda (v s)
    (let ((v (walk v s)))
      (cond
        ((var? v)
         (ext-s v (reify-name (size-s s)) s))
        ((pair? v) (reify-s (cdr v)
                     (reify-s (car v) s)))
        (else s)))))
```

Describe (*reify-s v empty-s*).

52 (*reify-s v empty-s*) returns a reified-name substitution in which each variable in *v* is associated with its reified name.†

† Here is *reify-name*.

```
(define reify-name
  (lambda (n)
    (string→symbol
      (string-append "_" "." (number→string n)))))
```

The functions *string→symbol*, *string-append*, and *number→string* are standard; and *size-s* is *length*, which is also standard.

What is the value of

```
(let ((r (w x y)))
  (walk* r (reify-s r empty-s)))
```

53 ($\_{0}$ $\_{1}$ $\_{2}$).

What is the value of

```
(let ((r (walk* (x y z) empty-s)))
  (walk* r (reify-s r empty-s)))
```

54 ($\_{0}$ $\_{1}$ $\_{2}$).

What is the value of

```
(let ((r (u (v (w x) y) x)))
  (walk* r (reify-s r empty-s)))
```

55 ($\_{0}$ ($\_{1}$ ($\_{2}$ $\_{3}$) $\_{4}$) $\_{3}$).

What is the value of

```
(let ((s ((y . (z w c w)) (x . y) (z . a))))
  (let ((r (walk* x s)))
    (walk* r (reify-s r empty-s))))
```

56 (a $\_{0}$ c $\_{0}$),
since *r*'s fresh variable *w* is replaced by the reified name $\_{0}$ (see frame 45).

[57] If every nonfresh variable has been removed from a value and every fresh variable has been replaced by a reified name, what do we know?

We know that there are no variables in the resultant value.

[58] Consider the definition of *reify*, where it is assumed that its only argument has been walk*ed.

```
(define reify
  (lambda (v)
    (walk* v (reify-s v empty-s))))
```

What is the value of

```
(let ((s ((y . (z w c w)) (x . y) (z . a))))
  (reify (walk* x s)))
```

(a $\_{0}$ c $\_{0}$),
since this is just a restatement of frame 56. Within **run**, (*reify* (*walk** *x* *s*)) transforms the value associated with *x* by first removing all nonfresh variables. This is done by (*walk** *x* *s*), which returns a value whose variables are fresh. The call to *reify* then transforms the walk*ed value, replacing each fresh variable with its reified name.

[59] Here are *ext-s*√, a new way to extend a substitution, and *occurs*√, which it uses.

```
(define ext-s√
  (lambda (x v s)
    (cond
      ((occurs√ x v s) #f)
      (else (ext-s x v s)))))

(define occurs√
  (lambda (x v s)
    (let ((v (walk v s)))
      (cond
        ((var? v) (eq? v x))
        ((pair? v)
         (or
           (occurs√ x (car v) s)
           (occurs√ x (cdr v) s)))
        (else #f)))))
```

Where might we want to use *ext-s*√

We use *ext-s*√ where we used *ext-s* in *unify*, so here is the definition of *unify*√.

```
(define unify√
  (lambda (v w s)
    (let ((v (walk v s))
          (w (walk w s)))
      (cond
        ((eq? v w) s)
        ((var? v) (ext-s√ v w s))
        ((var? w) (ext-s√ w v s))
        ((and (pair? v) (pair? w))
         (cond
           ((unify√ (car v) (car w) s) ⇒
            (lambda (s)
              (unify√ (cdr v) (cdr w) s)))
           (else #f)))
        ((equal? v w) s)
        (else #f)))))
```

60 Why might we want to use *ext-s*$^{\surd}$

Because we might want to avoid creating a circular substitution that if passed to *walk** might lead to no value.

61 What is the value of

```
(run¹ (x)
  (≡ (x) x))
```

It has no value.

62 What is the value of

```
(run¹ (q)
  (fresh (x)
    (≡ (x) x)
    (≡ #t q)))
```

(#t).
Although the substitution is circular, x is not reached by the *walk** of q from within **run**.

63 What is the value of

```
(run¹ (q)
  (fresh (x y)
    (≡ (x) y)
    (≡ (y) x)
    (≡ #t q)))
```

(#t).
Although the substitution is circular, neither x nor y is reached by the *walk** of q from within **run**.

64 What is the value of

```
(run¹ (x)
  (≡✓ (x) x))
```

(),
where ≡$^{\surd}$ is the same as ≡, except that it relies on *unify*$^{\surd}$ instead of *unify*.†

† Here is ≡$^{\surd}$.

```
(define ≡✓
  (lambda (v w)
    (λG (s)
      (cond
        ((unify✓ v w s) ⇒ #s)
        (else (#u s))))))
```

where #s and #u are defined in the appendix, and λ_{G} is just **lambda**.

What is the value of

```
(run¹ (x)
  (fresh (y z)
    (≡ x z)
    (≡ (a b z) y)
    (≡ x y)))
```

65 It has no value.

What is the value of

```
(run¹ (x)
  (fresh (y z)
    (≡ x z)
    (≡ (a b z) y)
    (≡✓ x y)))
```

66 ().

What is the substitution when (≡✓ *x y*) fails in the previous frame?

67 ((*y* . (a b *z*)) (*z* . *x*)).
(≡✓ *x y*) fails because (*occurs*✓ *x y* ((*y* . (a b *z*)) (*z* . *x*))) returns #t. *occurs*✓ first finds *y*'s association, (a b *z*). *occurs*✓ then searches (a b *z*) and at each step makes sure that the *rhs* is walked if it is a variable. When *z* is walked, *walk* returns the fresh variable *x*, which means that we have an *occurrence* of *x* in *y*.

When should we use ≡✓

68 When we want to avoid creating a circular (frame 61) substitution.

So, why indeed does

```
(run¹ (x)
  (≡ (x) x))
```

have no value?

69 It has no value because **run** uses *walk** (see frame 58) on *x* and the circular substitution. This call of *walk**, however, has no value.

What is the substitution generated by (**run^1** (x) $\quad(\equiv$ **(**x**)** $x))$	70 **(**$($x$ **.** **(**x**)**$)$**)**, which is a circular substitution.

$\Longrightarrow$ The end, sort of. Time for vacation. $\Longleftarrow$

This space reserved for

PALM TREES!

10.
Thin Ice

Does

(**cond**^{a}
  (#u #s)
  (**else** #u))

succeed?†

† **cond**^{a} is written `conda` and is pronounced "con-day". **cond**^{a} is like the so-called *soft-cut* (also known as *if-then-else*) and is described on page 45 of William F. Clocksin. *Clause and Effect*. Springer, 1997.

1 No,
because the question of the first **cond**^{a} line is the goal #u, so **cond**^{a} tries the second line.

Does

(**cond**^{a}
  (#u #s)
  (**else** #s))

succeed?

2 Yes,
because the question of the first **cond**^{a} line is the goal #u, so **cond**^{a} tries the second line.

Does

(**cond**^{a}
  (#s #u)
  (**else** #s))

succeed?

3 No,
because the question of the first **cond**^{a} line is the goal #s, so **cond**^{a} tries the answer of the first line.

Does

(**cond**^{a}
  (#s #s)
  (**else** #u))

succeed?

4 Yes,
because the question of the first **cond**^{a} line is the goal #s, so **cond**^{a} tries the answer of the first line.

What is the value of

(**run*** (x)
  (**cond**^{a}
    (($\equiv$ olive x) #s)
    (($\equiv$ oil x) #s)
    (**else** #u)))

5 (olive),
because ($\equiv$ olive x) succeeds; therefore, the answer of the first **cond**^{a} line is #s. The #s preserves the association of x to olive.

The Law of cond^a

If the question of a cond^a line succeeds, pretend that the remaining cond^a lines have been replaced by a single (else #u).

What does the "*a*" stand for in **cond**^{a}

6 It stands for *a single line*, since at most a single line can succeed.

What is the value of

```
(run* (x)
  (conda
    ((≡ virgin x) #u)
    ((≡ olive x) #s)
    ((≡ oil x) #s)
    (else #u)))
```

7 **()**,
because (≡ virgin x) succeeds, but the answer of the first **cond**^{a} line fails. We cannot pretend that (≡ virgin x) fails because we are within neither a **cond**^{e} nor a **cond**^{i}.

What is the value of

```
(run* (q)
  (fresh (x y)
    (≡ split x)
    (≡ pea y)
    (conda
      ((≡ split x) (≡ x y))
      (else #s)))
  (≡ #t q))
```

8 **()**.
(≡ split x) succeeds, since x is already associated with split. (≡ x y) fails, however, since x and y are associated with different values.

What value is associated with q in

```
(run* (q)
  (fresh (x y)
    (≡ split x)
    (≡ pea y)
    (conda
      ((≡ x y) (≡ split x))
      (else #s)))
  (≡ #t q))
```

9 #t.
(≡ x y) fails, since x and y are associated with different values. The question of the first **cond**^{a} line fails, therefore we try the second **cond**^{a} line, which succeeds.

Why does the value change when we switch the order of (≡ split x) and (≡ x y) within the first **cond**^{a} line?

10 Because only if the question of a **cond**^{a} line fails do we consider the remaining **cond**^{a} lines. If the question succeeds, it is as if the remaining **cond**^{a} lines have been replaced by a single (**else** #u).

Consider the definition of *not-pasta*^{o}.

```
(define not-pastaᵒ
  (lambda (x)
    (condᵃ
      ((≡ pasta x) #u)
      (else #s))))
```

What is the value of

```
(run* (x)
  (condᵃ
    ((not-pastaᵒ x) #u)
    (else (≡ spaghetti x))))
```

11 (spaghetti),
because x starts out fresh, but the question (*not-pasta*^{o} x) associates x with pasta, but then fails. Since (*not-pasta*^{o} x) fails, we try (≡ spaghetti x).

Then, what is the value of

```
(run* (x)
  (≡ spaghetti x)
  (condᵃ
    ((not-pastaᵒ x) #u)
    (else (≡ spaghetti x))))
```

12 (),
because (*not-pasta*^{o} x) succeeds, which shows the risks involved when using **cond**^{a}. We can't allow a fresh variable to become nonfresh as part of a **cond**^{a} question.

The Third Commandment

If prior to determining the question of a cond^a line a variable is fresh, it must remain fresh in the question of that line.

What is the value of

(**run*** (q)
  (**cond**^{a}
    ($always^o$ #s)
    (**else** #u))
  ($\equiv$ #t q))

13 It has no value,
since **run*** never finishes building the list of #t's.

What is the value of[†]

(**run*** (q)
  (**cond**^{u}
    ($always^o$ #s)
    (**else** #u))
  ($\equiv$ #t q))

14 **(**#t**)**,
because **cond**^{u} is like **cond**^{a}, except that the successful question, here $always^o$, succeeds only once.

† **cond**^{u} is written `condu` and is pronounced "cond-you". **cond**^{u} corresponds to committed-choice of Mercury (so-called "once"), which is described in Fergus Henderson, Thomas Conway, Zoltan Somogyi, and David Jeffery. "The Mercury language reference manual." University of Melbourne Technical Report 96/10, 1996. Mercury was the first language to effectively combine and extensively use soft-cuts (frame 1) and committed choice, avoiding the *cut* of Prolog. See Lee Naish. "Pruning in logic programming." University of Melbourne Technical Report 95/16, 1995.

What is the value of

(**run*** (q)
  (**cond**^{u}
    (#s $always^o$)
    (**else** #u))
  ($\equiv$ #t q))

15 It has no value,
since **run*** never finishes building the list of #t's.

What does the "u" stand for in **cond**^{u}

16 It stands for *uni-*, because the successful *question* of a **cond**^{u} line succeeds only once.

What is the value of

```
(run¹ (q)
  (condᵃ
    (alwaysᵒ #s)
    (else #u))
  #u
  (≡ #t q))
```

17 It has no value, since *always*^{o} keeps succeeding after the outer #u fails.

What is the value of

```
(run¹ (q)
  (condᵘ
    (alwaysᵒ #s)
    (else #u))
  #u
  (≡ #t q))
```

18 **()**,
because **cond**^{u}'s successful question succeeds only once.

The Law of cond^u

cond^u behaves like cond^a, except that a successful question succeeds only once.

Here is *once*^{o}.

```
(define onceᵒ
  (lambda (g)
    (condᵘ
      (g #s)
      (else #u))))
```

What is the value of

```
(run* (x)
  (onceᵒ (teacupᵒ x)))
```

19 (tea).
The first **cond**^{e} line of *teacup*^{o} succeeds. Since *once*^{o}'s goal can succeed only once, there are no more values. But, this breaks **The Third Commandment**.

What is the value of

```
(run¹ (q)
  (onceᵒ (salᵒ neverᵒ))
  #u)
```

20 **()**.
The first **cond**^{e} line of *sal*^{o} succeeds. This is followed by #u, which fails. Since *once*^{o}'s goal can succeed only once, this avoids *never*^{o}, so the **run** fails. This use of *once*^{o} obeys **The Third Commandment**.

What is the value of

```
(run* (r)
  (condᵉ
    ((teacupᵒ r) #s)
    ((≡ #f r) #s)
    (else #u)))
```

21 **(**tea cup #f**)**.

What is the value of

```
(run* (r)
  (condᵃ
    ((teacupᵒ r) #s)
    ((≡ #f r) #s)
    (else #u)))
```

22 **(**tea cup**)**,
breaking **The Third Commandment**.

And, what is the value of

```
(run* (r)
  (≡ #f r)
  (condᵃ
    ((teacupᵒ r) #s)
    ((≡ #f r) #s)
    (else #u)))
```

23 **(**#f**)**,
since this value is included in frame 21.

What is the value of

```
(run* (r)
  (≡ #f r)
  (condᵘ
    ((teacupᵒ r) #s)
    ((≡ #f r) #s)
    (else #u)))
```

24 **(**#f**)**.
cond^a and **cond**^{u} often lead to fewer values than a similar expression that uses **cond**^{e}. Knowing that helps determine whether to use **cond**^{a} or **cond**^{u}, or the more general **cond**^{e} or **cond**^{i}.

Let's do a bit more arithmetic.

25 Okay.

Here is *bump*^{o}.

```
(define bumpᵒ
  (lambda (n x)
    (condᵉ
      ((≡ n x) #s)
      (else
        (fresh (m)
          (−ᵒ n (1) m)
          (bumpᵒ m x))))))
```

What is the value of

```
(run* (x)
  (bumpᵒ (1 1 1) x))
```

26
```
((1 1 1)
 (0 1 1)
 (1 0 1)
 (0 0 1)
 (1 1)
 (0 1)
 (1)
 ()).
```

Here is *gen&test*^{o}.

```
(define gen&testᵒ
  (lambda (op i j k)
    (onceᵒ
      (fresh (x y z)
        (op x y z)
        (≡ i x)
        (≡ j y)
        (≡ k z)))))
```

What value is associated with q in

```
(run* (q)
  (gen&testᵒ +ᵒ (0 0 1) (1 1) (1 1 1))
  (≡ #t q))
```

27 `#t`,
because four plus three is seven, but there is more.

What values are associated with x, y, and z after the call to (op x y z), where op is $+^o$

28 $\_{0}$, `()`, and $\_{0}$, respectively.

What happens next?

29 $(\equiv\ i\ x)$ succeeds,
since i is associated with (0 0 1) and x is fresh. As a result, x is associated with (0 0 1).

What happens after $(\equiv\ i\ x)$ succeeds?

30 $(\equiv\ j\ y)$ fails,
since j is associated with (1 1) and y is associated with ().

What happens after $(\equiv\ j\ y)$ fails?

31 $(op\ x\ y\ z)$ is tried again, and this time associates x with (), and both y and z with $(\_{0}\ .\ \_{1})$.

What happens next?

32 $(\equiv\ i\ x)$ fails,
since i is still associated with (0 0 1) and x is associated with ().

What happens after $(\equiv\ i\ x)$ fails?

33 $(op\ x\ y\ z)$ is tried again and this time associates both x and y with (1), and z with (0 1).

What happens next?

34 $(\equiv\ i\ x)$ fails,
since i is still associated with (0 0 1) and x is associated with (1).

What happens the eighty-second time that $(op\ x\ y\ z)$ is called?

35 $(op\ x\ y\ z)$ associates both x and z with $(0\ 0\ \_{0}\ .\ \_{1})$, and y with (1 1).

What happens next?

36 $(\equiv\ i\ x)$ succeeds,
associating x, and therefore z, with (0 0 1).

What happens after ($\equiv$ i x) succeeds?

[37] ($\equiv$ j y) succeeds, since both j and y are associated with (1 1).

What happens after ($\equiv$ j y) succeeds?

[38] ($\equiv$ k z) succeeds, since both k and z are associated with (0 0 1).

What values are associated with x, y, and z after the call to (op x y z) is made in the body of $gen\&test^o$

[39] x, y, and z are not associated with any values, since they are fresh.

What is the value of

(**run¹** (q)
  ($gen\&test^o$ $+^o$ (0 0 1) (1 1) (0 1 1)))

[40] It has no value.

Can (op x y z) fail when x, y, and z are fresh?

[41] Never.

Why doesn't

(**run¹** (q)
  ($gen\&test^o$ $+^o$ (0 0 1) (1 1) (0 1 1)))

have a value?

[42] (op x y z) *gen*erates various associations for x y, and z, and then *test*s ($\equiv$ i x), ($\equiv$ j y), and ($\equiv$ k z) if the given triple of values i, j, and k is present among the generated triple x, y, and z. All the generated triples x, y, and z satisfy, by definition, the relation op, $+^o$ in our case. If the triple of values i, j, and k is so chosen that $i + j$ is not equal to k, and our definition of $+^o$ is correct, then that triple of values cannot be found among those generated by $+^o$. (op x y z) will continue to generate associations, and the tests ($\equiv$ i x), ($\equiv$ j y), and ($\equiv$ k z) will continue to reject them. So this **run¹** expression will have no value.

Here is *enumerate*^{o}.

```
(define enumerateᵒ
  (lambda (op r n)
    (fresh (i j k)
      (bumpᵒ n i)
      (bumpᵒ n j)
      (op i j k)
      (gen&testᵒ op i j k)
      (≡ (i j k) r))))
```

What is the value of

```
(run* (s)
  (enumerateᵒ +ᵒ s (1 1)))
```

43
```
(((1 1) (1 1) (0 1 1))
 ((1 1) (0 1) (1 0 1))
 ((1 1) (1) (0 0 1))
 ((1 1) () (1 1))
 ((0 1) (1 1) (1 0 1))
 ((0 1) (0 1) (0 0 1))
 ((0 1) (1) (1 1))
 ((0 1) () (0 1))
 ((1) (1 1) (0 0 1))
 ((1) (0 1) (1 1))
 ((1) (1) (0 1))
 ((1) () (1))
 (() (1 1) (1 1))
 (() (0 1) (0 1))
 (() (1) (1))
 (() () ())).
```

Describe the values in the previous frame.

44 The values are arranged into four groups of four values. Within the first group, the first value is always (1 1); within the second group, the first value is always (0 1); etc. Then, within each group, the second value ranges from (1 1) to (), consecutively. And the third value, of course, is the sum of first two values.

What is true about the value in frame 43?

45 It appears to contain all triples (i j k) where $i + j = k$ with i and j ranging from () to (1 1).

All such triples?

46 It seems so.

Can we be certain without counting and analyzing the values? Can we be sure just by looking at the values?

47 That's confusing.

Okay, suppose one of the triples were missing. For example, suppose ((0 1) (1 1) (1 0 1)) were missing.	48	But how could that be? We know (*bump*^{o} n i) associates i with the numbers within the range () through n. So if we try it enough times, we eventually get all such numbers. The same is true for (*bump*^{o} n j). So, we definitely will determine (*op* i j k) when i is (0 1) and j is (1 1), which will then associate k with (1 0 1). We have already seen that.
Then what happens?	49	Then we will try to find if (*gen&test*^{o} $+^o$ i j k) can succeed, where i is (0 1), j is (1 1), and k is (1 0 1).
At least once?	50	Yes, since we are interested in only one value. We first determine (*op* x y z), where x, y, and z are fresh. Then we see if that result matches ((0 1) (1 1) (1 0 1)). If not, we try (*op* x y z) again, and again.
What if such a triple were found?	51	Then *gen&test*^{o} would succeed, producing the triple as the result of *enumerate*^{o}. Then, because the **fresh** expression in *gen&test*^{o} is wrapped in a *once*^{o}, we would pick a new pair of i-j values, etc.
What if we were unable to find such a triple?	52	Then the **run** expression would have no value.
Why would it have no value?	53	If no result of (*op* x y z) matches the desired triple, then, as in frame 40, we would keep trying (*op* x y z) forever.

54 So can we say that

(**run*** (s)
  (*enumerate*^{o} $+^o$ s **(1 1)**))

produces all such triples **(**i j k**)** where $i+j=k$ with i and j ranging from **()** through **(1 1)**, just by glancing at the value?

Yes, that's clear.
If one triple were missing, we would have no value at all!

---

55 So what does *enumerate*^{o} determine?

It determines that (op x y z) with x, y, and z being fresh eventually generates *all* triples where $x+y=z$. At least, *enumerate*^{o} determines that for numbers x and y being **()** through some n.

---

56 What is the value of

(**run**$^{\mathbf{1}}$ (s)
  (*enumerate*^{o} $+^o$ s **(1 1 1)**))

(((1 1 1) (1 1 1) (0 1 1 1))).

---

57 How does this definition of *gen-adder*^{o} differ from the one in 7:118?

```
(define gen-adderᵒ
  (lambda (d n m r)
    (fresh (a b c e x y z)
      (≡ (a . x) n)
      (≡ (b . y) m) (posᵒ y)
      (≡ (c . z) r) (posᵒ z)
      (all
        (full-adderᵒ d a b c e)
        (adderᵒ e x y z))))))
```

The definition in chapter 7 has an **all**^{i}, whereas this definition uses **all**.

---

58 What is the value of

(**run**$^{\mathbf{1}}$ (q)
  (*gen&test*^{o} $+^o$ **(0 1) (1 1) (1 0 1)**))

using the second definition of *gen-adder*^{o}

It has no value.

Why doesn't

(**run**$^{\mathbf{1}}$ (q)
  (*gen&test*^{o} $+^o$ (0 1) (1 1) (1 0 1)))

have a value?

59 When using **all** instead of **all**^{i}, things can get stuck.

Where does the second definition of *gen-adder*^{o} get stuck?

60 If a, b, c, d, x, y, and z are all fresh, then (*full-adder*^{o} d a b c e) finds such bits where $d + a + b = c + 2 \cdot e$ and (*adder*^{o} e x y z) will find the rest of the numbers. But there are several ways to solve this equation. For example, both $0 + 0 + 0 = 0 + 2 \cdot 0$ and $0 + 1 + 0 = 1 + 2 \cdot 0$ work. Because (*adder*^{o} e x y z) keeps generating new x, y, and z forever, we never get a chance to explore other values. Because (*full-adder*^{o} d a b c e) is within an **all**, not an **all**^{i}, the (*full-adder*^{o} d a b c e) gets stuck on its first value.

Good. Let's see if it is true. Redo the effort of frame 103 and frame 115 but using the second definition of *gen-adder*^{o}. What do we discover?

61 Some things are missing like ((1) (1 1 0 $\_{0}$. $\_{1}$) (0 0 1 $\_{0}$. $\_{1}$)) and ((0 1) (1 1) (1 0 1)).

If something is missing because we are using the second definition of *gen-adder*^{o}, can we predict the value of

(**run*** (q)
  (*enumerate*^{o} $+^o$ q (1 1 1)))

62 Of course, we know that it has no value.

Can *log*^{o} and $\div^o$ also be enumerated?

63 Yes, of course.

$\Longrightarrow$ Get ready to connect the wires. $\Longleftarrow$

Connecting the Wires

A goal g is a function that maps a substitution s to an ordered sequence s^∞ of zero or more substitutions. (For clarity, we notate **lambda** as λ_{G} when creating such a function g.) Because the sequence of substitutions may be infinite, we represent it not as a list but a stream.

Streams contain either zero, one, or more substitutions.[1] We use (**mzero**) to represent the empty stream of substitutions. For example, `#u` maps every substitution to (**mzero**). If a is a substitution, then (**unit** a) represents the stream containing just a. For instance, `#s` maps every substitution s to just (**unit** s). The goal created by an invocation of the $\equiv$ operator maps a substitution s to either (**mzero**) or to a stream containing a single (possibly extended) substitution, depending on whether that goal fails or succeeds. To represent a stream containing multiple substitutions, we use (**choice** a f), where a is the first substitution in the stream, and where f is a function of zero arguments. Invoking the function f produces the remainder of the stream, which may or may not be empty. (For clarity, we notate **lambda** as λ_{F} when creating such a function f.)

When we use the variable a rather than s for substitutions, it is to emphasize that this representation of streams works for other kinds of data, as long as a datum is never `#f` or a pair whose *cdr* is a function—in other words, as long as the three cases above are never represented in overlapping ways. To discriminate among the cases we define the macro **case**$^\infty$.

The second case is redundant in this representation: (**unit** a) can be represented as (**choice** a (λ_{F} () `#f`)). We include **unit**, which avoids building and taking apart pairs and invoking functions, because many goals never return multiple substitutions. **run** converts a stream of substitutions s^∞ to a list of values using map^∞.

Two streams can be merged either by concatenating them using *mplus* (also known as *stream-append*) or by interleaving them using $mplus^i$. The only difference between the definitions *mplus* and $mplus^i$ lies in the recursive case: $mplus^i$ swaps the two streams; *mplus* does not.

Given a stream s^∞ and a goal g, we can feed each value in s^∞ to the goal g to get a new stream, then merge all these new streams together using either *mplus* or $mplus^i$. When using *mplus*, this operation is called monadic[2] *bind*, and it is used to implement the conjunction **all**. When using $mplus^i$, this operation is called $bind^i$, and it is used to implement the fair conjunction **all**^{i}. The operators **all** and **all**^{i} are like **and**, since they are short-circuiting: the false value short-circuits **and**, and any failed goal short-circuits **all** and **all**^{i}. Also, the **let** in the third clause of **all-aux** ensures that (**all** e), (**all**^{i} e), (**all** e `#s`), and (**all**^{i} e `#s`) are equivalent to e, even if the expression e has no value. The addition of the superfluous second clause allows **all-aux** expressions to expand to simpler code.

To take the disjunction of goals we define **cond**^{e}, and to take the fair disjunction we define **cond**^{i}. They combine successive question-answer lines using *mplus* and $mplus^i$, respectively. Two stranger kinds of disjunction are **cond**^{a} and **cond**^{u}. When a question g_0 succeeds, both **cond**^{a} and **cond**^{u} skip the remaining lines. However, **cond**^{u} chops off every substitution after the first produced by g_0, whereas **cond**^{a} leaves the stream produced by g_0 intact.

[1] See Philip L. Wadler. How to replace failure by a list of successes: a method for exception handling, backtracking, and pattern matching in lazy functional languages. *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 201, Springer, pages 113–128; J. Michael Spivey and Silvija Seres. Combinators for logic programming. *The Fun of Programming.* Palgrave; and Mitchell Wand and Dale Vaillancourt. Relating Models of Backtracking. *Ninth International Conference on Functional Programming.* 2004, pages 54–65.

[2] See Eugenio Moggi. Notions of computation and monads. *Information and Computation 93(1)*:55–92, 1991; Philip L. Wadler. The essence of functional programming. *Nineteenth Symposium on Principles of Programming Languages*. 1992, pages 1–14; and Ralf Hinze. Deriving backtracking monad transformers. *Fifth International Conference on Functional Programming*. 2000, pages 186–197.

1

```
(define-syntax run                                9 : 6, 13, 47, 58
  (syntax-rules ()
    ((_ n̂ (x) g ...)
     (let ((n n̂) (x (var x)))
       (if (or (not n) (> n 0))
         (map∞ n
           (lambda (s)
             (reify (walk* x s)))
           ((all g ...) empty-s))
         ())))))

(define-syntax case∞
  (syntax-rules ()
    ((_ e on-zero ((â) on-one) ((a f) on-choice))
     (let ((a∞ e))
       (cond
         ((not a∞) on-zero)
         ((not (and
                 (pair? a∞)
                 (procedure? (cdr a∞))))
          (let ((â a∞))
            on-one))
         (else (let ((a (car a∞)) (f (cdr a∞)))
                 on-choice)))))))
```

```
(define-syntax mzero
  (syntax-rules ()
    ((_) #f)))

(define-syntax unit
  (syntax-rules ()
    ((_ a) a)))

(define-syntax choice
  (syntax-rules ()
    ((_ a f) (cons a f))))

(define map∞
  (lambda (n p a∞)
    (case∞ a∞
      ()
      ((a)
       (cons (p a) ()))
      ((a f)
       (cons (p a)
         (cond
           ((not n) (map∞ n p (f)))
           ((> n 1) (map∞ (− n 1) p (f)))
           (else ())))))))
```

2

```
(define #s (λG (s) (unit s)))

(define #u (λG (s) (mzero)))

(define ≡                                         9 : 27, 36
  (lambda (v w)
    (λG (s)
      (cond
        ((unify v w s) ⇒ #s)
        (else (#u s))))))

(define-syntax fresh                              9 : 6
  (syntax-rules ()
    ((_ (x ...) g ...)
     (λG (s)
       (let ((x (var x)) ...)
         ((all g ...) s))))))

(define-syntax condᵉ
  (syntax-rules ()
    ((_ c ...) (cond-aux ifᵉ c ...))))
```

```
(define-syntax all
  (syntax-rules ()
    ((_ g ...) (all-aux bind g ...))))

(define-syntax allⁱ
  (syntax-rules ()
    ((_ g ...) (all-aux bindⁱ g ...))))

(define-syntax condⁱ
  (syntax-rules ()
    ((_ c ...) (cond-aux ifⁱ c ...))))

(define-syntax condᵃ
  (syntax-rules ()
    ((_ c ...) (cond-aux ifᵃ c ...))))

(define-syntax condᵘ
  (syntax-rules ()
    ((_ c ...) (cond-aux ifᵘ c ...))))
```

3

```
(define mplus
  (lambda (a∞ f)
    (case∞ a∞
      (f)
      ((a) (choice a f))
      ((a f_0) (choice a
                 (λ_F () (mplus (f_0) f)))))))

(define bind
  (lambda (a∞ g)
    (case∞ a∞
      (mzero)
      ((a) (g a))
      ((a f) (mplus (g a)
               (λ_F () (bind (f) g)))))))
```

```
(define mplus^i
  (lambda (a∞ f)
    (case∞ a∞
      (f)
      ((a) (choice a f))
      ((a f_0) (choice a
                 (λ_F () (mplus^i (f) f_0)))))))

(define bind^i
  (lambda (a∞ g)
    (case∞ a∞
      (mzero)
      ((a) (g a))
      ((a f) (mplus^i (g a)
               (λ_F () (bind^i (f) g)))))))
```

4

```
(define-syntax cond-aux
  (syntax-rules (else)
    ((_ ifer) #u)
    ((_ ifer (else g ...)) (all g ...))
    ((_ ifer (g ...)) (all g ...))
    ((_ ifer (g_0 g ...) c ...)
     (ifer g_0
       (all g ...)
       (cond-aux ifer c ...)))))
```

```
(define-syntax all-aux
  (syntax-rules ()
    ((_ bnd) #s)
    ((_ bnd g) g)
    ((_ bnd g_0 g ...)
     (let ((ĝ g_0))
       (λ_G (s)
         (bnd (ĝ s)
           (λ_G (s) ((all-aux bnd g ...) s))))))))
```

5

```
(define-syntax if^e
  (syntax-rules ()
    ((_ g_0 g_1 g_2)
     (λ_G (s)
       (mplus ((all g_0 g_1) s) (λ_F () (g_2 s)))))))
```

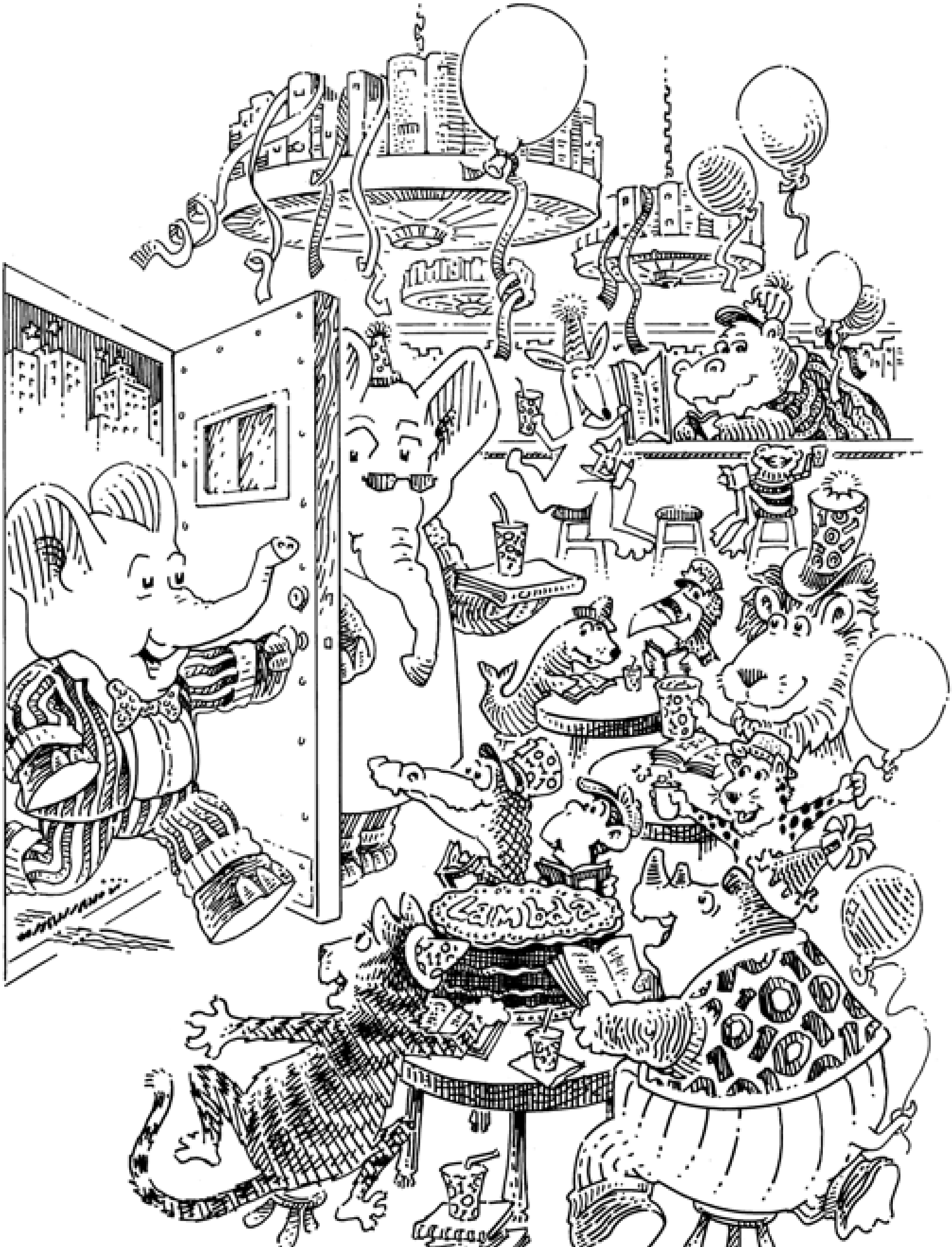
```
(define-syntax if^i
  (syntax-rules ()
    ((_ g_0 g_1 g_2)
     (λ_G (s)
       (mplus^i ((all g_0 g_1) s) (λ_F () (g_2 s)))))))
```

6

```
(define-syntax if^a
  (syntax-rules ()
    ((_ g_0 g_1 g_2)
     (λ_G (s)
       (let ((s∞ (g_0 s)))
         (case∞ s∞
           (g_2 s)
           ((s) (g_1 s))
           ((s f) (bind s∞ g_1))))))))
```

```
(define-syntax if^u
  (syntax-rules ()
    ((_ g_0 g_1 g_2)
     (λ_G (s)
       (let ((s∞ (g_0 s)))
         (case∞ s∞
           (g_2 s)
           ((s) (g_1 s))
           ((s f) (g_1 s))))))))
```

Welcome to the Club



Here is a small collection of entertaining and illuminating books.

Carroll, Lewis. *The Annotated Alice: The Definitive Edition.* W. W. Norton & Company, New York, 1999. Introduction and notes by Martin Gardner.

Hein, Piet. *Grooks.* The MIT Press, 1960.

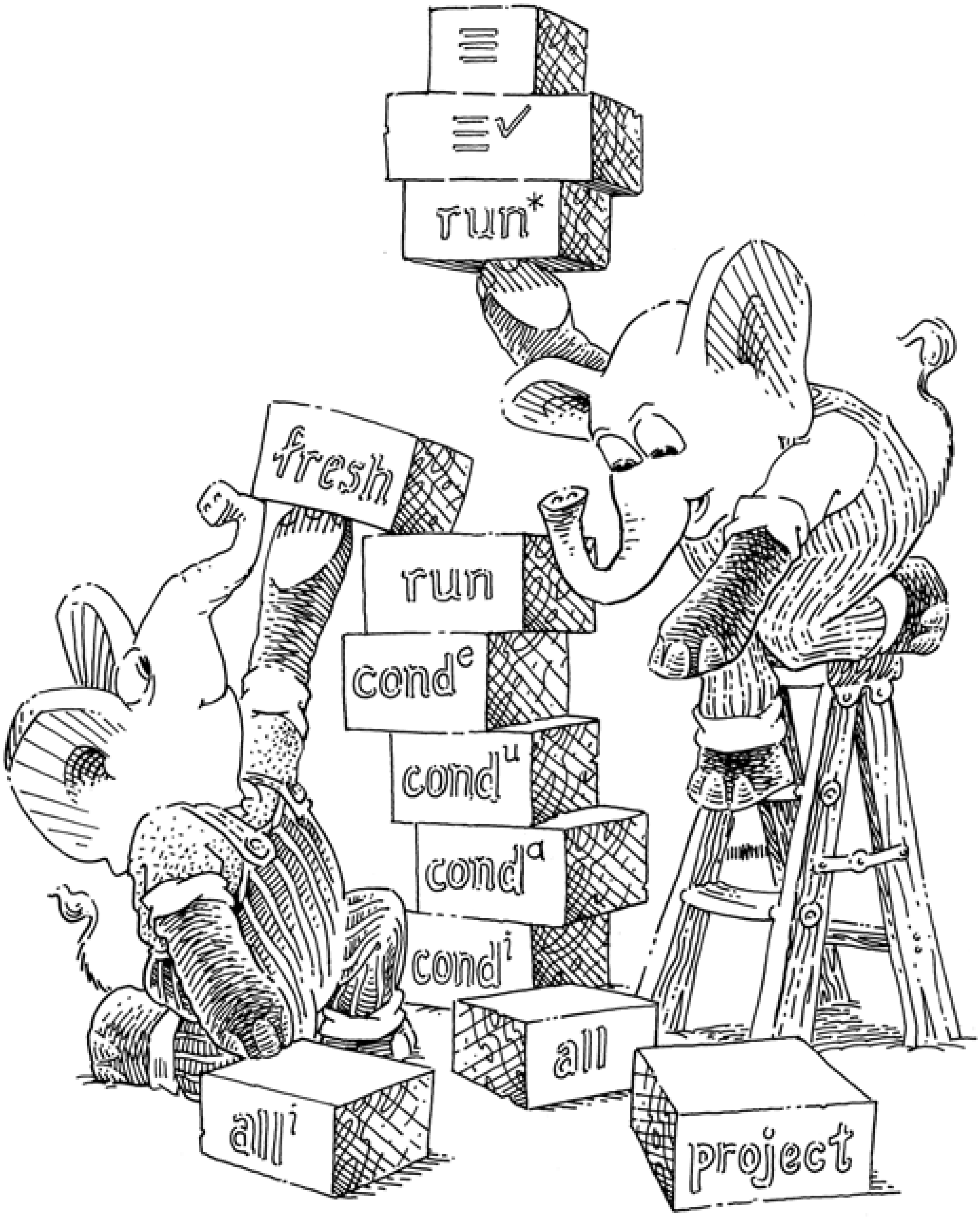
Hofstadter, Douglas R. *Gödel, Escher, Bach: an Eternal Golden Braid.* Basic Books, Inc., 1979.

Nagel, Ernest, and James R. Newman. *Gödel's Proof.* New York University Press, 1958.

Smullyan, Raymond. *To Mock a Mockingbird.* Alfred A. Knopf, Inc., 1985.

Suppes, Patrick. *Introduction to Logic.* Van Nostrand Co., 1957.

Index



Index

Italic page numbers refer to definitions.