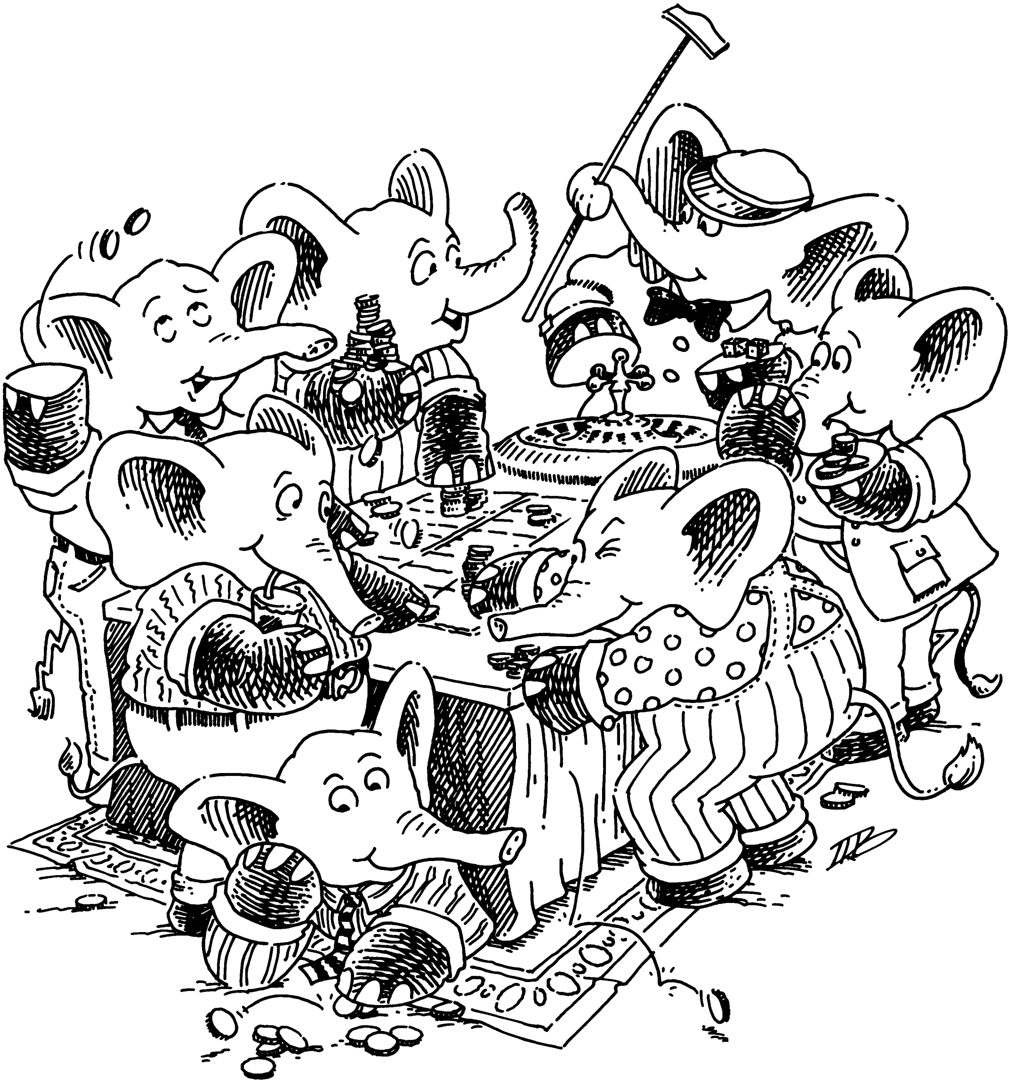
4. Numbers Games



Is 14 an atom?

Yes, because all numbers are atoms.

Is (*atom? n*) true or false
where
n is 14

True, because 14 is an atom.

Is −3 a number?

Yes,
but we do not consider negative numbers.

Is 3.14159 a number?

Yes,
but we consider only whole numbers.

Are −3 and 3.14159 numbers?

Yes,
but the only numbers we use are the nonnegative integers (i.e., 0, 1, 2, 3, 4, ...).

What is (*add1*[1] *n*)
where *n* is 67

[1] L: `1+`
S:
```
(define add1
  (lambda (n)
    (+ n 1)))
```

68.

What is (*add1* 67)

Also 68,
because we don't need to say "where *n* is 67" when the argument is a number.

What is (*sub1*[1] *n*)
where *n* is 5

[1] L: `1-`
S:
```
(define sub1
  (lambda (n)
    (- n 1)))
```

4.

What is (*sub1* 0)

No answer.[1]

[1] (*sub1 n*), where *n* is 0, has no answer because we consider only nonnegative numbers. In practice, this result is -1.

Is (*zero?*[1] 0) true or false?

True.

[1] L: zerop

Is (*zero?* 1492) true or false?

False.

What is (✚ 46 12)

58.

Try to write the function ✚
Hint: It uses *zero? add1*[1] and *sub1*[1]

[1] Remember to use our definitions for add1 and sub1.

```
(define ✚ [1]
  (lambda (n m)
    (cond
      ((zero? m) n)
      (else (add1 (✚ n (sub1 m)))))))
```

Wasn't that easy?

[1] L, S: This is like +. Write it as o+ (see preface).

But didn't we just violate The First Commandment?

Yes, but we can treat *zero?* like *null?* since *zero?* asks if a number is empty and *null?* asks if a list is empty.

If *zero?* is like *null?*
is *add1* like *cons*

Yes! *cons* builds lists and *add1* builds numbers.

What is (− 14 3)

11.

What is (− 17 9)

8.

What is (− 18 25)

No answer. There are no negative numbers.

Try to write the function −
Hint: Use *sub1*

How about this:

```
(define −[1]
  (lambda (n m)
    (cond
      ((zero? m) n)
      (else (sub1 (− n (sub1 m)))))))
```

[1] L, S: This is like -. Write it as o- (see preface).

Can you describe how (− *n* *m*) works?

It takes two numbers as arguments, and reduces the second until it hits zero. It subtracts one from the result as many times as it did to cause the second one to reach zero.

Is this a tup?
(2 11 3 79 47 6)

Yes: tup is short for tuple.

Is this a tup?
(8 55 5 555)

Yes, of course, it is also a list of numbers.

Is this a tup?
(1 2 8 apple 4 3)

No, it is just a list of atoms.

Is this a tup?
(3 (7 4) 13 9)

No, because it is not a list of numbers. (7 4) is not a number.

Is this a tup? ()	Yes, it is a list of zero numbers. This special case is the empty tup.
What is (*addtup tup*) where *tup* is (3 5 2 8)	18.
What is (*addtup tup*) where *tup* is (15 6 7 12 3)	43.
What does *addtup* do?	It builds a number by totaling all the numbers in its argument.
What is the natural way to build numbers from a list?	Use + in place of *cons*: + builds numbers in the same way as *cons* builds lists.
When building lists with *cons* the value of the terminal condition is () What should be the value of the terminal condition when building numbers with +	0.
What is the natural terminal condition for a list?	(*null? l*).
What is the natural terminal condition for a tup?	(*null? tup*).
When we build a number from a list of numbers, what should the terminal condition line look like?	((*null? tup*) 0), just as ((*null? l*) (**quote** ())) is often the terminal condition line for lists.
What is the terminal condition line of *addtup*	((*null? tup*) 0).

How is a lat defined?	It is either an empty list, or it contains an atom, (*car lat*), and a rest, (*cdr lat*), that is also a lat.
How is a tup defined?	It is either an empty list, or it contains a number and a rest that is also a tup.
What is used in the natural recursion on a list?	(*cdr lat*).
What is used in the natural recursion on a tup?	(*cdr tup*).
Why?	Because the rest of a non-empty list is a list and the rest of a non-empty tup is a tup.
How many questions do we need to ask about a list?	Two.
How many questions do we need to ask about a tup?	Two, because it is either empty or it is a number and a rest, which is again a tup.
How is a number defined?	It is either zero or it is one added to a rest, where rest is again a number.
What is the natural terminal condition for numbers?	(*zero? n*).
What is the natural recursion on a number?	(*sub1 n*).
How many questions do we need to ask about a number?	Two.

The First Commandment

(*first revision*)

When recurring on a list of atoms, *lat*, ask two questions about it: (*null? lat*) and else.
When recurring on a number, *n*, ask two questions about it: (*zero? n*) and else.

What does *cons* do?

It builds lists.

What does *addtup* do?

It builds a number by totaling all the numbers in a tup.

What is the terminal condition line of *addtup*

((*null? tup*) 0).

What is the natural recursion for *addtup*

(*addtup* (*cdr tup*)).

What does *addtup* use to build a number?

It uses +, because + builds numbers, too!

Fill in the dots in the following definition:

```
(define addtup
  (lambda (tup)
    (cond
      ((null? tup) 0)
      (else ...))))
```

Here is what we filled in:
(+ (*car tup*) (*addtup* (*cdr tup*))).
Notice the similarity between this line, and the last line of the function *rember*:
(*cons* (*car lat*) (*rember a* (*cdr lat*))).

What is (× 5 3)

15.

What is (× 13 4)

52.

What does (× *n m*) do?

It builds up a number by adding *n* up *m* times.

What is the terminal condition line for ×

((*zero? m*) 0), because $n \times 0 = 0$.

Since (*zero? m*) is the terminal condition, *m* must eventually be reduced to zero. What function is used to do this?

sub1.

The Fourth Commandment

(*first revision*)

Always change at least one argument while recurring. It must be changed to be closer to termination. The changing argument must be tested in the termination condition:

when using *cdr*, test termination with *null?* and
when using *sub1*, test termination with *zero?*.

What is another name for (× *n* (*sub1 m*)) in this case?

It's the natural recursion for ×.

Try to write the function ×

```
(define ×[1]
  (lambda (n m)
    (cond
      ((zero? m) 0)
      (else (+ n (× n (sub1 m)))))))
```

[1] L, S: This is like *.

What is (× 12 3)	36, but let's follow through the function one time to see how we get this value.
(*zero?* *m*)	No.
What is the meaning of (+ *n* (× *n* (*sub1* *m*)))	It adds *n* (where *n* = 12) to the natural recursion. If × is correct then (× 12 (*sub1* 3)) should be 24.
What are the new arguments of (× *n* *m*)	*n* is 12, and *m* is 2.
(*zero?* *m*)	No.
What is the meaning of (+ *n* (× *n* (*sub1* *m*)))	It adds *n* (where *n* = 12) to (× *n* (*sub1* *m*)).
What are the new arguments of (× *n* *m*)	*n* is 12, and *m* is 1.
(*zero?* *m*)	No.
What is the meaning of (+ *n* (× *n* (*sub1* *m*)))	It adds *n* (where *n* = 12) to (× *n* (*sub1* *m*)).
What is the value of the line ((*zero?* *m*) 0)	0, because (*zero?* *m*) is now true.
Are we finished yet?	No.

Why not?

Because we still have three ✚es to pick up.

What is the value of the original application?

Add 12 to 12 to 12 to 0 yielding 36, Notice that n has been ✚ed m times.

Argue, using equations, that × is the conventional multiplication of nonnegative integers, where n is 12 and m is 3.

$$\begin{aligned}(\times\ 12\ 3) &= 12 + (\times\ 12\ 2)\\ &= 12 + 12 + (\times\ 12\ 1)\\ &= 12 + 12 + 12 + (\times\ 12\ 0)\\ &= 12 + 12 + 12 + 0,\end{aligned}$$

which is as we expected. This technique works for all recursive functions, not just those that use numbers. You can use this approach to write functions as well as to argue their correctness.

Again, why is 0 the value for the terminal condition line in ×

Because 0 will not affect +. That is, $n + 0 = n$.

The Fifth Commandment

When building a value with ✚ , always use 0 for the value of the terminating line, for adding 0 does not change the value of an addition.

When building a value with ×, always use 1 for the value of the terminating line, for multiplying by 1 does not change the value of a multiplication.

When building a value with *cons*, always consider () for the value of the terminating line.

What is (*tup+ tup1 tup2*)
where
tup1 is (3 6 9 11 4)
and
tup2 is (8 5 2 0 7)

(11 11 11 11 11).

What is (*tup*+ *tup1* *tup2*) where *tup1* is (2 3) and *tup2* is (4 6)	(6 9).
What does (*tup*+ *tup1* *tup2*) do?	It adds the first number of *tup1* to the first number of *tup2*, then it adds the second number of *tup1* to the second number of *tup2*, and so on, building a tup of the answers, for tups of the same length.
What is unusual about *tup*+	It looks at each element of two tups at the same time, or in other words, it recurs on two tups.
If you recur on one tup how many questions do you have to ask?	Two, they are (*null? tup*) and **else**.
When recurring on two tups, how many questions need to be asked about the tups?	Four: if the first tup is empty or non-empty, and if the second tup is empty or non-empty.

Do you mean the questions
  (**and** (*null? tup1*) (*null? tup2*))
  (*null? tup1*)
  (*null? tup2*)
and
  else
	Yes.

Can the first *tup* be () at the same time as the second is other than ()	No, because the tups must have the same length.
Does this mean (**and** (*null? tup1*) (*null? tup2*)) and **else** are the only questions we need to ask?	Yes, because (*null? tup1*) is true exactly when (*null? tup2*) is true.

Write the function *tup+*

```
(define tup+
  (lambda (tup1 tup2)
    (cond
      ((and (null? tup1) (null? tup2))
       (quote ()))
      (else
        (cons (✚ (car tup1) (car tup2))
          (tup+
            (cdr tup1) (cdr tup2)))))))
```

What are the arguments of ✚ in the last line?

(*car tup1*) and (*car tup2*).

What are the arguments of *cons* in the last line?

(✚ (*car tup1*) (*car tup2*)) and (*tup+* (*cdr tup1*) (*cdr tup2*)).

What is (*tup+ tup1 tup2*)
where
  tup1 is (3 7)
and
  tup2 is (4 6)

(7 13).
  But let's see how it works.

(*null? tup1*)

No.

```
(cons
  (✚ (car tup1) (car tup2))
  (tup+ (cdr tup1) (cdr tup2)))
```

cons 7 onto the natural recursion:
  (*tup+* (*cdr tup1*) (*cdr tup2*)).

Why does the natural recursion include the *cdr* of both arguments?

Because the typical element of the final value uses the *car* of both tups, so now we are ready to consider the rest of both tups.

(*null? tup1*)
where
  tup1 is now (7)
and
  tup2 is now (6)

No.

(*cons*
  (+ (*car tup1*) (*car tup2*))
  (*tup*+ (*cdr tup1*) (*cdr tup2*)))
	cons 13 onto the natural recursion.

(*null? tup1*)	Yes.
Then, what must be the value?	(), because (*null? tup2*) must be true.
What is the value of the application?	(7 13). That is, the *cons* of 7 onto the *cons* of 13 onto ().
What problem arises when we want (*tup*+ *tup1 tup2*) where *tup1* is (3 7) and *tup2* is (4 6 8 1)	No answer, since *tup1* will become null before *tup2*. See The First Commandment: We did not ask all the necessary questions! But, we would like the final value to be (7 13 8 1).
Can we still write *tup*+ even if the tups are not the same length?	Yes!
What new terminal condition line can we add to get the correct final value?	Add ((*null? tup1*) *tup2*).
What is (*tup*+ *tup1 tup2*) where *tup1* is (3 7 8 1) and *tup2* is (4 6)	No answer, since *tup2* will become null before *tup1*. See The First Commandment: We did not ask all the necessary questions!
What do we need to include in our function?	We need to ask two more questions: (*null? tup1*) and (*null? tup2*).
What does the second new line look like?	((*null? tup2*) *tup1*).

Here is a definition of *tup+* that works for any two tups:

```
(define tup+
  (lambda (tup1 tup2)
    (cond
      ((and (null? tup1) (null? tup2))
       (quote ()))
      ((null? tup1) tup2)
      ((null? tup2) tup1)
      (else
        (cons (+ (car tup1) (car tup2))
          (tup+
            (cdr tup1) (cdr tup2)))))))
```

Can you simplify it?

```
(define tup+
  (lambda (tup1 tup2)
    (cond
      ((null? tup1) tup2)
      ((null? tup2) tup1)
      (else
        (cons (+ (car tup1) (car tup2))
          (tup+
            (cdr tup1) (cdr tup2)))))))
```

Does the order of the two terminal conditions matter?

No.

Is **else** the last question?

Yes, because either (*null? tup1*) or (*null? tup2*) is true if either one of them does not contain at least one number.

What is (> 12 133)

#f—false.

What is (> 120 11)

#t—true.

On how many numbers do we have to recur?

Two, *n* and *m*.

How do we recur?

With (*sub1 n*) and (*sub1 m*).

When do we recur?

When we know neither number is equal to 0.

How many questions do we have to ask about *n* and *m*

Three: (*zero? n*), (*zero? m*), and **else**.

Can you write the function > now using *zero?* and *sub1*

How about

```
(define >
  (lambda (n m)
    (cond
      ((zero? m) #t)
      ((zero? n) #f)
      (else (> (sub1 n) (sub1 m))))))
```

Is the way we wrote (> *n* *m*) correct?

No, try it for the case where *n* and *m* are the same number. Let *n* and *m* be 3.

(*zero?* 3)

No, so move to the next question.

(*zero?* 3)

No, so move to the next question.

What is the meaning of
(> (*sub1* *n*) (*sub1* *m*))

Recur, but with both arguments reduced by one.

(*zero?* 2)

No, so move to the next question.

(*zero?* 2)

No, so move to the next question.

What is the meaning of
(> (*sub1* *n*) (*sub1* *m*))

Recur, but with both arguments closer to zero by one.

(*zero?* 1)

No, so move to the next question.

(*zero?* 1)

No, so move to the next question.

What is the meaning of
(> (*sub1* *n*) (*sub1* *m*))

Recur, but with both arguments reduced by one.

(*zero?* 0)

Yes, so the value of (> *n* *m*) is #t.

Is this correct?

No, because 3 is not greater than 3.

Does the order of the two terminal conditions matter?

Think about it.

Does the order of the two terminal conditions matter?

Try it out!

Does the order of the two previous answers matter?

Yes. Think first, then try.

How can we change the function > to take care of this subtle problem?

Switch the *zero?* lines:

```
(define >
  (lambda (n m)
    (cond
      ((zero? n) #f)
      ((zero? m) #t)
      (else (> (sub1 n) (sub1 m))))))
```

What is (< 4 6)

#t.

(< 8 3)

#f.

(< 6 6)

#f.

Now try to write <

```
(define <
  (lambda (n m)
    (cond
      ((zero? m) #f)
      ((zero? n) #t)
      (else (< (sub1 n) (sub1 m))))))
```

Here is the definition of =

```
(define =
  (lambda (n m)
    (cond
      ((zero? m) (zero? n))
      ((zero? n) #f)
      (else (= (sub1 n) (sub1 m))))))
```

Rewrite = using < and >

```
(define =
  (lambda (n m)
    (cond
      ((> n m) #f)
      ((< n m) #f)
      (else #t))))
```

Does this mean we have two different functions for testing equality of atoms?

Yes, they are = for atoms that are numbers and *eq?* for the others.

(↑ 1 1)

1.

(↑ 2 3)

8.

(↑ 5 3)

125.

Now write the function ↑
Hint: See the The First and Fifth Commandments.

```
(define ↑[1]
  (lambda (n m)
    (cond
      ((zero? m) 1)
      (else (× n (↑ n (sub1 m)))))))
```

[1] L, S: This is like **expt**.

What is a good name for this function?

```
(define ???
  (lambda (n m)
    (cond
      ((< n m) 0)
      (else (add1 (??? (− n m) m))))))
```

We have never seen this kind of definition before; the natural recursion also looks strange.

What does the first question check?	It determines whether the first argument is less than the second one.
And what happens in the second line?	We recur with a first argument from which we subtract the second argument. When the function returns, we add 1 to the result.
So what does the function do?	It counts how many times the second argument fits into the first one.
And what do we call this?	Division.

```
(define ÷[1]
  (lambda (n m)
    (cond
      ((< n m) 0)
      (else (add1 (÷ (− n m) m))))))
```

1 L: `(defun quotient (n m)`
`       (values (truncate (/ n m))))`
S: This is like `quotient`.

What is (÷ 15 4)	Easy, it is 3.
How do we get there?	Easy, too:

$$\begin{aligned}(\div\ 15\ 4) &= 1 + (\div\ 11\ 4)\\ &= 1 + (1 + (\div\ 7\ 4))\\ &= 1 + (1 + (1 + (\div\ 3\ 4)))\\ &= 1 + (1 + (1 + 0)).\end{aligned}$$

Wouldn't a (ham and cheese on rye) be good right now?

Don't forget the mustard!

What is the value of (*length lat*)
where
lat is (hotdogs with mustard sauerkraut and pickles)

6.

What is (*length lat*)
where
lat is (ham and cheese on rye)

5.

Now try to write the function *length*

```
(define length
  (lambda (lat)
    (cond
      ((null? lat) 0)
      (else (add1 (length (cdr lat)))))))
```

What is (*pick n lat*)
where *n* is 4
and
lat is (lasagna spaghetti ravioli macaroni meatball)

macaroni.

What is (*pick* 0 *lat*)
where *lat* is (a)

No answer.

Try to write the function *pick*

```
(define pick
  (lambda (n lat)
    (cond
      ((zero? (sub1 n)) (car lat))
      (else (pick (sub1 n) (cdr lat))))))
```

What is (*rempick n lat*)
where *n* is 3
and
lat is (hotdogs with hot mustard)

(hotdogs with mustard).

Now try to write *rempick*

```
(define rempick
  (lambda (n lat)
    (cond
      ((zero? (sub1 n)) (cdr lat))
      (else (cons (car lat)
                  (rempick (sub1 n)
                    (cdr lat)))))))
```

Is (*number?*[1] *a*) true or false where *a* is tomato

False.

[1] L: **numberp**

Is (*number?* 76) true or false?

True.

Can you write *number?* which is true if its argument is a numeric atom and false if it is anthing else?

No: *number?*, like *add1*, *sub1*, *zero?*, *car*, *cdr*, *cons*, *null?*, *eq?*, and *atom?*, is a primitive function.

Now using *number?* write the function *no-nums* which gives as a final value a lat obtained by removing all the numbers from the lat. For example,
where
  lat is (5 pears 6 prunes 9 dates)
the value of (*no-nums lat*) is
  (pears prunes dates)

```
(define no-nums
  (lambda (lat)
    (cond
      ((null? lat) (quote ()))
      (else (cond
              ((number? (car lat))
               (no-nums (cdr lat)))
              (else (cons (car lat)
                          (no-nums
                            (cdr lat)))))))))
```

Now write *all-nums* which extracts a tup from a lat using all the numbers in the lat.

```
(define all-nums
  (lambda (lat)
    (cond
      ((null? lat) (quote ()))
      (else
        (cond
          ((number? (car lat))
           (cons (car lat)
             (all-nums (cdr lat))))
          (else (all-nums (cdr lat))))))))
```

Write the function *eqan?* which is true if its two arguments (*a1* and *a2*) are the same atom. Remember to use = for numbers and *eq?* for all other atoms.

```
(define eqan?
  (lambda (a1 a2)
    (cond
      ((and (number? a1) (number? a2))
       (= a1 a2))
      ((or (number? a1) (number? a2))
       #f)
      (else (eq? a1 a2)))))
```

Can we assume that all functions written using *eq?* can be generalized by replacing *eq?* by *eqan?*

Yes, except, of course, for *eqan?* itself.

Now write the function *occur* which counts the number of times an atom *a* appears in a *lat*

```
(define occur
  (lambda (a lat)
    (cond
      (________ ________)
      (else
        (cond
          (________ ________)
          (________ ________))))))
```

```
(define occur
  (lambda (a lat)
    (cond
      ((null? lat) 0)
      (else
        (cond
          ((eq? (car lat) a)
           (add1 (occur a (cdr lat))))
          (else (occur a (cdr lat))))))))
```

Write the function *one?* where (*one? n*) is #t if *n* is 1 and #f (i.e., false) otherwise.

```
(define one?
  (lambda (n)
    (cond
      ((zero? n) #f)
      (else (zero? (sub1 n))))))
```

or

```
(define one?
  (lambda (n)
    (cond
      (else (= n 1)))))
```

Guess how we can further simplify this function, making it a one-liner.

By removing the (**cond** ...) clause:

```
(define one?
  (lambda (n)
    (= n 1)))
```

Now rewrite the function *rempick* that removes the n^{th} atom from a lat. For example,
where
n is 3
and
lat is (lemon meringue salty pie)
the value of (*rempick n lat*) is
(lemon meringue pie)
Use the function *one?* in your answer.

```
(define rempick
  (lambda (n lat)
    (cond
      ((one? n) (cdr lat))
      (else (cons (car lat)
                  (rempick (sub1 n)
                    (cdr lat)))))))
```