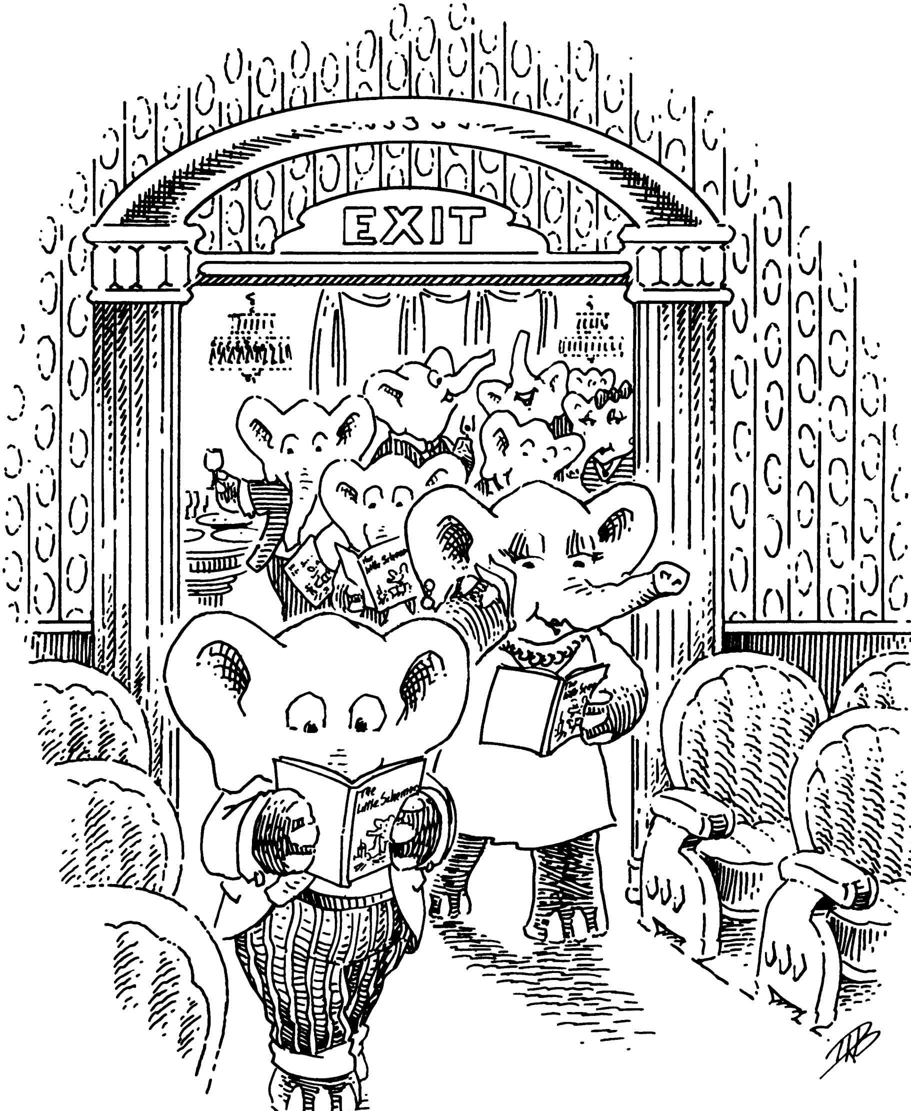
11. Welcome Back to the Show



Welcome back.

It's a pleasure.

Have you read *The Little LISPer*?[1]

[1] Or *The Little Schemer*.

#f.

Are you sure you haven't read *The Little LISPer*?

Well, ...

Do you know about Lambda the Ultimate?

#t.

Are you sure you have read that much of *The Little LISPer*?

Absolutely.[1]

[1] If you are familiar with recursion and know that functions are values, you may continue anyway.

Are you acquainted with *member?*

```
(define member?
  (lambda (a lat)
    (cond
      ((null? lat) #f)
      (else (or (eq? a (car lat))
                (member? a (cdr lat)))))))
```

Sure, *member?* is a good friend.

What is the value of (*member? a lat*)
where *a* is sardines
and
lat is (Italian sardines spaghetti parsley)

#t, but this is not interesting.

What is the value of (*two-in-a-row? lat*)
where
lat is (Italian sardines spaghetti parsley)

#f.

Are *two-in-a-row?* and *member?* related?

Yes, both visit each element of a list of atoms up to some point. One checks whether an atom is in a list, the other checks whether any atom occurs twice in a row.

What is the value of (*two-in-a-row? lat*) where
lat is (Italian sardines sardines spaghetti parsley)

#t.

What is the value of (*two-in-a-row? lat*) where
lat is (Italian sardines more sardines spaghetti)

#f.

Explain precisely what *two-in-a-row?* does.

Easy.
It determines whether any atom occurs twice in a row in a list of atoms.

Is this close to what *two-in-a-row?* should look like?

```
(define two-in-a-row?
  (lambda (lat)
    (cond
      ((null? lat) ...)
      (else ...
           (two-in-a-row? (cdr lat))
           ...))))
```

That looks fine. The dots in the first line should be replaced by #f.

What should we do with the dots in the second line?

We know that there is at least one element in *lat*. We must find out whether the next element in *lat*, if there is one, is identical to this element.

Doesn't this sound like we need a function to do this? Define it.

```
(define is-first?
  (lambda (a lat)
    (cond
      ((null? lat) #f)
      (else (eq? (car lat) a)))))
```

Can we now complete the definition of *two-in-a-row?*

Yes, now we have all the pieces and we just need to put them together:

```
(define two-in-a-row?
  (lambda (lat)
    (cond
      ((null? lat) #f)
      (else
        (or (is-first? (car lat) (cdr lat))
          (two-in-a-row? (cdr lat)))))))
```

There is a different way to accomplish the same task.

We have seen this before: most functions can be defined in more than one way.

What does *two-in-a-row?* do when *is-first?* returns #f

It continues to search for two atoms in a row in the rest of *lat*.

Is it true that (*is-first? a lat*) may respond with #f for two different situations?

Yes, it returns #f when *lat* is empty or when the first element in the list is different from *a*.

In which of the two cases does it make sense for *two-in-a-row?* to continue the search?

In the second one only, because the rest of the list is not empty.

Should we change the definitions of *two-in-a-row?* and *is-first?* in such a way that *two-in-a-row?* leaves the decision of whether continuing the search is useful to the revised version of *is-first?*

That's an interesting idea.

Here is a revised version of *two-in-a-row?*

```
(define two-in-a-row?
  (lambda (lat)
    (cond
      ((null? lat) #f)
      (else
        (is-first-b? (car lat) (cdr lat))))))
```

Can you define the function *is-first-b?* which is like *is-first?* but uses *two-in-a-row?* only when it is useful to resume the search?

That's easy. If *lat* is empty, the value of (*is-first-b? a lat*) is #f. If *lat* is non-empty and if (*eq?* (*car lat*) *a*) is not true, it determines the value of (*two-in-a-row? lat*).

```
(define is-first-b?
  (lambda (a lat)
    (cond
      ((null? lat) #f)
      (else (or (eq? (car lat) a)
                (two-in-a-row? lat))))))
```

Why do we determine the value of (*two-in-a-row? lat*) in *is-first-b?*

If *lat* contains at least one atom and if the atom is not the same as *a*, we must search for two atoms in a row in *lat*. And that's the job of *two-in-a-row?*.

When *is-first-b?* determines the value of (*two-in-a-row? lat*) what does *two-in-a-row?* actually do?

Since *lat* is not empty, it will request the value of (*is-first-b?* (*car lat*) (*cdr lat*)).

Does this mean we could write a function like *is-first-b?* that doesn't use *two-in-a-row?* at all?

Yes, we could. The new function would recur directly instead of through *two-in-a-row?*.

Let's use the name *two-in-a-row-b?* for the new version of *is-first-b?*

That sounds like a good name.

How would *two-in-a-row-b?* recur?

With (*two-in-a-row-b?* (*car lat*) (*cdr lat*)), because that's the way *two-in-a-row?* used *is-first-b?*, and *two-in-a-row-b?* is used in its place now.

So what is *a* when we are asked to determine the value of (*two-in-a-row-b? a lat*)

It is the atom that precedes the atoms in *lat* in the original list.

Can you fill in the dots in the following definition of *two-in-a-row-b?*

```
(define two-in-a-row-b?
  (lambda (preceding lat)
    (cond
      ((null? lat) #f)
      (else ...
            (two-in-a-row-b? (car lat)
              (cdr lat))
            ...))))
```

That's easy. It is just like *is-first?* except that we know what to do when (*car lat*) is not equal to *preceding*:

```
(define two-in-a-row-b?
  (lambda (preceding lat)
    (cond
      ((null? lat) #f)
      (else (or (eq? (car lat) preceding)
                (two-in-a-row-b? (car lat)
                  (cdr lat)))))))
```

What is the natural recursion in *two-in-a-row-b?*

The natural recursion is
(*two-in-a-row-b?* (*car lat*) (*cdr lat*)).

Is this unusual?

Definitely: both arguments change even though the function asks questions about its second argument only.

Why does the first argument to *two-in-a-row-b?* change all the time?

As the name of the argument says, the first argument is always the atom that precedes the current *lat* in the list of atoms that *two-in-a-row?* received.

Now that we have *two-in-a-row-b?* can you define *two-in-a-row?* a final time?

Trivial:

```
(define two-in-a-row?
  (lambda (lat)
    (cond
      ((null? lat) #f)
      (else (two-in-a-row-b? (car lat)
              (cdr lat))))))
```

Let's see one more time how *two-in-a-row?* works.

Okay.

(*two-in-a-row? lat*) where *lat* is (b d e i i a g)	This looks like a good example. Since *lat* is not empty, we need the value of (*two-in-a-row-b? preceding lat*) where *preceding* is b and *lat* is (d e i i a g)
(*null? lat*) where *lat* is (d e i i a g)	#f.
(*eq?* (*car lat*) *preceding*) where *preceding* is b and *lat* is (d e i i a g)	#f, because d is not b.
And now?	Next we need to determine the value of (*two-in-a-row-b? preceding lat*) where *preceding* is d and *lat* is (e i i a g).
Does it stop here?	No, it doesn't. After determining that *lat* is not empty and that (*eq?* (*car lat*) *preceding*) is not true, we must determine the value of (*two-in-a-row-b? preceding lat*) where *preceding* is e and *lat* is (i i a g).
Enough?	Not yet. We also need to determine the value of (*two-in-a-row-b? preceding lat*) where *preceding* is i and *lat* is (i a g).

And?

Now (*eq?* (*car lat*) *preceding*) is true because *preceding* is i and
lat is (i a g).

So what is the value of (*two-in-a-row? lat*) where
lat is (b d e i i a g)

#t.

Do we now understand how *two-in-a-row?* works?

Yes, this is clear.

What is the value of (*sum-of-prefixes tup*) where
tup is (2 1 9 17 0)

(2 3 12 29 29).

(*sum-of-prefixes tup*) where
tup is (1 1 1 1 1)

(1 2 3 4 5).

Should we try our usual strategy again?

We could. The function visits the elements of a tup, so it should follow the pattern for such functions:

```
(define sum-of-prefixes
  (lambda (tup)
    (cond
      ((null? tup) ...)
      (else ...
            (sum-of-prefixes (cdr tup))
            ...))))
```

What is a good replacement for the dots in the first line?

The first line is easy again. We must replace the dots with (**quote** ()), because we are building a list.

Then how about the second line?	The second line is the hard part.
Why?	The answer should be the sum of all the numbers that we have seen so far *cons*ed onto the natural recursion.
Let's do it!	The function does not know what all these numbers are. So we can't form the sum of the prefix.
How do we get around this?	The trick that we just saw should help.
Which trick?	Well, *two-in-a-row-b?* receives two arguments and one tells it something about the other.
What does *two-in-a-row-b?*'s first argument say about the second argument.	Easy: the first argument, *preceding*, always occurs just before the second argument, *lat*, in the original list.
So how does this help us with *sum-of-prefixes*	We could define *sum-of-prefixes-b*, which receives *tup* and the sum of all the numbers that precede *tup* in the tup that *sum-of-prefixes* received.

Let's do it!

```
(define sum-of-prefixes-b
  (lambda (sonssf tup)
    (cond
      ((null? tup) (quote ()))
      (else (cons (+ sonssf (car tup))
                  (sum-of-prefixes-b
                    (+ sonssf (car tup))
                    (cdr tup)))))))
```

Isn't *sonssf* a strange name?	It is an abbreviation. Expanding it helps a lot: *sum of numbers seen so far.*

What is the value of
  (*sum-of-prefixes-b sonssf tup*)
where *sonssf* is 0
and
  tup is (1 1 1)

Since *tup* is not empty, we need to determine the value of
  (*cons* 1 (*sum-of-prefixes-b* 1 *tup*))
where
  tup is (1 1).

And what do we do now?

We *cons* 2 onto the value of
  (*sum-of-prefixes-b* 2 *tup*)
where
  tup is (1).

Next?

We need to remember to *cons* the value 3 onto (*sum-of-prefixes-b* 3 *tup*)
where
  tup is ().

What is left to do?

We need to:
  a. cons 3 onto ()
  b. cons 2 onto the result of a
  c. cons 1 onto the result of b.
And then we are done.

Is *sonssf* a good name?

Yes, every natural recursion with *sum-of-prefixes-b* uses the sum of all the numbers preceding *tup*.

Define *sum-of-prefixes* using *sum-of-prefixes-b*

Obviously the first sum for *sonssf* must be 0:

```
(define sum-of-prefixes
  (lambda (tup)
    (sum-of-prefixes-b 0 tup)))
```

The Eleventh Commandment

Use additional arguments when a function needs to know what other arguments to the function have been like so far.

Do you remember what a tup is?	A tup is a list of numbers.
Is (1 1 1 3 4 2 1 1 9 2) a tup?	Yes, because it is a list of numbers.
What is the value of (*scramble tup*) where *tup* is (1 1 1 3 4 2 1 1 9 2)	(1 1 1 1 1 4 1 1 1 9).
(*scramble tup*) where *tup* is (1 2 3 4 5 6 7 8 9)	(1 1 1 1 1 1 1 1 1).
(*scramble tup*) where *tup* is (1 2 3 1 2 3 4 1 8 2 10)	(1 1 1 1 1 1 1 1 2 8 2).
Have you figured out what it does yet?	It's okay if you haven't. It's kind of crazy. Here's our explanation: "The function *scramble* takes a non-empty tup in which no number is greater than its own index, and returns a tup of the same length. Each number in the argument is treated as a backward index from its own position to a point earlier in the tup. The result at each position is found by counting backward from the current position according to this index."
If *l* is (1 1 1 3 4 2 1 1 9 2) what is the prefix of (4 2 1 1 9 2) in *l*	(1 1 1 3 4), because the prefix contains the first element, too.
And if *l* is (1 1 1 3 4 2 1 1 9 2) what is the prefix of (2 1 1 9 2) in *l*	(1 1 1 3 4 2).

Is it true that (*scramble tup*) must know something about the prefix for every element of *tup*

We said that it needs to know the entire prefix of each element so that it can use the first element of *tup* as a backward index to *pick* the corresponding number from this prefix.

Does this mean we have to define another function that does most of the work for *scramble*

Yes, because *scramble* needs to collect information about the prefix of each element in the same manner as *sum-of-prefixes*.

What is the difference between *scramble* and *sum-of-prefixes*

The former needs to know the actual prefix, the latter needs to know the sum of the numbers in the prefix.

What is (*pick n lat*)
where *n* is 4
and
lat is (4 3 1 1 1)

1.

What is (*pick n lat*)
where *n* is 2
and
lat is (2 4 3 1 1 1)

4.

Do you remember *pick* from chapter 4?

If you do, have an ice cream. If you don't, here it is:

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

Here is *scramble-b*

```
(define scramble-b
  (lambda (tup rev-pre)
    (cond
      ((null? tup) (quote ()))
      (else
        (cons (pick (car tup)
                    (cons (car tup) rev-pre))
          (scramble-b (cdr tup)
            (cons (car tup) rev-pre)))))))
```

How do we get *scramble-b* started?

A better question is: how does it work?

What does *rev-pre* abbreviate?

That is always the key to these functions. Apparently, *rev-pre* stands for reversed prefix.

If
tup is (1 1 1 3 4 2 1 1 9 2)
and
rev-pre is ()
what is the reversed prefix of
(*cdr tup*)

It is the result of *cons*ing (*car tup*) onto *rev-pre*: (1).

If
tup is (2 1 1 9 2)
and
rev-pre is (4 3 1 1 1)
what is the reversed prefix of
(1 1 9 2)
which is (*cdr tup*)

Since (*car tup*) is 2, it is
(2 4 3 1 1 1).

Do we need to know what
rev-pre is when
tup is ()

No, because we know the result of
(*scramble tup rev-pre*)
when *tup* is the empty list.

How does *scramble-b* work?

The function *scramble-b* receives a tup and the reverse of its proper prefix. If the tup is empty, it returns the empty list. Otherwise, it constructs the reverse of the complete prefix and uses the first element of tup as a backward index into this list. It then processes the rest of the tup and *cons*es the two results together.

How does *scramble* get *scramble-b* started?

Now, it's no big deal. We just give *scramble-b* the tup and the empty list, which represents the reverse of the proper prefix of the tup:

```
(define scramble
  (lambda (tup)
    (scramble-b tup (quote ()))))
```

Let's try it.

That's a good idea.

The function *scramble* is an unusual example. You may want to work through it a few more times before we have a snack.

Okay.

Tea time.

Don't eat too much. Leave some room for dinner.