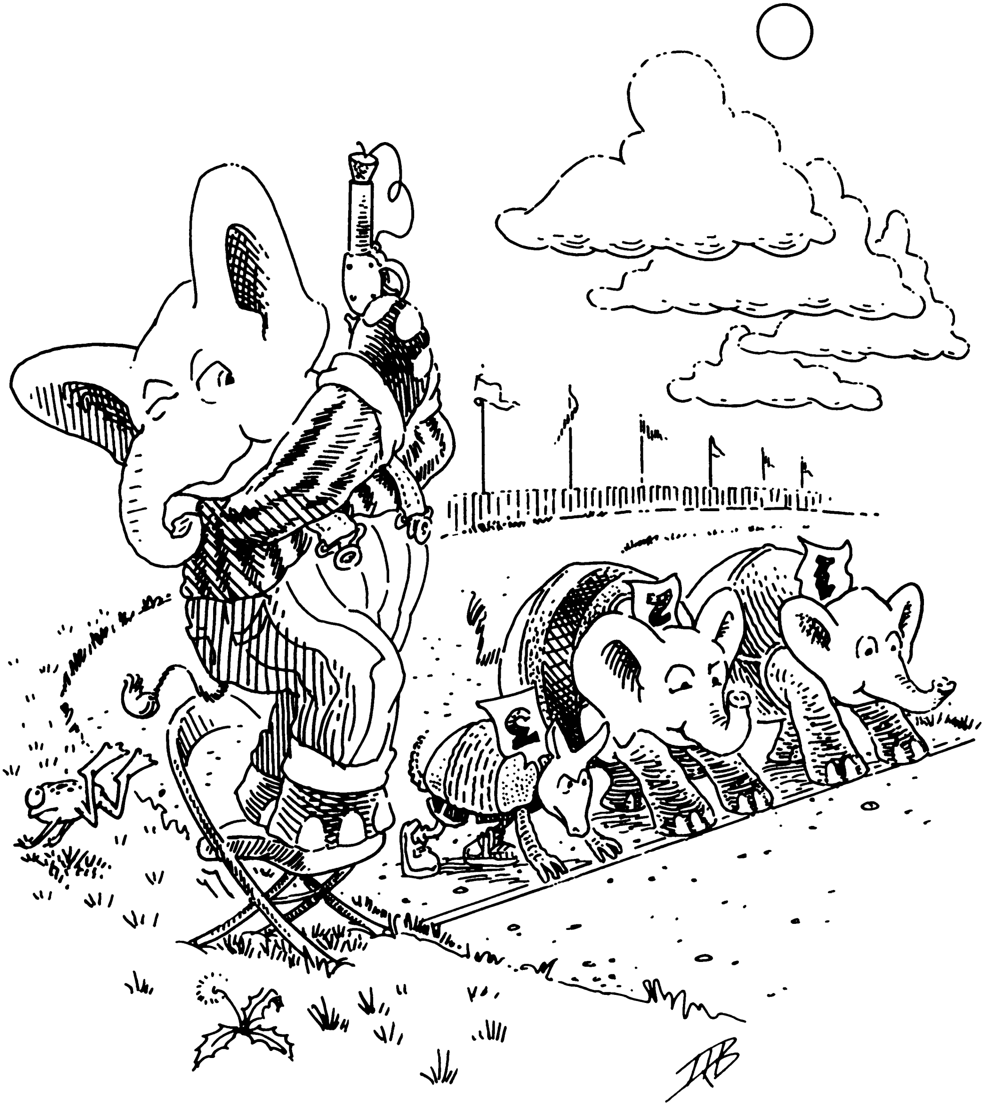
16.
Ready, Set, Bang!



Here are *sweet-tooth* and *last*

```
(define sweet-tooth
  (lambda (food)
    (cons food
      (cons (quote cake)
        (quote ())))))
```

```
(define last (quote angelfood))
```

More food: did you exercise after your snack?

What is the value of (*sweet-tooth* *x*) where *x* is chocolate

(chocolate cake).

What does *last* refer to?

angelfood.

What is the value of (*sweet-tooth* *x*) where *x* is fruit

(fruit cake).

Now, what does *last* refer to?

Still angelfood.

Can you write the function *sweet-toothL* which returns the same value as *sweet-tooth* and which, in addition, changes *last* so that it refers to the last *food* that *sweet-toothL* has seen?

We have used this trick twice before. Here we go:

```
(define sweet-toothL
  (lambda (food)
    (set! last food)
    (cons food
      (cons (quote cake)
        (quote ())))))
```

What is the value of
(*sweet-toothL* (**quote** chocolate))

(chocolate cake).

And the value of *last* is ...

chocolate.

What is the value of (*sweet-toothL* (**quote** fruit))	(fruit cake).
And *last*	It refers to fruit.
Isn't this easy?	Easy as pie!
Find the value of (*sweet-toothL* x) where x is cheese	It is (cheese cake).
What is the value of (*sweet-toothL* (**quote** carrot))	(carrot cake).
Do you still remember the ingredients that went into *sweet-toothL*	There was chocolate, fruit, cheese, and carrot.
How did you put this list together?	By quickly glancing over the last few questions and answers.
But couldn't you just as easily have memorized the list as you were reading the questions?	Of course, but why?
Can you write a function *sweet-toothR* that returns the same results as *sweet-toothL* but also memorizes the list of ingredients as they are passed to the function?	Yes, you can. Here's a hint. (**define** *ingredients* (**quote** ()))
What is that hint about?	This is the name that refers to the list of ingredients that *sweet-toothR* has seen.
One more hint: The Second Commandment.	Is this the commandment about using *cons* to build lists?

Yes, that's the one.

Here's the function:

```
(define sweet-toothR
  (lambda (food)
    (set! ingredients
      (cons food ingredients))
    (cons food
      (cons (quote cake)
        (quote ())))))
```

What is the value of (*sweet-toothR* x)
where
  x is chocolate

(chocolate cake).

What are the *ingredients*

(chocolate).

What is the value of
  (*sweet-toothR* (**quote** fruit))

(fruit cake).

Now, what are the *ingredients*

(fruit chocolate).

Find the value of (*sweet-toothR* x)
where
  x is cheese

It is (cheese cake).

What does the name *ingredients* refer to?

(cheese fruit chocolate).

What is the value of
  (*sweet-toothR* (**quote** carrot))

(carrot cake).

And now, what are the *ingredients*

(carrot cheese fruit chocolate).

Now that you have had the dessert ...

Is it time for the real meal?

Did we forget about The Sixteenth Commandment?	Sometimes it is easier to explain things when we ignore the commandments. We will use names introduced by (**let** ...) next time we use (**set!** ...).
What is the value of (*deep* 3)	No, it is not a pizza. It is (((pizza))).
What is the value of (*deep* 7)	Don't get the pizza yet. But, yes, it is (((((((pizza))))))).
What is the value of (*deep* 0)	Let's guess: pizza.
Good guess.	This is easy: no toppings, plain pizza.
Is this *deep*	It would give the right answers.

```
(define deep
  (lambda (m)
    (cond
      ((zero? m) (quote pizza))
      (else (cons (deep (sub1 m))
              (quote ()))))))
```

Do you remember the value of (*deep* 3)	It is (((pizza))), isn't it?
How did you determine the answer?	Well, *deep* checks whether its argument is 0, which it is not, and then it recurs.
Did you have to go through all of this to determine the answer?	No, the answer is easy to remember.

Is it easy to write the function *deepR* which returns the same answers as *deep* but remembers all the numbers it has seen?

This is trivial by now:

```
(define Ns (quote ()))
```

```
(define deepR
  (lambda (n)
    (set! Ns (cons n Ns))
    (deep n)))
```

Great! Can we also extend *deepR* to remember all the results?

This should be easy, too:

```
(define Rs (quote ()))
```

```
(define Ns (quote ()))
```

```
(define deepR
  (lambda (n)
    (set! Rs (cons (deep n) Rs))
    (set! Ns (cons n Ns))
    (deep n)))
```

Wait! Did we forget a commandment?

The Fifteenth: we say (*deep* ***n***) twice.

Then rewrite it.

```
(define deepR
  (lambda (n)
    (let ((result (deep n)))
      (set! Rs (cons result Rs))
      (set! Ns (cons n Ns))
      result)))
```

Does it work?

Let's see.

What is the value of (*deepR* 3)

(((pizza))).

What does *Ns* refer to?	(3).
And *Rs*	((((pizza)))).
Let's do this again. What is the value of (*deepR* 5)	(((((pizza))))).
Ns refers to ...	(5 3).
And *Rs* to ...	((((((pizza))))) (((pizza)))).

The Nineteenth Commandment

Use (set! ...) to remember valuable things between two distinct uses of a function.

Do it again with 3	But we just did. It is (((pizza))).
Now, what does *Ns* refer to?	(3 5 3).
How about *Rs*	((((pizza))) (((((pizza))))) (((pizza)))).
We didn't have to do this, did we?	No, we already knew the result. And we could have just looked inside *Ns* and *Rs*, if we really couldn't remember it.

How should we have done this?

Ns contains 3. So we could have found the value (((pizza))) without using *deep*.

Where do we find (((pizza)))

In *Rs*.

What is the value of (*find* 3 *Ns Rs*)

(((pizza))).

What is the value of (*find* 5 *Ns Rs*)

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

What is the value of (*find* 7 *Ns Rs*)

No answer, since 7 does not occur in *Ns*.

Write the function *find*
In addition to *Ns* and *Rs* it takes a number *n* which is guaranteed to occur in *Ns* and returns the value in the corresponding position of *Rs*

```
(define find
  (lambda (n Ns Rs)
    (letrec
      ((A (lambda (ns rs)
             (cond
               ((= (car ns) n) (car rs))
               (else
                 (A (cdr ns) (cdr rs)))))))
      (A Ns Rs))))
```

We are happy to see that you are truly comfortable with (**letrec** ...)

No problem.

Use *find* to write the function *deepM* which is like *deepR* but avoids unnecessary *cons*ing onto *Ns*

No problem, just use (**if** ...):

```
(define deepM
  (lambda (n)
    (if (member? n Ns)
       (find n Ns Rs)
       (deepR n))))
```

What is *Ns*

(3 5 3).

And *Rs*

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

Now that we have *deepM* should we remove the duplicates from *Ns* and *Rs*

How could we possibly do this?

You forgot: we have (**set!** ...)

```
(set! Ns (cdr Ns))
```

```
(set! Rs (cdr Rs))
```

What is *Ns* now?

(5 3).

And how about *Rs*

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

Is *deepM* simple enough?

Sure looks simple.

Do we need to waste the name *deepR*

No, the function *deepR* is not recursive.

And *deepR* is used in only one place.

That's correct.

So we can write *deepM* without using *deepR*

```
(define deepM
  (lambda (n)
    (if (member? n Ns)
        (find n Ns Rs)
        (let ((result (deep n)))
          (set! Rs (cons result Rs))
          (set! Ns (cons n Ns))
          result))))
```

This is another form of simplifying.	Which is why we did it after the function was correct.
If we now ask one more time what the value of (*deepM* 3) is	... then we use *find* to determine the result.
Ready? What is the value of (*deepM* 6)	((((((pizza)))))).
Good, but how did we get there?	We used *deepM* and *deep*, which *cons*ed onto *Ns* and *Rs*.
But, isn't (*deep* 6) the same as (*cons* (*deep* 5) (**quote** ()))	What kind of question is this?
When we find (*deep* 6) we also determine the value of (*deep* 5)	Which we can already find in *Rs*.
That's right.	Should we try to help *deep* by changing the recursion in *deep* from (*deep* (*sub1* *m*)) to (*deepM* (*sub1* *m*))?

Do it.

```
(define deep
  (lambda (m)
    (cond
      ((zero? m) (quote pizza))
      (else (cons (deepM (sub1 m))
                  (quote ()))))))
```

What is the value of (*deepM* 9)	(((((((((pizza))))))))).
What is *Ns* now?	(9 8 7 6 5 3).

Where did the 7 and 8 come from?

The function *deep* asks for (*deepM* 8).

And that is why 8 is in the list.

(*deepM* 8) requires the value of (*deepM* 7).

Is this it?

Yes, because (*deepM* 6) already knows the answer.

Can we eat the pizza now?

No, because *deepM* still disobeys The Sixteenth Commandment.

That's true. The names in (**set!** *Ns* ...) and (**set!** *Rs* ...) are not introduced by (**let** ...)

It is easy to do that.

Here it is:

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (lambda (n)
      (if (member? n Ns)
          (find n Ns Rs)
          (let ((result (deep n)))
            (set! Rs (cons result Rs))
            (set! Ns (cons n Ns))
            result)))))
```

What is the value of this definition?

Two imaginary names and *deepM*.

```
(define Rs1 (quote ()))
```

```
(define Ns1 (quote ()))
```

```
(define deepM
  (lambda (n)
    (if (member? n Ns1)
        (find n Ns1 Rs1)
        (let ((result (deep n)))
          (set! Rs1 (cons result Rs1))
          (set! Ns1 (cons n Ns1))
          result))))
```

What is the value of (*deepM* 16)

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

Here is what $\underline{Ns}_1$ refers to:

Our favorite food!

```
(16      ((((((((((((((((((pizza))))))))))))))))
 15       (((((((((((((((pizza)))))))))))))))
 14        ((((((((((((((pizza))))))))))))))
 13         (((((((((((((pizza)))))))))))))
 12          ((((((((((((pizza))))))))))))
 11           (((((((((((pizza)))))))))))
 10            ((((((((((pizza))))))))))
  9             (((((((((pizza)))))))))
  8              ((((((((pizza))))))))
  7               (((((((pizza)))))))
  6                ((((((pizza))))))
  5                 (((((pizza)))))
  4                  ((((pizza))))
  3                   (((pizza)))
  2                    ((pizza))
  1                     (pizza)
  0)                     pizza)
```

What does $\underline{Rs}_1$ refer to?

Doesn't this look like a slice of pizza?

What is (*find* 3 (**quote** ()) (**quote** ()))

This questions is meaningless. Neither $\underline{Ns}_1$ nor $\underline{Rs}_1$ is empty so *find* would never be used like that.

But what would be the result?

No answer.

What would be a good answer?

If *n* is not in *Ns*, then (*find n Ns Rs*) should be #f. We just have to add one line to *find* if we want to cover this case:

```
(define find
  (lambda (n Ns Rs)
    (letrec
      ((A (lambda (ns rs)
            (cond
              ((null? ns) #f)
              ((= (car ns) n) (car rs))
              (else
                (A (cdr ns) (cdr rs)))))))
      (A Ns Rs))))
```

Why is #f a good answer in that case?

When *find* succeeds, it returns a list, and #f is an atom.

Can we now replace *member?* with *find* since the new version also handles the case when its second argument is empty?

Yes, that's no problem now. If the answer is #f, *Ns* does not contain the number we are looking for. And if the answer is a list, then it does.

Okay, then let's do it.

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (lambda (n)
      (if (atom? (find n Ns RS))
          (let ((result (deep n)))
            (set! Rs (cons result Rs))
            (set! Ns (cons n Ns))
            result)
          (find n Ns Rs)))))
```

That's one way of doing it. But if we follow The Fifteenth Commandment, the function looks even better.

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (lambda (n)
      (let ((exists (find n Ns RS)))
        (if (atom? exists)
            (let ((result (deep n)))
              (set! Rs (cons result Rs))
              (set! Ns (cons n Ns))
              result)
            exists)))))
```

Take a deep breath or a deep pizza, now.

Do you remember *length*

Sure:

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

What is the value of

```
(define length
  (lambda (l)
    0))
```

```
(set! length
  (lambda (l)
    (cond
      ((null? l) 0)
      (else (add1 (length (cdr l)))))))
```

It is as if we had written:

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

But doesn't this disregard The Sixteenth Commandment? Aren't we supposed to use names in (**set!** ...) that have been introduced by (**let** ...)?

Here is one way to do it without using a name introduced by (**define** ...) in a (**set!** ...)

```
(define length
  (let ((h (lambda (l) 0)))
    (set! h
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (h (cdr l)))))))
    h))
```

And this one disregards the The Seventeenth Commandment: there is no (**lambda** ... between the
(**let** ((*h* ...)) ...)
and the
(**set!** *h* ...).

The Seventeenth Commandment

(final version)

Use (set! *x* ...) for (let ((*x* ...)) ...) only if there is at least one (lambda ... between it and the (let ...), or if the new value for *x* is a function that refers to *x*.

What is the value of

```
(define length
  (let ((h (lambda (l) 0)))
    (set! h
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (h (cdr l)))))))
    h))
```

It is as if we had written:

```
(define h1
  (lambda (l)
    0))
```

```
(define length
  (let ()
    (set! h1
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (h1 (cdr l)))))))
    h1))
```

True. Evaluating the definition creates an imaginary definition for *h* by removing it from the (**let** ...)

Yes, and the (**let** () ...) is now only used to order two events: changing the value of $\underline{h}_1$ and returning the value of $\underline{h}_1$.

What is the value of

```
(define h1
  (lambda (l)
    0))
```

```
(define length
  (let ()
    (set! h1
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (h1 (cdr l)))))))
    h1))
```

It is as if we had written:

```
(define h1
  (lambda (l)
    (cond
      ((null? l) 0)
      (else (add1 (h1 (cdr l)))))))
```

```
(define length
  (let ()
    h1))
```

What is the value of

```
(define length
  (let ()
    h1))
```

It is as if we had written:

```
(define length
  (lambda (l)
    (cond
      ((null? l) 0)
      (else (add1 (h1 (cdr l)))))))
```

Does this mean *length* would perform as we expect it to?

Yes, it would because it is basically the same function it used to be. It just refers to a recursive copy of itself through the imaginary name $\underline{h}_1$.

Okay, let's start over. Here is the definition of *length* again:

```
(define length
  (let ((h (lambda (l) 0)))
    (set! h
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (h (cdr l)))))))
    h))
```

Can you eliminate the parts of the definition that are specific to *length*

The right-hand side of (**set!** ...) needs to be eliminated:

```
(define length
  (let ((h (lambda (l) 0)))
    (set! h ...)
    h))
```

The rest could be reused to construct any recursive function of one argument.

Here is *L*

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

Can we use it to express the right-hand side of (**set!** ...) in *length*

That should be possible.

Is this a good solution?

```
(define length
  (let ((h (lambda (l) 0)))
    (set! h
      (L (lambda (arg) (h arg))))
    h))
```

Yes, except that (**lambda** (*arg*) (*h arg*)) seems to be a long way of saying *h*.

Why can we write
(**lambda** (*arg*) (*h arg*))

Because *h* is a function of one argument.

Does *h* always refer to
(**lambda** (*l*) 0)

No, it is changed to the value of
(*L* (**lambda** (*arg*) (*h arg*))).

What is the value of
(**lambda** (*arg*) (*h arg*))

We don't know because it depends on *h*.

How many times does the value of *h* change?

Once.

What is the value of
(*L* (**lambda** (*arg*) (*h arg*)))

It is a function:

```
(lambda (l)
  (cond
    ((null? l) 0)
    (else (add1
            ((lambda (arg) (h arg))
             (cdr l)))))).
```

What is the value of

```
(lambda (l)
  (cond
    ((null? l) 0)
    (else (add1
            ((lambda (arg) (h arg))
             (cdr l))))))
```

We don't know because *h* changes. Indeed, it changes and becomes this function.

And then?

Then the value of *h* is the recursive function *length*.

Rewrite the definition of *length* so that it becomes a function of *L*. Call the new function $Y_!$

```
(define Y!
  (lambda (L)
    (let ((h (lambda (l) (quote ()))))
      (set! h
        (L (lambda (arg) (h arg))))
      h)))
```

Thank you, Peter J. Landin.

Can you explain *Y-bang*

```
(define Y-bang
  (lambda (f)
    (letrec
      ((h (f (lambda (arg) (h arg)))))
      h)))
```

Here are our words:
"A **(letrec** ...) is an abbreviation for an expression consisting of **(let** ...) and **(set!** ...). So another way of writing $Y_!$ is *Y-bang*."[1]

[1] A **(letrec** ...) that defines mutually recursive definitions can be abbreviated using **(let** ...) and **(set!** ...) expressions:

```
(letrec
  ((x1 α1)
   ...
   (xn αn))
  β)
=
(let ((x1 0) ... (xn 0))
  (let ((y1 α1) ... (yn αn))
    (set! x1 y1)
    ...
    (set! xn yn))
  β)
```

The names $y_1 \ldots y_n$ must not occur in $\alpha_1 \ldots \alpha_n$ and they must not be chosen from the names $x_1 \ldots x_n$. Initializing with 0 is arbitrary and it is wrong to assume the names $x_1 \ldots x_n$ are 0 in $\alpha_1 \ldots \alpha_n$.

Write *length* using $Y_!$ and *L*

```
(define length (Y! L))
```

You have just worked through the derivation of a function called "the applicative-order, imperative Y combinator." The interesting aspect of $Y_!$ is that it produces recursive definitions without requiring that the functions be named by (**define** ...)
Define *D* so that *depth** is

```
(define depth* (Y! D))
```

```
(define D
  (lambda (depth*)
    (lambda (s)
      (cond
        ((null? s) 1)
        ((atom? (car s))
         (depth* (cdr s)))
        (else
          (max
            (add1 (depth* (car s)))
            (depth* (cdr s))))))))
```

How do we go from a recursive function definition to a function *f* such that ($Y_!$ *f*) builds the corresponding recursive function without (**define** ...)

Our words:
" *f* is like the recursive function except that the name of the recursive function is replaced by the name *recfun* and the whole expression is wrapped in
(**lambda** (*recfun*) ...)."

Is it true that the value of (*Y f*) is the same recursive function as the value of ($Y_!$ *f*)

Yes, the function $Y_!$ produces the same recursive function as *Y* for all *f* that have this shape.

What happens when we use *Y* and $Y_!$ with a function that does not have this shape?

Let's see.

Give the following function a name:

```
(define ...
  (let ((x 0))
    (lambda (f)
      (set! x (add1 x))
      (lambda (a)
        (if (= a x)
            0
            (f a))))))
```

How about *biz*, an abbreviation for bizarre?

That is as good a name as any other. What is the value of this definition?

It is as if we had written:

```
(define x1 0)
```

```
(define biz
  (lambda (f)
    (set! x1 (add1 x1))
    (lambda (a)
      (if (= a x1)
          0
          (f a)))))
```

What is the value of
((*Y* *biz*) 5)

It's 0.

What is the value of
(($Y_!$ *biz*) 5)

It's not 0. It doesn't even have an answer!

Does your hat still fit?

Of course it does. After you have worked through the definition of the *Y* combinator, nothing will ever affect your hat size again, not even an attempt to understand the difference between *Y* and $Y_!$.

Then again, eating some more scrambled eggs and pancakes may do things to you!

Something lighter, like Belgian waffles, would do it, too.

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*