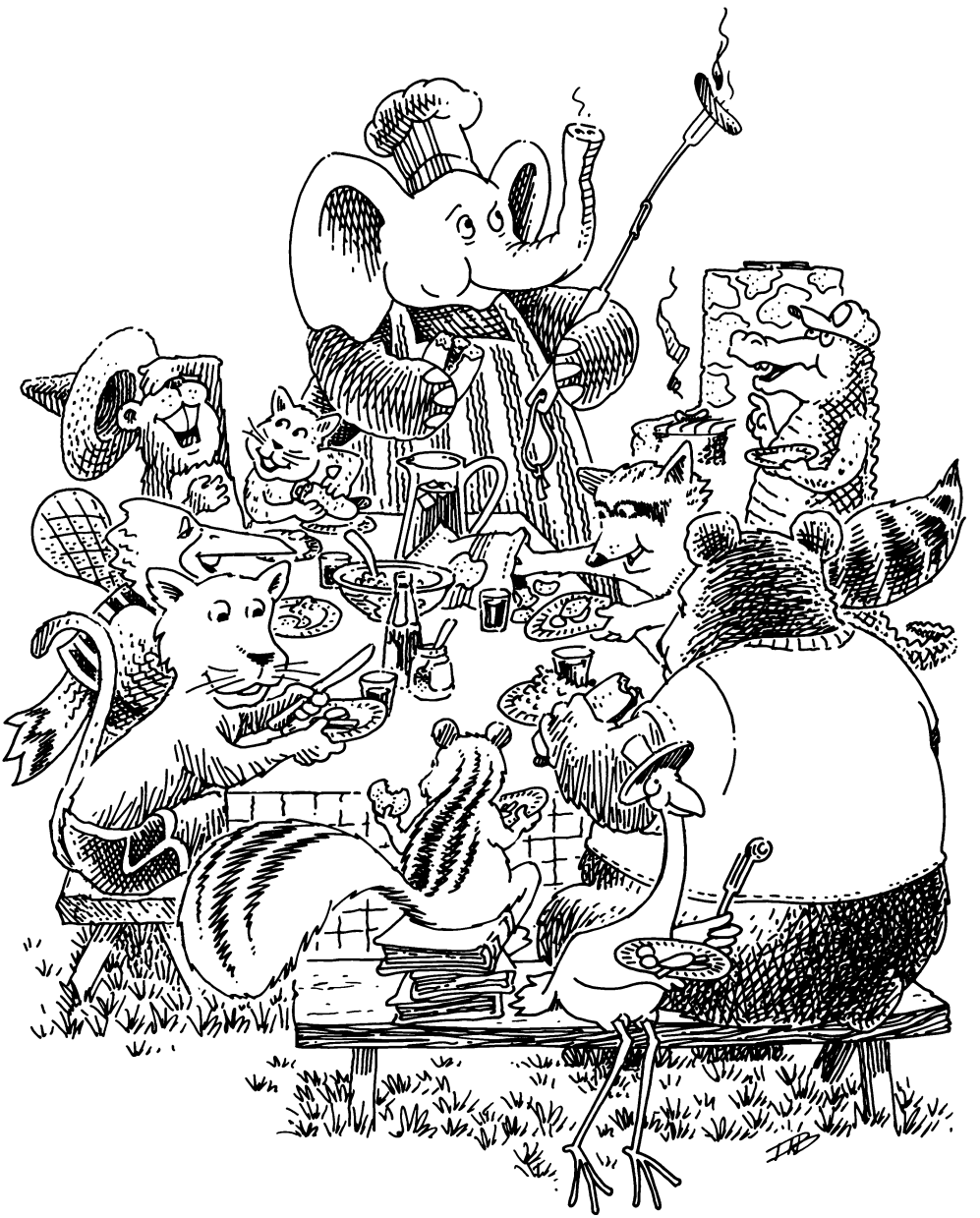
7.
Friends and Relations



Is this a set?
(apple peaches apple plum)

No, since apple appears more than once.

True or false: (*set? lat*)
where
lat is (apples peaches pears plums)

#t, because no atom appears more than once.

How about (*set? lat*)
where
lat is ()

#t, because no atom appears more than once.

Try to write *set?*

```
(define set?
  (lambda (lat)
    (cond
      ((null? lat) #t)
      (else
        (cond
          ((member? (car lat) (cdr lat))
           #f)
          (else (set? (cdr lat))))))))
```

Simplify *set?*

```
(define set?
  (lambda (lat)
    (cond
      ((null? lat) #t)
      ((member? (car lat) (cdr lat)) #f)
      (else (set? (cdr lat))))))
```

Does this work for the example
(apple 3 pear 4 9 apple 3 4)

Yes, since *member?* is now written using *equal?* instead of *eq?*.

Were you surprised to see the function *member?* appear in the definition of *set?*

You should not be, because we have written *member?* already, and now we can use it whenever we want.

What is (*makeset lat*)
where
lat is (apple peach pear peach
plum apple lemon peach)

(apple peach pear plum lemon).

Try to write *makeset* using *member?*

```
(define makeset
  (lambda (lat)
    (cond
      ((null? lat) (quote ()))
      ((member? (car lat) (cdr lat))
       (makeset (cdr lat)))
      (else (cons (car lat)
                  (makeset (cdr lat)))))))
```

Are you surprised to see how short this is?

We hope so. But don't be afraid: it's right.

Using the previous definition, what is the result of (*makeset lat*)
where
lat is (apple peach pear peach
plum apple lemon peach)

(pear plum apple lemon peach).

Try to write *makeset* using *multirember*

```
(define makeset
  (lambda (lat)
    (cond
      ((null? lat) (quote ()))
      (else (cons (car lat)
                  (makeset
                    (multirember (car lat)
                      (cdr lat))))))))
```

What is the result of (*makeset lat*) using this second definition
where
lat is (apple peach pear peach
plum apple lemon peach)

(apple peach pear plum lemon).

Describe in your own words how the second definition of *makeset* works.

Here are our words:
"The function *makeset* remembers to *cons* the first atom in the lat onto the result of the natural recursion, after removing all occurrences of the first atom from the rest of the lat."

Does the second *makeset* work for the example
(apple 3 pear 4 9 apple 3 4)

Yes, since *multirember* is now written using *equal?* instead of *eq?*.

What is (*subset? set1 set2*)
where
set1 is (5 chicken wings)
and
set2 is (5 hamburgers
2 pieces fried chicken and
light duckling wings)

#t, because each atom in *set1* is also in *set2*.

What is (*subset? set1 set2*)
where
set1 is (4 pounds of horseradish)
and
set2 is (four pounds chicken and
5 ounces horseradish)

#f.

Write *subset?*

```
(define subset?
  (lambda (set1 set2)
    (cond
      ((null? set1) #t)
      (else (cond
              ((member? (car set1) set2)
               (subset? (cdr set1) set2))
              (else #f))))))
```

Can you write a shorter version of *subset?*

```
(define subset?
  (lambda (set1 set2)
    (cond
      ((null? set1) #t)
      ((member? (car set1) set2)
       (subset? (cdr set1) set2))
      (else #f))))
```

Try to write *subset?* with (**and** ...)

```
(define subset?
  (lambda (set1 set2)
    (cond
      ((null? set1) #t)
      (else
        (and (member? (car set1) set2)
             (subset? (cdr set1) set2))))))
```

What is (*eqset? set1 set2*)
where
set1 is (6 large chickens with wings)
and
set2 is (6 chickens with large wings)

#t.

Write *eqset?*

```
(define eqset?
  (lambda (set1 set2)
    (cond
      ((subset? set1 set2)
       (subset? set2 set1))
      (else #f))))
```

Can you write *eqset?* with only one **cond**-line?

```
(define eqset?
  (lambda (set1 set2)
    (cond
      (else (and (subset? set1 set2)
                 (subset? set2 set1))))))
```

Write the one-liner.

```
(define eqset?
  (lambda (set1 set2)
    (and (subset? set1 set2)
         (subset? set2 set1))))
```

What is (*intersect? set1 set2*)
where
set1 is (stewed tomatoes and macaroni)
and
set2 is (macaroni and cheese)

#t,
because at least one atom in *set1* is in *set2*.

Define the function *intersect?*

```
(define intersect?
  (lambda (set1 set2)
    (cond
      ((null? set1) #f)
      (else
        (cond
          ((member? (car set1) set2) #t)
          (else (intersect?
                  (cdr set1) set2)))))))
```

Write the shorter version.

```
(define intersect?
  (lambda (set1 set2)
    (cond
      ((null? set1) #f)
      ((member? (car set1) set2) #t)
      (else (intersect? (cdr set1) set2)))))
```

Try writing *intersect?* with (**or** ...)

```
(define intersect?
  (lambda (set1 set2)
    (cond
      ((null? set1) #f)
      (else (or (member? (car set1) set2)
                (intersect?
                  (cdr set1) set2))))))
```

Compare *subset?* and *intersect?*.

What is (*intersect set1 set2*)
where
  set1 is (stewed tomatoes and macaroni)
and
  set2 is (macaroni and cheese)

(and macaroni).

Now you can write the short version of *intersect*

```
(define intersect
  (lambda (set1 set2)
    (cond
      ((null? set1) (quote ()))
      ((member? (car set1) set2)
       (cons (car set1)
         (intersect (cdr set1) set2)))
      (else (intersect (cdr set1) set2)))))
```

What is (*union set1 set2*)
where
  set1 is (stewed tomatoes and
    macaroni casserole)
and
  set2 is (macaroni and cheese)

(stewed tomatoes casserole macaroni
and cheese)

Write *union*

```
(define union
  (lambda (set1 set2)
    (cond
      ((null? set1) set2)
      ((member? (car set1) set2)
       (union (cdr set1) set2))
      (else (cons (car set1)
              (union (cdr set1) set2))))))
```

What is this function?

```
(define xxx
  (lambda (set1 set2)
    (cond
      ((null? set1) (quote ()))
      ((member? (car set1) set2)
       (xxx (cdr set1) set2))
      (else (cons (car set1)
                  (xxx (cdr set1) set2))))))
```

In our words:
"It is a function that returns all the atoms in *set1* that are not in *set2*."
That is, *xxx* is the (set) difference function.

What is (*intersectall l-set*)
where
l-set is ((a b c) (c a d e) (e f g h a b))

(a).

What is (*intersectall l-set*)
where
l-set is ((6 pears and)
(3 peaches and 6 peppers)
(8 pears and 6 plums)
(and 6 prunes with some apples))

(6 and).

Now, using whatever help functions you need, write *intersectall* assuming that the list of sets is non-empty.

```
(define intersectall
  (lambda (l-set)
    (cond
      ((null? (cdr l-set)) (car l-set))
      (else (intersect (car l-set)
                       (intersectall (cdr l-set)))))))
```

Is this a pair?[1]
(pear pear)

Yes, because it is a list with only two atoms.

[1] A pair in Scheme (or Lisp) is a different but related object.

Is this a pair? (3 7)	Yes.
Is this a pair? ((2) (pair))	Yes, because it is a list with only two S-expressions.
(*a-pair? l*) where *l* is (full (house))	#t, because it is a list with only two S-expressions.
Define *a-pair?*	(**define** *a-pair?* (**lambda** (*x*) (**cond** ((*atom? x*) #f) ((*null? x*) #f) ((*null?* (*cdr x*)) #f) ((*null?* (*cdr* (*cdr x*))) #t) (**else** #f))))
How can you refer to the first S-expression of a pair?	By taking the *car* of the pair.
How can you refer to the second S-expression of a pair?	By taking the *car* of the *cdr* of the pair.
How can you build a pair with two atoms?	You *cons* the first one onto the *cons* of the second one onto (). That is, (*cons x1* (*cons x2* (**quote** ()))).
How can you build a pair with two S-expressions?	You *cons* the first one onto the *cons* of the second one onto (). That is, (*cons x1* (*cons x2* (**quote** ()))).
Did you notice the differences between the last two answers?	No, there aren't any.

```
(define first
  (lambda (p)
    (cond
      (else (car p)))))
```

```
(define second
  (lambda (p)
    (cond
      (else (car (cdr p))))))
```

```
(define build
  (lambda (s1 s2)
    (cond
      (else (cons s1
                  (cons s2 (quote ()))))))))
```

What possible uses do these three functions have?

They are used to make representations of pairs and to get parts of representations of pairs. See chapter 6.

They will be used to improve readability, as you will soon see.

Redefine *first*, *second*, and *build* as one-liners.

Can you write *third* as a one-liner?

```
(define third
  (lambda (l)
    (car (cdr (cdr l)))))
```

Is l a rel where
l is (apples peaches pumpkin pie)

No, since l is not a list of pairs. We use rel to stand for relation.

Is l a rel where
l is ((apples peaches)
(pumpkin pie)
(apples peaches))

No, since l is not a set of pairs.

Is l a rel where
l is ((apples peaches) (pumpkin pie))

Yes.

Is l a rel where
l is ((4 3) (4 2) (7 6) (6 2) (3 4))

Yes.

Is *rel* a fun
where
rel is ((4 3) (4 2) (7 6) (6 2) (3 4))

No. We use fun to stand for function.

What is (*fun? rel*)
where
rel is ((8 3) (4 2) (7 6) (6 2) (3 4))

#t, because (*firsts rel*) is a set
—See chapter 3.

What is (*fun? rel*)
where
rel is ((d 4) (b 0) (b 9) (e 5) (g 4))

#f, because b is repeated.

Write *fun?* with *set?* and *firsts*

```
(define fun?
  (lambda (rel)
    (set? (firsts rel))))
```

Is *fun?* a simple one-liner?

It sure is.

How do we represent a finite function?

For us, a finite function is a list of pairs in which no first element of any pair is the same as any other first element.

What is (*revrel rel*)
where
rel is ((8 a) (pumpkin pie) (got sick))

((a 8) (pie pumpkin) (sick got)).

You can now write *revrel*

```
(define revrel
  (lambda (rel)
    (cond
      ((null? rel) (quote ()))
      (else (cons (build
                    (second (car rel))
                    (first (car rel)))
                  (revrel (cdr rel)))))))
```

Would the following also be correct:

```
(define revrel
  (lambda (rel)
    (cond
      ((null? rel) (quote ()))
      (else (cons (cons
                    (car (cdr (car rel)))
                    (cons (car (car rel))
                      (quote ())))
                  (revrel (cdr rel)))))))
```

Yes, but now do you see how representation aids readability?

Suppose we had the function *revpair* that reversed the two components of a pair like this:

```
(define revpair
  (lambda (pair)
    (build (second pair) (first pair))))
```

How would you rewrite *revrel* to use this help function?

No problem, and it is even easier to read:

```
(define revrel
  (lambda (rel)
    (cond
      ((null? rel) (quote ()))
      (else (cons (revpair (car rel))
                  (revrel (cdr rel)))))))
```

Can you guess why *fun* is not a fullfun where
fun is ((8 3) (4 2) (7 6) (6 2) (3 4))

fun is not a fullfun, since the 2 appears more than once as a second item of a pair.

Why is #t the value of (*fullfun? fun*) where
fun is ((8 3) (4 8) (7 6) (6 2) (3 4))

Because (3 8 6 2 4) is a set.

What is (*fullfun? fun*) where
fun is ((grape raisin)
(plum prune)
(stewed prune))

#f.

What is (*fullfun? fun*)
where
fun is ((grape raisin)
(plum prune)
(stewed grape))

#t, because (raisin prune grape) is a set.

Define *fullfun?*

```
(define fullfun?
  (lambda (fun)
    (set? (seconds fun))))
```

Can you define *seconds*

It is just like *firsts*.

What is another name for *fullfun?*

one-to-one?.

Can you think of a second way to write *one-to-one?*

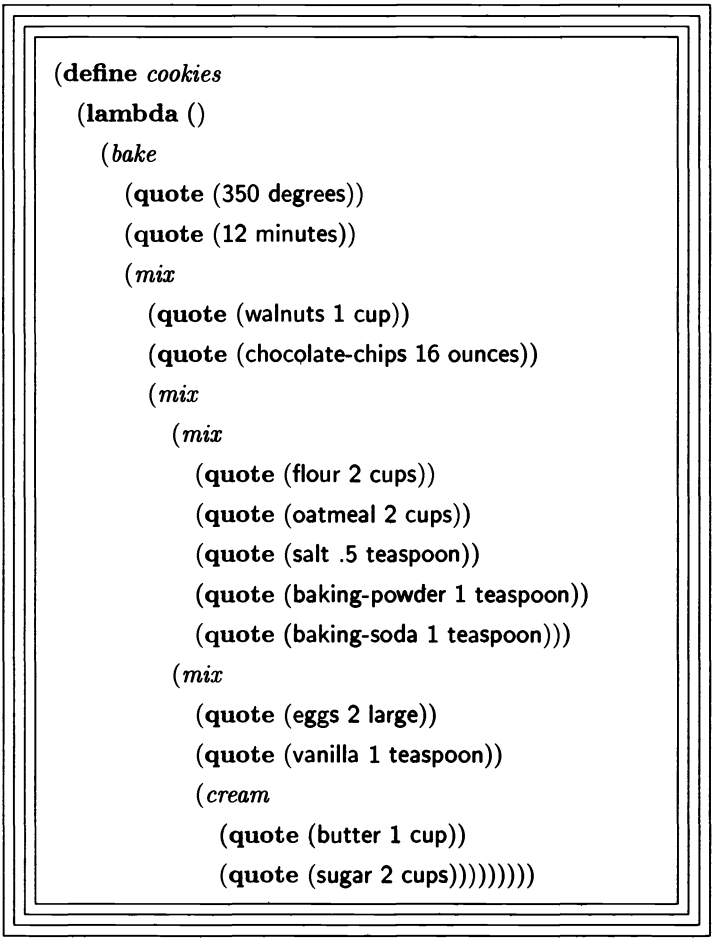
```
(define one-to-one?
  (lambda (fun)
    (fun? (revrel fun))))
```

Is ((chocolate chip) (doughy cookie)) a one-to-one function?

Yes, and you deserve one now!

Go and get one!

Or better yet, make your own.



```
(define cookies
  (lambda ()
    (bake
      (quote (350 degrees))
      (quote (12 minutes))
      (mix
        (quote (walnuts 1 cup))
        (quote (chocolate-chips 16 ounces))
        (mix
          (mix
            (quote (flour 2 cups))
            (quote (oatmeal 2 cups))
            (quote (salt .5 teaspoon))
            (quote (baking-powder 1 teaspoon))
            (quote (baking-soda 1 teaspoon)))
          (mix
            (quote (eggs 2 large))
            (quote (vanilla 1 teaspoon))
            (cream
              (quote (butter 1 cup))
              (quote (sugar 2 cups)))))))))
```