The Little LISPer

Third Edition

Daniel P. Friedman

Indiana University
Bloomington, Indiana

Matthias Felleisen

Rice University
Houston, Texas

Exercises

1.1 Think of ten different atoms and write them down.

1.2 Using the atoms of Exercise 1.1, make up twenty different lists.

1.3 The list (all these problems) can be constructed by (cons a (cons b (cons c d))), where
a is all,
b is these,
c is problems, and
d is ( ).

Write down how you would construct the following lists:

(all (these problems))
(all (these) problems)
((all these) problems)
((all these problems))

1.4 What is (car (cons a l)), where a is french, and l is (fries);
and what is (cdr (cons a l)), where a is oranges, and l is (apples and peaches)?

1.5 Find an atom x that makes (eq? x y) true, where y is lisp. Are there any others?

1.6 If a is atom, is there a list l that makes (null? (cons a l)) true?

1.7 Determine the value of

(cons s l), where s is x, and l is y
(cons s l), where s is ( ), and l is ( )
(car s), where s is ( )
(cdr l), where l is (( ))

1.8 True or false,

(atom? (car l)), where l is ((meatballs) and spaghetti)
(null? (cdr l)), where l is ((meatballs))
(eq? (car l) (car (cdr l))), where l is (two meatballs)
(atom? (cons a l)), where l is (ball) and a is meat

1.9 What is

(car (cdr (cdr (car l)))) where l is ((kiwis mangoes lemons) and (more))
(car (cdr (car (cdr l)))) where l is (( ) (eggs and (bacon)) (for) (breakfast))
(car (cdr (cdr (cdr l)))) where l is (( ) ( ) ( ) (and (coffee)) please)

1.10 To get the atom and in (peanut butter and jelly on toast) we can write (car (cdr (cdr l))). What would you write to get:

Harry in l, where l is (apples in (Harry has a backyard))
where l is (apples and Harry)
where l is (((apples) and ((Harry))) in his backyard)

Exercises

For these exercises,

l1 is (german chocolate cake)
l2 is (poppy seed cake)
l3 is ((linzer) (torte) ( ))
l4 is ((bleu cheese) (and) (red) (wine))
l5 is (( ) ( ))
a1 is coffee
a2 is seed
a3 is poppy

2.1 What are the values of: (lat? *l1*), (lat? *l2*), and (lat? *l3*)?

2.2 For each case in Exercise 2.1 step through the application as we did in this chapter.

2.3 What is the value of (member? *a1* *l1*), and (member? *a2* *l2*)?
Step through the application for each case.

2.4 Most Lisp dialects have an (**if** ...)-form. In general an (**if** ...)-form looks like this:

(**if** *aexp* *bexp* *cexp*).

When *aexp* is true, (**if** *aexp* *bexp* *cexp*) is *bexp*; when it is false, (**if** *aexp* *bexp* *cexp*) is *cexp*. For example,

```
(cond
  ((null? l) nil)
  (t (or
       (eq? (car l) a)
       (member? a (cdr l)))))
```

in member? can be replaced by:

```
(if (null? l)
    nil
    (or
      (eq? (car l) a)
      (member? a (cdr l))))
```

Rewrite all the functions in the chapter using (**if** ...) instead of (**cond** ...).

2.5 Write the function nonlat?, which determines whether a list of S-expressions does not contain atomic S-expressions.

Example: (nonlat? *l1*) is false,
(nonlat? '()) is true,
(nonlat? *l3*) is false,
(nonlat? *l4*) is true.

2.6 Write a function member-cake?, which determines whether a lat contains the atom cake.

Example: (member-cake? *l1*) is true,
(member-cake? *l2*) is true,
(member-cake? *l5*) is false.

2.7 Consider the following new definition of member?

```
(define member2?
  (lambda (a lat)
    (cond
      ((null? lat) nil)
      (t (or
           (member2? a (cdr lat))
           (eq? a (car lat)))))))
```

Do (member2? *a l*) and (member? *a l*) give the same answer when we use the same arguments? Consider the examples *a1* and *l1*, *a1* and *l2*, and *a2* and *l2*.

2.8 Step through the applications (member? *a3 l2*) and (member2? *a3 l2*). Compare the steps of the two applications.

2.9 What happens when you step through (member? *a2 l3*)? Fix this problem by having member? ignore lists.

2.10 The function member? tells whether some atom appears *at least once* in a lat. Write a function member-twice?, which tells whether some atom appears *at least twice* in a lat.

Exercises

For these exercises,

$l1$ is ((paella spanish) (wine red) (and beans))
$l2$ is ( )
$l3$ is (cincinnati chili)
$l4$ is (texas hot chili)
$l5$ is (soy sauce and tomato sauce)
$l6$ is ((spanish) ( ) (paella))
$l7$ is ((and hot) (but dogs))
$a1$ is chili
$a2$ is hot
$a3$ is spicy
$a4$ is sauce
$a5$ is soy

3.1 Write the function seconds, which takes a list of lats and makes a new lat consisting of the second atom from each lat in the list.

Example: (seconds $l1$) is (spanish red beans)
(seconds $l2$) is ( )
(seconds $l7$) is (hot dogs)

3.2 Write the function dupla of an atom a and a list of atoms l, which makes a new lat containing as many a's as there are elements in l.

Example: (dupla $a2$ $l4$) is (hot hot hot)
(dupla $a2$ $l2$) is ( )
(dupla $a1$ $l5$) is (chili chili chili chili chili)

3.3 Write the function double of a and l, which is a converse to rember. The function doubles the first a in l instead of removing it.

Example: (double $a2$ $l2$) is ( )
(double $a1$ $l3$) is (cincinnati chili chili)
(double $a2$ $l4$) is (texas hot hot chili)

3.4 Write the function subst-sauce of a and l, which substitutes a for the first atom sauce in l.

Example: (subst-sauce $a1$ $l4$) is (texas hot chili)
(subst-sauce $a1$ $l5$) is (soy chili and tomato sauce)
(subst-sauce $a4$ $l2$) is ( )

3.5 Write the function subst3 of new, $o1$, $o2$, $o3$, and lat, which—like subst2—replaces the first occurrence of either $o1$, $o2$, or $o3$ in lat by new.

Example: (subst3 $a5$ $a1$ $a2$ $a4$ $l5$) is (soy soy and tomato sauce)
(subst3 $a4$ $a1$ $a2$ $a3$ $l4$) is (texas sauce chili)
(subst3 $a3$ $a1$ $a2$ $a5$ $l2$) is ( )

3.6 Write the function substN of new, $slat$, and lat, which replaces the first atom in lat that also occurs in $slat$ by the atom new.

Example: (substN $a2$ $l3$ $l4$) is (texas hot hot)
(substN $a4$ $l3$ $l5$) is (soy sauce and tomato sauce)
(substN $a4$ $l3$ $l2$) is ( )

3.7 Step through the application (rember $a4$ $l5$). Also step through (insertR $a5$ $a2$ $l5$) for the "bad" definitions of insertR.

3.8 Determine the *typical elements* and the *natural recursions* of the functions that you have written so far.

3.9 Write the function rember2 of a and l, which removes the second occurrence of a in l.

Example: (rember2 $a1$ $l3$) is (cincinnati chili)
(rember2 $a4$ $l5$) is (soy sauce and tomato)
(rember2 $a4$ $l2$) is ( )

Hint: Use the function rember.

3.10 Consider the functions insertR, insertL, and subst. They are all very similar. Write the functions next to each other and draw boxes around the parts that they have in common. Can you see what rember has in common with these functions?

Exercises

For these exercises,

$vec1$ is (1 2)
$vec2$ is (3 2 4)
$vec3$ is (2 1 3)
$vec4$ is (6 2 1)
l is ( )
$zero$ is 0
one is 1
$three$ is 3
obj is (x y)

4.1 Write the function duplicate of n and obj, which builds a list containing n objects obj.

Example: (duplicate $three$ obj) is ((x y) (x y) (x y)),
(duplicate $zero$ obj) is ( ),
(duplicate one $vec1$) is ((1 2)).

4.2 Write the function multvec that builds a number by multiplying all the numbers in a vec.

Example: (multvec $vec2$) is 24,
(multvec $vec3$) is 6,
(multvec l) is 1.

4.3 When building a value with ↑, which value should you use for the terminal line?

4.4 Argue the correctness for the function ↑ as we did for $(\times\ n\ m)$. Use 3 and 4 as data.

4.5 Write the function index of an atom a and a list of atoms l that returns the place of the atom a in l. If a is not in l, then the value of (index a l) is false.

Example: When a is car,
$lat1$ is (cons cdr car null? eq?),
b is motor, and
$lat2$ is (car engine auto motor),

we have (index a $lat1$) is 3,
(index a $lat2$) is 1,
(index a '()) is *nil*,
(index b $lat2$) is 4.

4.6 Write the function product of $vec1$ and $vec2$ that multiplies corresponding numbers in $vec1$ and $vec2$ and builds a new vec from the results. The vecs, $vec1$ and $vec2$, may differ in length.

Example: (product $vec1$ $vec2$) is (3 4 4),
(product $vec2$ $vec3$) is (6 2 12),
(product $vec3$ $vec4$) is (12 2 3).

4.7 Write the function dot-product of $vec1$ and $vec2$ that multiplies corresponding numbers in $vec1$ and $vec2$ and builds a new $number$ by summing the results. The vecs, $vec1$ and $vec2$, are the same length.

Example: (dot-product $vec2$ $vec2$) is 29,
(dot-product $vec2$ $vec4$) is 26,
(dot-product $vec3$ $vec4$) is 17.

4.8 Write the function / that divides nonnegative integers.

Example: (/ n m) is 1, when n is 7 and m is 5.
(/ n m) is 4, when n is 8 and m is 2.
(/ n m) is 0, when n is 2 and m is 3.

Hint: A number is now defined as a rest (between 0 and $m - 1$) and a multiple addition of m. The number of additions is the result.

4.9 Here is the function remainder:

```
(define remainder
  (lambda (n m)
    (cond
      (t (− n (× m (/ n m)))))))
```

Make up examples for the application (remainder n m) and work through them.

4.10 Write the function $\leq$, which tests if two numbers are equal or if the first is less than the second.

Example: $(\leq\ zero\ \ one)$ is true,
$(\leq\ one\ \ one)$ is true,
$(\leq\ three\ \ one)$ is false.

Exercises

For these exercises,

x is comma
y is dot
a is kiwis
b is plums
$lat1$ is (bananas kiwis)
$lat2$ is (peaches apples bananas)
$lat3$ is (kiwis pears plums bananas cherries)
$lat4$ is (kiwis mangoes kiwis guavas kiwis)
$l1$ is ((curry) ( ) (chicken) ( ))
$l2$ is ((peaches) (and cream))
$l3$ is ((plums) and (ice) and cream)
$l4$ is ( )

5.1 For Exercise 3.4 you wrote the function subst-cake. Write the function multisubst-kiwis.

Example: (multisubst-kiwis b $lat1$) is (bananas plums),
(multisubst-kiwis y $lat2$) is (peaches apples bananas),
(multisubst-kiwis y $lat4$) is (dot mangoes dot guavas dot),
(multisubst-kiwis y $l4$) is ( ).

5.2 Write the function multisubst2. You can find subst2 at the end of Chapter 3.

Example: (multisubst2 x a b $lat1$) is (bananas comma),
(multisubst2 y a b $lat3$) is (dot pears dot bananas cherries),
(multisubst2 a x y $lat1$) is (bananas kiwis).

5.3 Write the function multidown of *lat* which replaces every atom in *lat* by a list containing the atom.

Example: (multidown *lat1*) is ((bananas) (kiwis)),
(multidown *lat2*) is ((peaches) (apples) (bananas)),
(multidown *l4*) is ( ).

5.4 Write the function occurN of a list of atoms *markers* and a second list of atoms *lat*, which counts how many times the atoms in *markers* also occur in *lat*.

Example: (occurN *lat1* *l4*) is 0,
(occurN *lat1* *lat2*) is 1,
(occurN *lat1* *lat3*) is 2.

5.5 The function I of *lat1* and *lat2* returns the first atom in *lat2* that is in both *lat1* and *lat2*. Write the functions I and multiI. multiI returns a list of atoms common to *lat1* and *lat2*.

Example: (I *lat1* *l4*) is ( ),
(I *lat1* *lat2*) is bananas,
(I *lat1* *lat3*) is kiwis;
(multiI *lat1* *l4*) is ( ),
(multiI *lat1* *lat2*) is (bananas),
(multiI *lat1* *lat3*) is (kiwis bananas).

5.6 Consider the following alternative definition of one?

```
(define one?
  (lambda (n)
    (cond
      ((zero? (sub1 n)) t)
      (t nil))))
```

Which Laws and/or Commandments does it violate?

5.7 Consider the following definition of =

```
(define =
  (lambda (n m)
    (cond
      ((zero? n) (zero? m))
      (t (= n (sub1 m))))))
```

This definition violates The Sixth Commandment. Why?

5.8 The function count0 of *vec* counts the number of zero elements in *vec*. What is wrong with the following definition? Can you fix it?

```
(define count0
  (lambda (vec)
    (cond
      ((null? vec) 1)
      (t (cond
           ((zero? (car vec))
            (cons 0 (count0 (cdr vec))))
           (t (count0 (cdr vec))))))))
```

5.9 Write the function multiup of *l*, which replaces every lat of length one in *l* by the atom in that list, and which also removes every empty list.

Example: (multiup *l4*) is ( ),
(multiup *l1*) is (curry chicken),
(multiup *l2*) is (peaches (and cream)).

5.10 Review all the Laws and Commandments. Check the functions in Chapters 4 and 5 to see if they obey the Commandments. When did we not obey them literally? Did we act according to their spirit?

But answer came there none—
And this was scarcely odd, because
They'd eaten every one.

The Walrus and The Carpenter
—*Lewis Carroll*

Exercises

For these exercises,

l1 is ((fried potatoes) (baked (fried)) tomatoes)
l2 is (((chili) chili (chili)))
l3 is ( )
lat1 is (chili and hot)
lat2 is (baked fried)
a is fried

6.1 Write the function down* of a general list *l*, which puts every atom in *l* in a list by itself.

Example: (down* *l2*) is ((((chili)) (chili) ((chili)))),
(down* *l3*) is ( ),
(down* *lat1*) is ((chili) (and) (hot)).

6.2 Write the function occurN* of a list of atoms *markers* and a general list *l*, which counts how many times the atoms in *markers* also occur in *l*.

Example: (occurN* *lat1* *l2*) is 3,
(occurN* *lat2* *l1*) is 3,
(occurN* *lat1* *l3*) is 0.

6.3 Write the function double* of an atom *a* and a general list *l*, which doubles each occurrence of *a* in *l*.

Example: (double* *a* *l1*) is ((fried fried potatoes) (baked (fried fried)) tomatoes),
(double* *a* *l2*) is (((chili) chili (chili))),
(double* *a* *lat2*) is (baked fried fried).

6.4 Consider the function lat? from Chapter 2. Argue why lat? has to ask three questions (and not two like the other functions in Chapter 2). Why does lat? not have to recur on the car?

6.5 Make sure that (member* *a* *l*), where

a is chips and
l is ((potato) (chips ((with) fish) (chips))),

really discovers the first chips. Can you change member* so that it finds the last chips first?

6.6 Write the function list+, which adds up all the numbers in a general list of numbers.

Example: When *l1* is ((1 (6 6 ( )))),
and *l2* is ((1 2 (3 6)) 1), then
(list+ *l1*) is 13,
(list+ *l2*) is 13,
(list+ *l3*) is 0.

6.7 Consider the following function g* of *lvec* and *acc*.

```
(define g*
  (lambda (lvec acc)
    (cond
      ((null? lvec) acc)
      ((atom? (car lvec))
       (g* (cdr lvec) (+ (car lvec) acc)))
      (t (g* (car lvec) (g* (cdr lvec) acc))))))
```

The function is always applied to a (general) list of numbers and 0. Make up examples and find out what the function does.

6.8 Consider the following function f* of *l* and *acc*.

```
(define f*
  (lambda (l acc)
    (cond
      ((null? l) acc)
      ((atom? (car l))
       (cond
         ((member? (car l) acc) (f* (cdr l)
acc))
         (t (f* (cdr l) (cons (car l) acc)))))
      (t (f* (car l) (f* (cdr l) acc))))))
```

The function is always applied to a list and the empty list. Make up examples for *l* and step through the applications. Generalize in one sentence what f* does.

6.9 The functions in Exercises 6.7 and 6.8 employ the *accumulator technique*. This means that they pass along an argument that represents the result that has been computed so far. When these functions reach the bottom (null?, zero?), they just return the result contained in the accumulator. The original argument for the accumulator is the element that used to be the answer for the null?-case. Write the function occur (see Chapter 5) of *a* and *lat* using the accumulator technique. What is the original value for *acc*?

6.10 Step through an application of the original occur and the occur from Exercise 6.9 and compare the arguments in the recursive applications. Can you write occur* using the accumulator technique?

Have you taken a tea break yet?
We're taking ours now.

Exercises

For these exercises,

aexp1 is (1 + (3 × 4))
aexp2 is ((3 ↑ 4) + 5)
aexp3 is (3 × (4 × (5 × 6)))
aexp4 is 5
l1 is ( )
l2 is (3 + (66 6))
lexp1 is (AND (OR x y) y)
lexp2 is (AND (NOT y) (OR u v))
lexp3 is (OR x y)
lexp4 is z

7.1 So far we have neglected functions that build representations for arithmetic expressions. For example, mk+exp

```
(define mk+exp
  (lambda (aexp1 aexp2)
    (cons aexp1
      (cons (quote +)
        (cons aexp2 ( ))))))
```

makes an arithmetic expression of the form (*aexp1* + *aexp2*), where *aexp1*, *aexp2* are already arithmetic expressions. Write the corresponding functions mk×exp and mk↑exp.

The arithmetic expression (1 + 3) can now be built by (mk+exp x y), where x is 1 and y is 3. Show how to build *aexp1*, *aexp2*, and *aexp3*.

7.2 A useful function is aexp? that checks whether an S-expression is the representation of an arithmetic expression. Write the function aexp? and test it with some of the arithmetic expressions from the chapter. Also test it with S-expressions that are not arithmetic expressions.

Example: (aexp? *aexp1*) is true,
(aexp? *aexp2*) is true,
(aexp? *l1*) is false,
(aexp? *l2*) is false.

7.3 Write the function count-op that counts the operators in an arithmetic expression.

Example: (count-op *aexp1*) is 2,
(count-op *aexp3*) is 3,
(count-op *aexp4*) is 0.

Also write the functions count+, count×, and count↑ that count the respective operators.

Example: (count+ *aexp1*) is 1,
(count× *aexp1*) is 1,
(count↑ *aexp1*) is 0.

7.4 Write the function count-numbers that counts the numbers in an arithmetic expression.

Example: (count-numbers *aexp1*) is 3,
(count-numbers *aexp3*) is 4,
(count-numbers *aexp4*) is 1.

7.5 Since it is inconvenient to write (3 × (4 × (5 × 6))) for multiplying 4 numbers, we now introduce prefix notation and allow + and × expressions to contain 2, 3, or 4 subexpressions. For example, (+ 3 2 (× 7 8)), (× 3 4 5 6) etc. are now legal representations. ↑-expressions are also in prefix form but are still binary.

Rewrite the functions numbered? and value for the new definition of aexp.

Hint: You will need functions for extracting the third and the fourth subexpression of an arithmetic expression. You will also need a function cnt-aexp that counts the number of arithmetic subexpressions in the *list* following an operator.

Example: When *aexp1* is (+ 3 2 (× 7 8)),
aexp2 is (× 3 4 5 6), and
aexp3 is (↑ *aexp1* *aexp2*), then
(cnt-aexp *aexp1*) is 3,
(cnt-aexp *aexp2*) is 4,
(cnt-aexp *aexp3*) is 2.

For exercises 7.6 through 7.10 we define a representation for L-expressions. An L-expression is defined in the following way: It is either:

—(AND *l1 l2*), or
—(OR *l1 l2*), or
—(NOT *l*), or
—an arbitrary symbol. We call such a symbol a *variable*.

In this definition, AND, OR, and NOT are literal symbols; *l*, *l1*, *l2* stand for arbitrary L-expressions.

7.6 Write the function lexp? that tests whether an S-expression is a representation of an L-expression.

Example: (lexp? *lexp1*) is true,
(lexp? *lexp2*) is true,
(lexp? *lexp3*) is true,
(lexp? *aexp1*) is false,
(lexp? *l2*) is false.

Also write the functions and-exp? or-exp? and not-exp? which test whether or not an S-expression is a representation of an L-expression of the respective shape.
Write the functions and-exp-left and and-exp-right, which extract the left and the right part of an (recognized) L-expression.

Example: (and-exp-left *lexp1*) is (OR x y),
(and-exp-right *lexp1*) is y,
(and-exp-left *lexp2*) is (NOT y),
(and-exp-right *lexp2*) is (OR u v).

Finally, write the functions or-exp-left, or-exp-right, and not-exp-subexp, which extract the respective piecs of OR and NOT L-expressions.

7.7 Write the function covered? of an L-expression *lexp* and a list of symbols *los* that tests whether all the variables in *lexp* are in *los*.

Example: When *l1* is (x y z u), then
(covered? *lexp1 l1*) is true,
(covered? *lexp2 l1*) is false,
(covered? *lexp4 l1*) is true.

7.8 For the evaluation of L-expressions we need *association lists* (*alists*). An alist for L-expressions is a list of pairs. The first component of a pair is always a symbol, the second one is either the number 0 (signifying false) or 1 (signifying true). The second component is referred to as the value of the variable. Write the function lookup of the symbol *var* and the association list *al*, which returns the value of the first pair in *al* whose car is eq? to *var*.

Example: When *l1* is ((x 1) (y 0)),
l2 is ((u 1) (v 1)),
l3 is ( ),
a is y,
b is u, then
(lookup *a l1*) is 0,
(lookup *b l2*) is 1,
(lookup *a l3*) has an unspecified answer.

7.9 If the list of symbols in an alist for L-expressions contains all the variables of an L-expression *lexp*, then *lexp* is called closed and can be evaluated with respect to this alist. Write the function Mlexp of an L-expression *lexp* and an alist *al*, which, after verifying that *lexp* is closed, determines whether *lexp* means true or false.

Given *al* such that *lexp* is covered *lexp*, *exp* means true

— if *lexp* is a variable and its value means true, or

— if *lexp* is an AND-expression and both subexpressions mean true, or

— if *lexp* is an OR-expression and one of the subexpressions means true, or

— if *lexp* is a NOT-expression and the subexpression means false.

Otherwise *lexp* means false.

If *lexp* is not closed in *al*, then (Mlexp *lexp al*) returns the symbol not-covered.

Example: When *l1* is ((x 1) (y 0) (z 0)),
l2 is ((y 0) (u 0) (v 1)), then
(Mlexp *lexp1 l1*) is false,
(Mlexp *lexp2 l2*) is true,
(Mlexp *lexp4 l1*) is false.

Hint: You will need the function lookup from Exercise 7.8 and covered? from Exercise 7.7.

7.10 Extend the representation of L-expressions to AND and OR expressions that contain several subexpressions, i.e.,
(AND x (OR u v w) y).
Rewrite the function Mlexp from Exercise 7.9 for this representation.

Hint: Exercise 7.5 is a similar extension of arithmetic expressions.

Exercises

For these exercises,

r1 is ((a b) (a a) (b b))
r2 is ((c c))
r3 is ((a c) (b c))
r4 is ((a b) (b a))
f1 is ((a 1) (b 2) (c 2) (d 1))
f2 is ( )
f3 is ((a 2) (b 1))
f4 is ((1 $) (3 *))
d1 is (a b)
d2 is (c d)
x is a

8.1 Write the function domset of *rel*, which makes a set of all the atoms in *rel*. This set is referred to as *domain of discourse* of the relation *rel*.

Example: (domset *r1*) is (a b),
(domset *r2*) is (c),
(domset *r3*) is (a b c).

Also write the function idrel of s, which makes a relation of all pairs of the form $(d\ d)$ where d is an atom of the set s. (idrel s) is called the *identity relation on s*.

Example: (idrel *d1*) is ((a a) (b b)),
(idrel *d2*) is ((c c) (d d)),
(idrel *f2*) is ( ).

8.2 Write the function reflexive? , which tests whether a relation is *reflexive*. A relation is reflexive if it contains all pairs of the form $(d\ \ d)$ where d is an element of its domain of discourse (see Exercise 8.1).

Example: (reflexive? *r1*) is true,
(reflexive? *r2*) is true,
(reflexive? *r3*) is false.

8.3 Write the function symmetric? , which tests whether a relation is *symmetric*. A relation is symmetric if it is eqset? to its revrel.

Example: (symmetric? *r1*) is false,
(symmetric? *r2*) is true,
(symmetric? *f2*) is true.

Also write the function antisymmetric? , which tests whether a relation is *antisymmetric*. A relation is antisymmetric if the intersection of the relation with its revrel is a subset of the identity relation on its domain of discourse (see Exercise 8.1).

Example: (antisymmetric *r1*) is true,
(antisymmetric *r2*) is true,
(antisymmetric *r4*) is false.

And finally, this is the function asymmetric? , which tests whether a relation is asymmetric:

```
(define asymmetric?
  (lambda (rel)
    (null? (intersect rel (revrel rel)))))
```

Find out which of the sample relations is asymmetric. Characterize asymmetry in one sentence.

8.4 Write the function Fapply of f and x , which returns the value of f at place x. That is, it returns the second of the pair whose first is eq? to x.

Example: (Fapply *f1* x) is 1,
(Fapply *f2* x) has no answer,
(Fapply *f3* x) is 2.

8.5 Write the function Fcomp of f and g, which composes two functions. If g contains an element (x y) and f contains an element (y z), then the composed function (Fcomp f g) will contain (x z).

Example: (Fcomp *f1* *f4*) is ( ),
(Fcomp *f1* *f3*) is ( ),
(Fcomp *f4* *f1*) is ((a $) (d $)),
(Fcomp *f4* *f3*) is ((b $)).

Hint: The function Fapply from Exercise 8.4 may be useful.

8.6 Write the function Rapply of *rel* and x, which returns the *value set* of *rel* at place x. The value set is the set of second components of all the pairs whose first component is eq? to x.

Example: (Rapply *f1* x) is (1),
(Rapply *r1* x) is (b a),
(Rapply *f2* x) is ( ).

8.7 Write the function Rin of x and *set*, which produces a relation of pairs (x d) where d is an element of *set*.

Example: (Rin x *d1*) is ((a a) (a b)),
(Rin x *d2*) is ((a c) (a d)),
(Rin x *f2*) is ( ).

8.8 Relations can be composed with the following function:

```
(define Rcomp
  (lambda (rel1 rel2)
    (cond
      ((null? rel1) (quote ( )))
      (t (union
          (Rin
            (first (car rel1))
            (Rapply rel2 (second (car
rel1))))
          (Rcomp (cdr rel1) rel2))))))
```

See Exercises 8.6 and 8.7.

Find the values of (Rcomp *r1* *r3*), (Rcomp *r1* *f1*), and (Rcomp *r1* *r1*).

8.9 Write the function transitive? , which tests whether a relation is transitive. A relation *rel* is *transitive* if the composition of *rel* with *rel* is a subset of *rel* (see Exercise 8.8).

Example: (transitive? *r1*) is true,
(transitive? *r3*) is true,
(transitive? *f1*) is true.

Find a relation for which transitive? yields false.

8.10 Write the functions quasi-order?, partial-order?, and equivalence?, which test whether a relation is a *quasi-order*, a *partial-order*, or an *equivalence relation*, respectively. A relation is a

—quasi-order if it is reflexive and transitive,

—partial-order if it is a quasi-order and antisymmetric,

—equivalence relation if it is a quasi-order and symmetric.

See Exercises 8.2, 8.3, and 8.9.

For that elephant ate all night,
And that elephant ate all day;
Do what he could to furnish him food,
The cry was still more hay.

Wang: The Man with an Elephant
on His Hands [1891]
—*John Cheever Goodwin*

Exercises

9.1 Look up the functions firsts and seconds in Chapter 3. They can be generalized to a function map of f and l that applies f to every element in l and builds a new list with the resulting values. Write the function map. Then write the function firsts and seconds using map.

9.2 Write the function assq-sf of a, l, sk, and fk. The function searches through l, which is a list of pairs until it finds a pair whose first component is eq? to a. Then the function invokes the function sk with this pair. If the search fails, (fk a) is invoked.

Example: When a is apple,
$b1$ is ( ),
$b2$ is ((apple 1) (plum 2)),
$b3$ is ((peach 3)),
sk is (**lambda** (p)
(build (first p) (add1 (second p)))),
fk is (**lambda** ($name$)
(cons
$name$
(**quote** (not-in-list)))), then

(assq-sf a $b1$ sk fk) is (apple not-in-list),
(assq-sf a $b2$ sk fk) is (apple 2),
(assq-sf a $b3$ sk fk) is (apple not-in-list).

9.3 In the chapter we have derived a Y-combinator that allows us to write recursive functions of one argument without using define. Here is the Y-combinator for functions of two arguments:

```
(define Y2
  (lambda (M)
    ((lambda (future)
       (M (lambda (arg1 arg2)
            ((future future) arg1 arg2))))
     (lambda (future)
       (M (lambda (arg1 arg2)
            ((future future) arg1
arg2)))))))
```

Write the functions =, rempick, and pick from Chapter 4 using Y2.

Note: There is a version of (**lambda** ...) for defining a function of an arbitrary number of arguments, and an `apply` function for applying such a function to a list of arguments. With this you can write a single Y-combinator for all functions.

9.4 With the Y-combinator we can reduce the number of arguments on, which a function has to recur. For example member can be rewritten as:

```
(define member-Y
  (lambda (a l)
    ((Y (lambda (recfun)
          (lambda (l)
            (cond
              ((null? l) nil)
              (t (or
                   (eq? (car l) a)
                   (recfun (cdr l))))))))
     l)))
```

Step through the application (member-Y a l) where a is `x` and l is `(y x)`. Rewrite the functions rember, insertR, and subst2 from Chapter 3 in a similar manner.

9.5 In Exercises 6.7 through 6.10 we saw how to use the accumulator technique. Instead of accumulators, continuation functions are sometimes used. These functions abstract what needs to be done to complete an application. For example, multisubst can be defined as:

```
(define multisubst-k
  (lambda (new old lat k)
    (cond
      ((null? lat) (k (quote ( ))))
      ((eq? (car lat) old)
       (multisubst-k new old (cdr lat)
         (lambda (d)
           (k (cons new d)))))
      (t (multisubst-k new old (cdr lat)
           (lambda (d)
             (k (cons (car lat) d))))))))
```

The initial continuation function k is always the function (**lambda** (x) x). Step through the application of

(multisubst-k *new old lat k*),

where

new is y,
old is x, and
lat is (u v x x y z x).

Compare the steps to the application of multisubst to the same arguments. Write down the things you have to do when you return from a recursive application, and, next to it, write down the corresponding continuation function.

9.6 In Chapter 4 and Exercise 4.2 you wrote addvec and multvec. Abstract the two functions into a single function accum. Write the functions length and occur using accum.

9.7 In Exercise 7.3 you wrote the four functions count-op, count-+, count-×, and count-↑. Abstract them into a single function count-op-f, which generates the corresponding functions if passed an appropriate help function.

9.8 Functions of no arguments are called *thunks*. If f is a thunk, it can be evaluated with (f). Consider the following version of or as a function.

```
(define or-func
  (lambda (or1 or2)
    (or (or1) (or2))))
```

Assuming that *or1* and *or2* are always thunks, convince yourself that (**or** ...) and or-func are equivalent. Consider as an example

(**or** (null? *l*) (atom? (car *l*)))

and the corresponding application

```
(or-func
  (lambda ( ) (null? l))
  (lambda ( ) (atom? (car l)))),
```

where

l is ( ).

Write set-f? to take or-func and and-func. Write the functions intersect? and subset? with this set-f? function.

9.9 When you build a pair with an S-expression and a thunk (see Exercise 9.8) you get a *stream*. There are two functions defined on streams: first$ and second$.

Note: In practice, you can actually cons an S-expression directly onto a function. We prefer to stay with the less general cons function.

```
(define first$ first)
```

```
(define second$
  (lambda (str)
    ((second str))))
```

An example of a stream is (build 1 (**lambda** ( ) 2)). Let's call this stream *s*. (first$ *s*) is then 1, and (second$ *s*) is 2. Streams are interesting because they can be used to represent *unbounded* collections such as the integers. Consider the following definitions.

Str-maker is a function that takes a number *n* and a function *next* and produces a stream:

```
(define str-maker
  (lambda (next n)
    (build n (lambda ( ) (str-maker next (next
n))))))
```

With str-maker we can now define the stream of *all* integers like this:

```
(define int (str-maker add1 0))
```

Or we can define the stream of *all* even numbers:

```
(define even (str-maker (lambda (n) (+ 2 n)) 0))
```

With the function frontier we can obtain a finite piece of a stream in a list:

```
(define frontier
  (lambda (str n)
    (cond
      ((zero? n) (quote ( )))
      (t (cons (first$ str) (frontier (second$ str) (sub1 n)))))))
```

What is (frontier int 10)? (frontier int 100)? (frontier even 23)?
Define the stream of odd numbers.

9.10 This exercise builds on the results of Exercise 9.9. Consider the following functions:

```
(define Q
  (lambda (str n)
    (cond
      ((zero? (remainder (first$ str) n))
       (Q (second$ str) n))
      (t (build (first$ str)
              (lambda ( )
                (Q (second$ str) n)))))))
```

```
(define P
  (lambda (str)
    (build (first$ str) (lambda ( ) (P (Q str (first$ str)))))))
```

They can be used to construct streams. What is the result of

(frontier (P (second$ (second$ int))) 10)?

What is this stream of numbers? (See Exercise 4.9 for the definition of remainder.)

Exercises

For these exercises,

e1 is
```
((lambda (x)
   (cond
     ((atom? x) (quote done))
     ((null? x) (quote almost))
     (t (quote never))))
 (quote ____)),
```
e2 is
```
(((lambda (x y)
    (lambda (u)
      (cond
        (u x)
        (t y))))
  1 ( ))
 nil),
```
e3 is
```
((lambda (x)
   ((lambda (x)
      (add1 x))
    (add1 4)))
 6),
```
e4 is (3 (quote a) (quote b)),

e5 is (lambda (lat) (cons (quote lat) lat)),

e6 is (lambda (lat (lyst)) a (quote b)).

10.1 Make up examples for *e* and step through (value *e*). The examples should cover truth values, numbers, and quoted S-expressions.

10.2 Make up some S-expressions, plug them into the ____ of *e1*, and step through the application of (value *e1*).

10.3 Step through the application of (value *e2*). How many closures are produced during the application?

10.4 Consider the expression *e3*. What do you expect to be the value of *e3*? Which of the three x's are "related"? Verify your answers by stepping through (value *e3*). Observe to which x we add one.

10.5 Design a representation for closures and primitives such that the tags (i.e., primitive and non-primitive) at the beginning of the lists become unnecessary. Rewrite the functions that are knowledgeable of the structures. Step through (value *e3*) with the new interpreter.

10.6 Just as the table for predetermined identifiers, initial-table, all tables in our interpreter can be represented as functions. Then, the function extend-table is changed to:

```
(define extend-table
  (lambda (entry table)
    (lambda (name)
      (cond
        ((member? name (first entry))
         (pick (index name (first entry))
           (second entry)))
        (t (table name))))))
```

(For pick see Chapter 4; for index see Exercise 4.5.) What else has to be changed to make the interpreter work? Make the least number of changes. Make up an application of value to your favorite expression and step through it to make sure you understand the new representation. Hint: Look at all the places where tables are used to find out where changes have to be made.

10.7 Write the function ∗lambda?, which checks whether an S-expression is really a representation of a lambda-function.

Example: (∗lambda? *e5*) is true,
(∗lambda? *e6*) is false,
(∗lambda? *e2*) is false.

Also write the functions ∗quote? and ∗cond?, which do the same for quote- and cond-expressions.

10.8 Non-primitive functions are represented by lists in our interpreter. An alternative is to use functions to represent functions. For this we change *lambda to:

```
(define *lambda
  (lambda (e table)
    (build
      (quote non-primitive)
      (lambda (vals)
        (meaning (body-of e)
          (extend-table
            (new-entry (formals-of e) vals)
            table))))))
```

How do we have to change apply-closure to make this representation work? Do we need to change anything else? Walk through the application (value *e2*) to become familiar with this new representation.

10.9 Primitive functions are built repeatedly while finding the value of an expression. To see this, step through the application (value *e3*) and count how often the primitive for `add1` is built. However, consider the following table for predetermined identifiers:

```
(define initial-table
  ((lambda (add1)
     (lambda (name)
       (cond
         ((eq? name (quote t)) t)
         ((eq? name (quote nil)) nil)
         ((eq? name (quote add1)) add1)
         (t (build (quote primitive)
name)))))
   (build (quote primitive) add1)))
```

Using this initial-table, how does the count change? Generalize this approach to include all primitives.

10.10 In Exercise 2.4 we introduced the (**if** ...)-form. We saw that (**if** ...) and (**cond** ...) are interchangeable. If we replace the function *cond by *if where

```
(define *if
  (lambda (e table)
    (if (meaning (test-pt e) table)
        (meaning (then-pt e) table)
        (meaning (else-pt e) table))))
```

we can almost evaluate functions containing (**if** ...). What other changes do we have to make? Make the changes. Take all the examples from this chapter that contain a (**cond** ...), rewrite them with (**if** ...), and step through the modified interpreter. Do the same for *e1* and *e2*.